UNIVERSITE DE NICE-SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

PhD THESIS

to obtain the title of

Doctor in Sciences

of the University of Nice-Sophia Antipolis
Specialty: Computer Science

presented and defended by

Corrado LEITA

SGNET

Automated protocol learning for the observation of malicious threats

Thesis advisor: Marc DACIER

defended on the 4th of December 2008

Reviewers :	Herve Debar	-	Orange Labs
	Vern Paxson	-	UC Berkeley
Advisor :	Marc Dacier	-	EURECOM
Members:	Christopher Kruegel	-	TU Vienna
	Engin Kirda	-	TU Vienna
	Mohamed Kaaniche	-	LAAS-CNRS
	Sotiris Ioannidis	-	FORTH

"...the speed was power,
and the speed was joy,
and the speed was pure beauty."
(Richard Backman, Jonathan Livingston Seagull)

A Pino e Luciana

Acknowledgments

This work is the final result of a three years period spent at EURECOM, on the French riviera. There are a lot of people that deserve all my gratitude for their continuous support, and for having made these three years a wonderful period that made me grow a lot not only as a researcher, but also as a person.

The first words cannot go to anybody else than my thesis advisor Marc Dacier. This work would not exist without his valuable guidance and without the long and inspiring brainstorming sessions that led to the definition of the concepts described in the following pages. He has always been available to listen and discuss about research challenges and help me to shape blurry ideas into something concrete and valuable. He has opened my mind by showing me different perspectives to address problems, and has helped me to develop a critical sense. He has driven me through the world of research, and has left me fascinated about it.

I would like to thank Vern Paxson, Professor at UC Berkeley, and Hervé Debar, researcher at Orange Labs, for having accepted to review this work, and for the insightful comments that greatly contributed to increase its research quality. I would also express my sincere gratitude to all the other members of the jury for having accepted to participate to the defense, even when such participation was at the cost of a really long trip. Thanks to Christopher Kruegel, Engin Kirda, Mohammed Kaaniche and Sotiris Ioannidis for honoring me with your presence in this day.

I would like to thank all the great researchers with who I have had the opportunity, and the honor, to work with. This includes the rest of the "Leurré.com team", Van-Hau Pham and Olivier Thonnard, our friends Andrew Clark and George Mohay, that made me to discover the marvels of Australia, as well as all the "International Secure System Lab", a great group of really brilliant minds.

Thanks to all my friends in EURECOM: to Patrick Petitmengin for being always ready to give me a hand with the various hardware failures and installations; to Gwenaelle Le Stir for the heroic patience with which she faced the great number of adventures involved in the administration of my thesis; to Christine Mangiapan, Christine Russel, Marie Laure Victorin, for having always been so efficient and available in helping organizing my trips. Last but not least, a special place in my memories goes to all the professors, students and post-docs that shared with me these years: Emilie Dumont, Guillaume Urvoy Keller, Pietro Michiardi, Daniele Croce, Damiano Carra, Melek Onen, and many others.

I give my most profound gratitude to my parents for their incredible support in all the vicissitudes of my life. The entirety of what I am is the result of their continuous help, support, and love, that never left me in any moment of my life, and that went unchanged through all my bad moods and behaviors. Thanks to my grandparents for all the help and love that they have always given to me, and a special thanks to Margherita.

I cannot conclude in any other way than with a very special thanks to the crazy inhabitants of "La Schtroumpfette". You have really been a second "french family"

and I probably shared with you some of the best moments of my last 25 years. Thanks to my dearest friends Alessandro and Marco, that made these three years much more fun that what they could have ever possibly been, and that have always been there to help and fight with each other for my own entertainment.

And finally, Xiaolan. She does not want to appear here, but she appears in every single word, figure and footnote of this work. She has literally shared with me every single second of these three years. She has been on my side in the days and nights spent in producing this work, with a constance and a loyalty that are, believe me, heroic. I'm not able to express in any way the amount of gratitude that I have for all that you did for me.

Un grosso grazie, di cuore, a tutti quanti.

Abstract

One of the main prerequisites for the development of reliable defenses to protect a network resource consists in the collection of quantitative data on the Internet threats. This attempt to "know your enemy" leads to an increasing interest in the collection and exploitation of datasets providing intelligence on network attacks. The creation of these datasets is a very challenging task. The challenge derives from the need to cope with the spatial and quantitative diversity of malicious activities. The observations need to be performed on a broad perspective, since the activities are not uniformly distributed over the IP space. At the same time, the data collectors need to be sophisticated enough to extract a sufficient amount of information on each activity and perform meaningful inferences. How to combine the simultaneous need to deploy a vast number of data collectors with the need of sophistication required to make meaningful observations? Such a challenge constitutes the foundations of this work.

We propose in this work the usage of protocol learning techniques for the automated generation of protocol interaction models. Such techniques aim at achieving automatically what is normally considered a tedious and time intensive manual task, especially when dealing with complex protocols or close-specification ones. Starting from the hypothesis that most of the network interaction handled by a honeypot is generated by deterministic attack tools, we propose a technique called ScriptGen. ScriptGen is able to infer information on protocol semantics from a set of samples and to build a representation of the interaction in the form of a Finite State Machine. The approach is protocol agnostic: no assumption is made on the structure of the protocol. The protocol structure is partially reconstructed through the application of bioinformatics techniques and through a set of inferences on the statistical variability of the input samples.

ScriptGen allows the automated construction of protocol emulators also for binary protocols whose manual analysis would be tedious. ScriptGen-based honeypots provide a high level of interaction with the attackers at very low cost for all the activities falling within the current protocol knowledge. We show how we are able to detect deviations from this knowledge, and take advantage of a proxying algorithm to dynamically react to them by producing refinements of the Finite State Machine.

We exploit the characteristics of the ScriptGen approach to design and implement a distributed honeypot deployment, SGNET. Coupling the ScriptGen learning with memory tainting techniques and with a simple shellcode emulator, we show how we are able to enable SGNET to emulate code injection attacks, downloading malware samples. We integrate the data collected by honeypot sensors deployed in 23 different testing sites to build a centralized dataset. The dataset is enriched with the output of different analysis tools providing different perspectives on the collected data.

The value of the resulting dataset is twofold. Firstly, it provides us information on the behavior of the ScriptGen learning technique when dealing with Internet attacks. We are able to validate the approach, showing its capability to correctly carry on the network interaction and, at the same time, achieve a very high scalability. Secondly, it is a rich and valuable source of intelligence on the structure of code injection attacks and on the malware propagation techniques. We propose a simple clustering algorithm to explore the complexity of the interrelationships among the different stages of a code injection attack and we evaluate its potential in studying the propagation strategy of modern malware.

Abstract

Un des préalables au développement de défenses fiables pour la protection d'un réseau informatique est la collection de données quantitatives sur les menaces qui le visent depuis l'Internet. Ce besoin de « connaître notre ennemi » induit un intérêt croissant pour la collecte et l'exploitation des informations sur les activités malveillantes observables. La création de bases de données recensant ces événements n'est pas une tâche facile. En effet, il faut pouvoir tenir compte de la diversité quantitative et spatiale des attaques. La collecte des données doit être pratiquée sur une grande échelle car les sources et destinations d'attaques ne sont pas uniformément réparties sur l'espace des adresses IP. En même temps, les techniques de collecte de données doivent être suffisamment sophistiquées pour extraire une quantité suffisante d'informations sur chaque activité et permettre des déductions sur les phénomènes observés. Il faut donc pouvoir déployer un grand nombre de capteurs et chacun de ces capteurs doit être à même de fournir des informations riches. Ce travail propose une solution qui concilie facilité de déploiement et richesse de collecte.

Partant du postulat que la plupart des activités observables par un pot de miel sont générées par des outils d'exploitation de vulnérabilités déterministes, nous proposons une technique automatique d'apprentissage des protocoles d'attaque, appelée ScriptGen. ScriptGen est en mesure de déduire la sémantique des protocoles à partir d'un ensemble d'échantillons de ce protocole et de construire une représentation de l'interaction sous la forme d'une machine a états finis. L'approche est agnostique par rapport à la sémantique des protocoles : aucune supposition n'est faite quant à sa structure ou sémantique. La structure du protocole est partiellement reconstruite en utilisant les techniques de bio-informatique et par le biais d'une série de déductions statistiques sur la variabilité des échantillons.

ScriptGen permet également la construction automatique d'émulateurs pour des protocoles difficiles a analyser manuellement, par exemple parce que leurs messages ne sont pas sous forme de caractères ASCII. Les pots de miel construits grâce à ScriptGen assurent un niveau élevé d'interaction avec les clients malveillants à un coût très bas pour toutes les activités déjà représentées dans la machine à états finis. Nous montrons comment nous sommes en mesure de répondre à des écarts par rapport aux connaissances actuellement représentées et, en utilisant un algorithme de proxy, de réagir dynamiquement et affiner progressivement la machine à états finis.

Nous utilisons les fonctionnalités de ScriptGen pour définir et mettre en place un système distribué de pots de miel, SGNET. En combinant les capacités d'apprentissage de ScriptGen avec des techniques de "memory tainting" et un simple émulateur de code shell, nous montrons comment SGNET est capable d'imiter les attaques d'injection de code jusqu'au téléchargement du malware. Nous intégrons les informations recueillies à partir de 23 pots de miel dans une base de données. Cette base est enrichie grâce à différentes techniques d'analyse qui offrent différentes perspectives sur les données recueillies.

La valeur de la base de données résultante est double. D'une part, la base fournit des informations sur le comportement des techniques d'apprentissage de ScriptGen vis à vis de la multiplicité des activités malveillantes sur Internet. Cette information nous aide à valider ScriptGen, montrant sa capacité à poursuivre avec succès l'interaction avec les clients et, dans le même temps, à avoir une bonne extensibilité. Par ailleurs cette base de données est une source précieuse d'informations sur les familles d'attaques par injection de code et sur les techniques de propagation du malware. Nous proposons un simple algorithme de clustering pour mettre en relief la complexité des relations entre les différentes étapes d'une attaque par injection de code.

Contents

Chapter 1

Introduction

Security is a very wide concept comprising many different perspectives. One of the most predominant threats in the recent years is the outbreak of self-propagating malware. Self-propagating malware is a concept that is now 20 years old and that started with the outbreak of the Morris worm in 1988 [Spafford 1989]. The spread of worms such as Blaster [URL 8] and Slammer [URL 38] now coexists with the growth of bot-infected hosts whose behavior is coordinated by a Command and Control channel [Baecher 2005].

The security community has developed a wide number of initiatives aiming at monitoring and observing these threats in order to build adequate defenses. Speculations have been recently carried out on the increased efficiency of the attackers in compromising vulnerable hosts and increasing the size of their botnets [URL 5]. Figure 1.1 plots the evolution of the number of bots monitored by the ShadowServer foundation [URL 30] in the period between September 2007 and September 2008. The ShadowServer foundation discovers and monitors IRC-based Command and Control channels by directly joining the IRC channel and collecting data. Such numbers show a constant rise in the population of the monitored botnets that started in January 2008 and seems to indicate an increase in aggressiveness of the propagation methods used by attackers. This goes in parallel with an increase of the sophistication of the techniques used by malware to conceal itself from analysis and detection [Nazario 2007] and the exponential increase of the number of different malware variants [Turner 2008].

Unfortunately information on these threats is too often either based on pro-

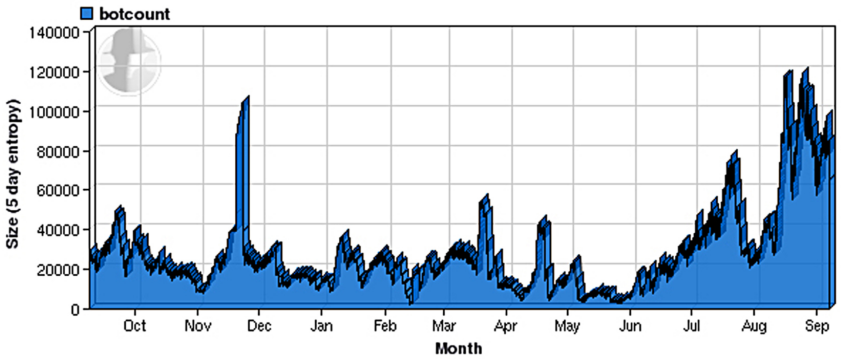

Figure 1.1: Botnet size evolution (September 2007 - September 2008)

prietary data or even anecdotal. This situation leads to an increasing interest in the collection of information to understand and quantify the propagation strategies used by modern malware. How does malware propagate? Is the increase in efficiency and sophistication leading also to the usage of more sophisticated exploitation techniques? What kind of vulnerabilities are effectively exploited to compromise victim hosts?

1.1 Problem statement

Answering these questions is an extremely complex task that implies addressing three main problems: application diversity, spatial diversity and depth of the observations.

1.1.1 Application diversity

The Internet is today an extremely complex setting as a consequence to the sophistication of the applications and the protocols taking advantage of it. The increasing popularity of rich web applications based on complex frameworks and the increasing number of clients for network applications such as, for instance, VoIP, opens an extremely diverse number of vulnerabilities to attackers and of ways to exploit them. According to [Turner 2008], 2134 vulnerabilities were discovered in the second half of 2007, 73% of which are considered easily exploitable. The applications affected by these vulnerabilities range from web applications, to web browsers, other clients, servers, and local applications. Each application class has different characteristics that can be exploited in different ways from attackers. The required process to monitor malicious activities targeting these vulnerabilities is thus specific to each application class.

This work addresses this challenge by selecting a specific class of activities, server-based exploitation attempts, and by developing a protocol learning technique allowing to cope with the diversity of the protocols involved in these activities. Differently from existing work, we avoid any *a priori* assumption on the nature of the expected observations and we aim at being protocol-agnostic.

1.1.2 Spatial diversity

Different research work aiming at the collection of data on unsolicited network traffic [Dacier 2004a, Cooke 2004] showed that different Internet blocks are affected by highly dissimilar profiles. The generation of a dataset starting from the observations of a single sensor in a single network location leads to the generation of data representative for that specific network block, but not representative of the global scenario of Internet attacks.

In order to cope with the spatial diversity of the activities, we need to distribute our observation points as much as possible over the IP space. For this to be possible,

the deployment and maintenance cost of each data collection sensor must be kept to a minimum.

1.1.3 Observation depth

The collected information needs to be sufficiently rich to perform reliable inferences on the nature and the root causes of the observations. For instance, simple aggregate information on high-level network events is not sufficient to correctly diversify the different activity types. As identified in [Yegneswaran 2004], only by engaging attackers into sufficiently long conversations it is possible to discriminate among different types of activity.

1.2 Research objectives

This work aims at addressing the previously introduced challenges for a specific class of activities, server-based exploitation attempts through blind scanning of the Internet.

A technique that proves to be very effective in monitoring and collecting data on this class of attacks consists in the usage of honeypots. Honeypots, formally introduced in Section 2.1.1, are network resources whose only purpose is being contacted by attackers, unaware of the network topology and thus unaware of their nature. Honeypots thus have an intrinsic capacity to collect suspicious traffic only. Despite the widespread usage of such techniques for the collection of data on suspicious activities, a main tradeoff exists between the representativeness of the collected data and its richness.

> **Claim:** Current approaches fail in correctly addressing the tradeoff between the requirements of representativeness and richness for the collected data.

This claim, addressed in depth in Chapter 2, can be articulated in these two points.

- In order to generate a representative dataset, the observation point chosen by the honeypot must be as global as possible. This led to the generation of distributed honeypot deployments such as Leurré.com [Pouget 2006], in which honeypot platforms are hosted by volunteering partners interested in exploiting the collected data.
- The distributed nature of these honeypot deployments has a direct impact on the richness of the collected data. Honeypot techniques allowing to be sufficiently interactive with the attackers to produce rich data are either too costly to cope with the distributed nature of the architecture or they are based on a set of assumptions on the nature of the observed attacks that compromises the representativeness of the dataset.

This work aims at addressing this trade-off.

Assumption. Self-propagating malware, that is any malicious code autonomously spreading to computer system through remote exploitation of vulnerabilities, is the main contributor to server-based code injection attacks observable on the Internet. The corresponding network interaction is generated by deterministic exploit scripts whose interaction with the vulnerable service follows a deterministic pattern.

This work builds upon this assumption to generate an automated protocol analysis method, called ScriptGen, able to learn from a set of samples of network interaction the underlying structure by partially inferring the protocol semantics. The approach is protocol agnostic: no assumption is made on the protocol semantics or on its structure, avoiding as much as possible to bias the corresponding observations. The generated responders take advantage of a Finite State Machine representation in order to interact with attacking clients at a very low cost, allowing the distributed deployment of low-cost sensors able to extract information on the nature of the conversations.

Thesis statement. In this thesis, we aim to demonstrate that:

- Automated learning of the protocol interaction is possible, and allow the generation of responders able to correctly handle future instances of a given type of observed activities.
- Automated learning can be used to incrementally increase the knowledge on protocol interactions and react to previously unknown activities.
- Automated learning allows honeypots to cheaply handle known activities and rely on more costly techniques only to learn unknown ones. The learning process allows the achievement of a very high degree of scalability at a reasonable cost in terms of complexity.
- The increased level of interaction of the generated honeypots can be coupled with memory tainting techniques and shellcode emulation to fully emulate code injection attacks and download malware samples.
- Such techniques can be used to generate a valuable and rich dataset providing valuable information on the propagation strategies of self-propagating malware and on the lifetime of the threats.

The effort in addressing the above points led to the generation of SGNET, a distributed honeypot deployment based on protocol learning techniques. As of

today, the deployment consists of 23 sensors deployed in different networks in America, Europe, Asia and Australia. The deployment is open to any institution wishing to take advantage of the generated dataset.

Chapter 2

State of the art

Contents

The collection of data on Internet threats has been widely addressed by the research community. We can distinguish two classes of contributions addressing the problem. On the one hand, we can identify in the literature different approaches aiming at the collection of data on malicious threats taking advantage of the concept of *honeypot*. On the other hand, different architectures are proposed to practically collect such information from the Internet. In this Chapter we overview such techniques assessing their strengths and weaknesses. In Section 2.1 we introduce the concept of honeypot, and we overview its different implementations. In Section 2.2 we review the different infrastructures and deployments that are being used to collect data on malicious threats on the Internet.

2.1 Retrieving information on malicious activities

In order to collect data on malicious activities, we need to take advantage of a data collection tool able to observe and characterize such activities.

2.1.1 Honeypots

The concept of honeypot has been investigated in various forms since 1990 [Cheswick 1990]. In [Halme 1995], the authors introduce the idea of "intrusion deflection" to attract attackers towards "a specially prepared, controlled environment for observation". This concept has been formalized by L. Spitzner in [Spitzner 2002] through the following definition:

> "A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource."

This definition groups together any network resource whose value consists in interacting with malicious users, eventually up to the point of being exploited and compromised. The practical implementation of a honeypot depends on a set of variables, the most important of which is the nature of the activity to be observed. A honeypot can aim at observing the behavior of network servers when contacted by malicious users (server-side honeypot). Instead, a honeypot can aim at observing the behavior of client applications when interacting with malicious remote services (client-side honeypot). This work focuses on the first of these two classes.

Most of the implementations of server-side honeypot solutions emulate the presence of network hosts on a set of unused IP addresses (also known as *dark space*). These addresses can belong to a company network as a mean to detect internal intrusions and anomalies, or can be assigned to routable IP addresses when the objective is to observe Internet worldwide malicious activities. The usage of unassigned IP addresses filters out benign traffic and allows focusing on the malicious one, since no benign user is supposed to be contacting these hosts. On the other hand, this biases the type of malicious network activities observable by these honeypots: only untargeted attacks blindly scanning an IP range comprising the honeypot address will be detected by the honeypot. This has two important consequences:

- An attack scanning a portion of IP space different from that to which a honeypot belongs is undetectable to that honeypot.
- More sophisticated targeted attacks aiming at the exploitation of a *specific* network resource considered valuable to the attacker are not visible to honeypots.

These limitations must be taken into account when exploiting server-side honeypots to collect data on Internet network attacks. On the one hand, the scope achievable by these techniques is in fact implicitly biased towards a class of attacks and attackers. On the other hand, a single honeypot deployed on a specific subnet of the Internet is likely to have an insufficient perspective on Internet attacks. We will see in Section 2.2.2 how this need led to the generation of distributed deployments to achieve a global view over Internet activities.

Despite the basic architectural similarities previously identified, a number of different server-based honeypot implementations exist in the literature. These implementations differ in the solutions adopted to emulate the presence of a network host in a network. Spitzner [Spitzner 2002] breaks down the different solutions according to the *level of interaction* with the attacker, talking about *low-interaction* and *high-interaction* solutions. Sections 2.1.2 and 2.1.3 provide an overview on the most significant solutions belonging to these two classes.

2.1.2 Low interaction honeypots

Low interaction honeypots aim at emulating the presence of a host through scripts that emulate the presence of different network services bound to the honeypot IPs. The emulation of a low-interaction honeypot can be as simple as a script binding a socket to different ports and closing connections as soon as they are established:

The degree with which a low interaction honeypot emulates a network host can vary significantly among different honeypot implementations. The different *depth* of these solutions offers different perspectives on the observed attacks, generating a trade-off between the complexity of the solution and the richness of the collected information.

We can find in the literature simple low-interaction implementations that do not aim at emulating protocol interaction. For instance, the Deception Toolkit by Cohen [Cohen 1998] aims at dissuading attackers from hitting a given machine by emulating the behavior of a vulnerable system. The LaBrea honeypot [URL 18] actively tries to *damage* the attacking clients by stalling their network connections.

The Tiny Honeypot [URL 4] developed by G.Bakos is a very simple example of protocol emulation. In this approach, a single daemon is generated taking advantage of *xinetd* and the various connection attempts are redirected towards the honeypot daemon taking advantage of netfilter rules. Any connection attempt on any port is presented with a login banner and a root shell, in the hope of collecting information on the intent of the attacker.

The attempt to increase the flexibility and the quality of the emulation has led to a set of more complex solutions described in the following Sections.

2.1.2.1 Honeyd

Honeyd [Provos 2004] by Niels Provos is an extremely flexible low-interaction honeypot that aims at emulating entire networks of honeypots.

Honeyd is able to emulate virtual networks composed of thousands of nodes, and to create complex topologies characterized by multiple routers and links with different latencies. Honeyd provides advanced features, such as the ability to take advantage of DHCP to allow the coexistence of honeypots with real hosts within production networks, or to receive packets through GRE tunnels.

Honeyd focuses on the emulation of all the layers of the network stack of a virtual host. Honeyd implements a *personality engine*, which modifies every packet generated by the virtual host in order to make it comply with the behavior of a particular TCP/IP stack implementation. A number of OS fingerprinting tools [URL 48, URL 1] take advantage of the diversities in the implementation of the TCP/IP stack and in the flags being used to detect the nature of a given operating system. Honeyd modifies the generated packets in order to comply with these heuristics and emulate the *personality* of the different virtual hosts.

While the personality engine is in charge of emulating the transport and link layer of the network stack, the emulation of the application protocols is left to a set

of external plugins in charge of the emulation of the protocol interaction. Honeyd supports three different techniques to provide emulators for application protocols:

- **Honeyd Service scripts.** It is possible to associate an executable file (such as a bash/perl/... script) to a given port on a given honeypot profile. Whenever a connection is established, an instance of the script is invoked receiving the attacker input on standard input. Any output generated by the process is sent back to the attacker.

- **Python services.** Honeyd provides an interface to generate python modules associated with the emulation of the protocol interaction on a given TCP port following a non-blocking I/O model. These modules are loaded within the honeyd core and do not require the instantiation of additional processes, and are thus more efficient. Each service is associated with a specific port, preventing the emulation of complex services spanning on different ports (e.g. the FTP protocol).

- **Subsystems.** It is possible to execute external Unix applications within the honeypots virtual address space. This is achieved using dynamic library preloading, intercepting the normal network libc calls and replacing them with their a custom honeyd implementation.

Despite the flexibility offered by honeyd in interfacing to emulators for application-level protocols, the availability of good emulation scripts for the different protocols is scarse. Looking at the honeyd website [URL 29], it is possible to find a few scripts that provide basic emulation for ASCII protocols (such as telnet or IIS services). More complex protocols, such as the NetBios protocols, cannot be easily emulated with Honeyd and require the usage of full-fledged implementations running as subsystems.

2.1.2.2 Nepenthes

Nepenthes [Baecher 2006] and its python-based counterpart Amun [URL 13] focus on a specific class of network activities, code injection attacks. We have seen that Honeyd limits its emulation capabilities to the network stack of a host and focuses on the collection of network-level information. Nepenthes offers a more sophisticated emulation of application-level protocols, even for binary ones. The emulation task is although restricted to a set of specific network interactions that lure self-propagating malware into propagating to the honeypot. This task-specific emulation allows Nepenthes to collect malware samples.

Nepenthes' implementation is based on a set of plugins organized on different functional layers.

- **Vulnerability modules.** Nepenthes vulnerability modules can be incorporated into Honeyd's service scripts. Each vulnerability module emulates the

application-level interaction associated with a specific exploit. These modules can be created by reverse engineering an exploit source code. They provide sufficient network interaction for luring the attacker into carrying on his exploit and retrieve a sample of shellcode.

- **Shellcode parsing modules.** During an interaction with an attacker, in case of successful exploit the vulnerability modules identify the shellcode within the protocol stream. The shellcode parsing modules apply heuristics on the binary shellcode sample in order to "understand" its behavior. The heuristics are very simple, and consist in applying an XOR decoder to reverse a common obfuscation technique used by malware authors, and in taking advantage of a set of patterns to detect common shellcode strategies. When needed, this stage can take advantage of a *shell emulator* to further interact with the attacker and receive information from it. The final output of this stage consists in the retrieval of a URL representing the location of the malware to be downloaded.

- **Fetch modules.** If the shellcode parsing stage has succeeded in understanding the location of the malware, Nepenthes takes advantage of a set of plugins to emulate the protocol interaction. This ranges from FTP/HTTP downloads to malware-specific protocols, such as *creceive*. The final output of this stage corresponds to a malware sample.

- **Submission modules.** These modules process the successfully downloaded files, eventually storing them locally or submitting them to remote collection services.

It is clear from this preliminary description that Nepenthes bases its observations on an *a priori* knowledge of the attack processes and techniques. The level of interactivity of the vulnerability modules, for instance, is kept at the lowest possible level known to ensure interactivity with the attacking clients. For instance, follows a code snippet extracted by the proof-of-concept exploit developed by a Russian hacker called *HouseOfDabus* [URL 22]. The exploit targets the vulnerability MS04-011 [URL 20], classified under CVE-2003-0533 [URL 24], a critical vulnerability allowing arbitrary execution of code through communication with the Microsoft LSASS service on port 445.

```
if (send(sockfd, req1, sizeof(req1)-1, 0) == -1) {
        printf("[-] Send failed\n");
        exit(1);
}
len = recv(sockfd, recvbuf, 1600, 0);
```

The script sends each exploit packet and then waits for an answer from the attacked client. In order to maximize efficiency, the exploit code does not check the received answer, and simply receives and discards an amount of data that is smaller than 1600 bytes. As a consequence to this choice, the Nepenthes vulnerability module associated with this vulnerability replies to any incoming packet with a

binary blob of 64 random bytes. A simple modification of the previously mentioned attack script would easily allow the detection of a Nepenthes honeypot and thus prevent the disclosure to security researchers of the malware sample.

The limitations of the knowledge-based approach used by Nepenthes have appeared in a recent paper [Zhuge 2007] by Zhuge et al. In this paper, the malware collection capabilities of Nepenthes have been compared with those of high interaction honeypots, running the full implementation of an OS and instrumented to detect a successful infection and react to it. The difference in collection ability of the two approaches is striking: of 171 malware families observed by the high interaction honeypots, only 61 were correctly observed and downloaded by Nepenthes.

2.1.2.3 Honeytrap

Honeytrap [URL 47], developed by Tillman Werner, is a low-interaction honeypot aiming at maximizing the probability of observing a malicious behavior. Differently from other approaches, honeytrap is not bound a priori to a set of ports. It takes advantage of sniffers or user-space hooks in the netfilter library [Welte 2000] to detect incoming connections and bind consequently the required socket. Each inbound connection can be handled according to 4 different operation modes:

- **Service emulation.** It is possible to take advantage of responder plugins similarly to the previously analyzed solutions.
- **Mirror mode.** When enabling mirror mode for a given port, every packet sent by an attacker to that port is simply mirrored back to the attacker. An attacker attempting an exploit over the honeypot will see its exploitation attempt to be mirrored back to him by the honeypot. This mode is based on the assumption that, in case of a self-propagating worm, the attacker must be exposed to the same vulnerability that he is trying to exploit. It is although unclear whether such "active" honeypot behavior is legally allowed.
- **Proxy mode.** Honeytrap allows proxying of all the packets directed to a port or set of ports to another host, such as a high interaction honeypot.
- **Ignore mode.** Used to disable TCP ports that should not be handled by honeytrap.

Honeytrap takes advantage of a set of plugins to exploit the information collected by the network interaction. The ability of these plugins to handle network attacks can be assimilated to a *best effort* service. For instance, if an HTTP URL appears in the network stream, honeytrap will try to download the file located at that URL. If the HTTP URL is not directly present in the network stream, but is embedded within an obfuscated shellcode, honeytrap will not be able to detect it or download it. Honeytrap visibility on the network attacks is thus not uniform, but heavily depends on their structure and their complexity.

2.1.2.4 Billy Goat

Billy Goat [Riordan 2006] is a particular honeypot developed by IBM Zurich Research Labs that focuses on the detection of worm outbreaks in enterprise environments. It is thus called by the authors Worm Detection System (WDS) in opposition to classical Intrusion Detection Systems. Billy Goat automatically binds itself to any unused IP address in a company network, and aims at quickly detecting and identifying the infected machines and retrieve information on the type of activity being performed.

In order to gather as much information as possible on the kind of activity observed by the sensors, Billy Goat employes responders that emulate the application level protocol conversation. While the general responder architecture is very similar to that employed by Honeyd, Billy Goat takes advantage of a peculiar solution for the emulation of SMB protocols, which consists in taking advantage of a hardened version of the open-source implementation of the protocol[1]. Such an implementation is derived from the work done by Overton [Overton 2003], which takes advantage of a similar technique to collect worms trying to propagate through open SMB shares. This choice puts Billy Goat in a hybrid position between low and high interaction techniques, since it offers to attacking clients the real protocol implementation, even if hardened to reduce security risks. The increased level of interaction of this technique has allowed interesting analyses such as the work done by Zurutuza in [Zurutuza Ortega 2007]. In this work, the author takes advantage of a data mining tool [Julisch 2003] to automatically generate signatures for unknown events observed by the Billy Goat system.

2.1.2.5 Stateless approaches

All the previous methods keep track of the connection attempts performed by different attackers with levels of state of various complexity. For instance, in the case of honeyd every established connection leads to the instantiation of the responder associated with the service in order to carry on the application-level conversation. The scalability of these approaches is thus affected by the rate of the established connections and, implicitly, by the size of the monitored subnet.

iSink [Yegneswaran 2004] and HoneyTank [Vanderavero 2004, Vanderavero 2008] address the aforementioned issue by implementing a completely stateless approach. The response to each incoming packet is solely based on the information contained in that packet and no additional state is required. This approximation proved to be sufficient to carry on the conversation with the attackers long enough to discriminate among different types of activities. Of course, such approximation can be easily detected by human interaction or by ad-hoc scripts testing the correct behavior of the protocol. For instance, a stateless SMTP responder will always accept a *DATA* command even when not preceded by a *MAIL FROM* or *RCPT TO* commands.

[1] www.samba.org

These stateless approaches achieve a significant level of interactivity with the client in a very scalable way. For instance, iSink [Yegneswaran 2004] implements responders for most of the Windows protocols (CIFS/SMB/NetBIOS/...) and achieves a scalability shown to be sufficient for the emulation of entire Class A networks.

2.1.3 High interaction honeypots

High interaction honeypots take advantage of the full implementation of an OS to carry on the conversation with the attacker. Virtualization systems such as VMware [URL 46] or Qemu [Bellard 2005] are often used as a convenient way to emulate on a single physical host multiple instances of an OS binding them to multiple IP addresses.

Taking advantage of real operating systems running in virtualization environments, high interaction honeypots allow the maximum possible level of interaction with the attacker. Differently from low-interaction techniques, protocol emulation is no longer an issue. A new issue arises: *containment*. High-interaction techniques offer to the attacker a real environment, which is vulnerable and can thus be exploited. Containment techniques try to prevent the attacker from exploiting the honeypot for illicit purposes without compromising the quality of the collected information.

2.1.3.1 Network monitoring

A possible way of monitoring the status of a high interaction honeypot consists in monitoring the network traffic that it generates.

Honeynet project's Honeywall [The Honeynet Project 2005b, The Honeynet Project 2005a] is normally set up as a transparent bridge between the farm of high interaction honeypots and the Internet. The containment techniques employed by Honeywall combine together two different techniques.

Firstly, Honeywall takes advantage of a set of iptables [URL 26] rules to limit the ability of the attacker to take advantage of the bandwidth available to the honeypot. The default rules are extremely strict, and allow the honeypot to generate only 20 TCP connections, 20 UDP packets, 50 ICMP packets per hour. This prevents the attacker from being able to run Denial of Service (DoS) attacks against other victims.

While these rules are effective in preventing these kinds of attacks, a single TCP connection can be enough for an attacker to compromise another victim using the honeypot as a stepping stone. In order to address this class of attacks, Honeywall takes advantage of Snort-inline. Snort-inline is a modification of the popular Intrusion Detection System Snort [Roesch 1999] to allow the modification of packets that match a given set of rules. Honeywall uses this tool to perform a *sanitization* of the exploits: the network bytes known to be essential for the exploit to succeed are replaced with innocuous ones. The purpose of the sanitization technique is to prevent the attacker from succeeding in the exploit, and at the same time invite him to try again, eventually with different exploits or different exploit configurations.

While a solution such as Honeywall can be very effective in most cases, it is important to understand that it is a *knowledge-based* solution. The Snort-inline signatures used by Honeywall are able to sanitize only those exploits that follow known attack patterns. The signatures need to be kept constantly up to date and might not provide an exhaustive protection against all the known attacks either. Moreover, it is possible for an attacker having a specific interest in taking advantage of the honeypot to study these signatures and find a way to evade them.

An interesting network monitoring technique was developed by Georgia Tech, the BotHunter [Gu 2007] tool. BotHunter is built upon Snort and specifically aims at the detection and the collection of data on infections of self-propagating malware. Its nature is thus comparable to a special-purpose Intrusion Detection System (IDS), but the authors show in [Gu 2007] how BotHunter can be coupled with a honeynet and be used as a data collection and containment mechanism. BotHunter analysis is based on a high level model of a self-propagating malware, with special interest to IRC-based bots. Such model depicts a malware infection as composed of 5 different stages:

- E1: External to Internal Inbound Scan
- E2: External to Internal Inbound Exploit
- E3: Internal to External Binary Acquisition
- E4: Internal to External C&C Communication
- E5: Internal to External Outbound Infection Scanning

BotHunter tries to independently detect each of these 5 stages, and takes advantage of correlation techniques to infer the presence of a malware infection. In case an infection is detected, BotHunter is able to generate detailed reports on the infection process, along the 5 detection stages, such as information on the exploited TCP port, the malware binary being pushed to the victim, and information on the eventual Command & Control channel. BotHunter does not prevent malware from trying to propagate to other hosts: differently from Honeywall, outbound infection scanning is not blocked. The detection is focused on a behavioral model rather than exploit-specific signatures. Through such behavioral detection, BotHunter can be used to implement a more effective containment technique with respect to the previously analyzed solutions. In [Gu 2007] the authors react to a detected infection by tainting the corresponding high interaction honeypot and then by sanitizing it.

2.1.3.2 Host monitoring

High interaction honeypots, with their higher level of interactivity, provide rich information on the attacker activity. This level of interaction translates into verbose network information, but also in detailed information on the modifications performed by the attacker at *host level*. This information can be exploited to have

exact information on the success of attacks and on the status of the honeypot to implement a very reliable containment.

Sebek [URL 42] was developed by the Honeynet Project and is used in several high-interaction honeypot deployments [The Honeynet Project 2005b, Vrable 2005]. Sebek is a client/server solution in which a set of clients monitor the honeypot hosts and report the observations to a central server in charge of the logging task. Sebek clients intercepts the read() system call, and are thus able to observe all data accessed by a hacker or malware in an unencrypted way. Since Sebek runs *within* the honeypot operating system, it is detectable by a knowledgeable intruder and can even be *disabled*, allowing the intruder to act undisturbed on the honeypot [Tan 2004, URL 14, URL 27].

A more sophisticated system was proposed in [Zhuge 2007]. The Honeybow high interaction honeypot system combines together three different monitoring techniques to minimize the ability of an attacker to escape from detection. These techniques focus on a specific aspect linked with network exploits: the generic need for an attacker to push an executable file to the victim taking advantage of a successful exploit.

- **MwWatcher.** Similarly to Sebek, MwWatcher runs within the virtualized high-interaction honeypot and logs suspicious modifications to the filesystem.
- **MwFetcher.** MwFetcher monitors the virtualized host from outside, accessing the virtual drive of the honeypot and detecting changes to the filesystem.
- **MwHunter.** MwHunter analyzes the network stream taking advantage of Snort-inline, and detects the presence of (unencrypted) executable samples in the network stream.

The authors claim that the combination of these three techniques maximizes the probability of correctly detecting an infection. Whenever a detection is detected, the honeypot is automatically sanitized and restored to a clean snapshot.

While Honeybow focuses on the detection of the download of a malware sample, and thus allows the honeypot host to be compromised, Argos [Portokalidis 2006] follows a different approach. Argos is a high interaction honeypot based on the popular Qemu [Bellard 2005] system emulator. Argos is a modification of Qemu to implement a memory tainting technique. Memory tainting was proposed also by previous solutions [Crandall 2005, Costa 2005], but Argos is the first to implement memory tainting as a containment technique for high interaction honeypots. Argos tracks the data flow in the physical memory of the virtualized system, marking all the bytes *derived* from incoming network packets as tainted. Whenever a tainted byte is used in an *illegal* way (e.g. its value is copied to the virtualized host's Instruction Pointer and thus hijacks the instruction flow), Argos stops the execution of the virtualized host and inspects its memory to retrieve more information about the exploit strategy. This allows the implementation of a very reliable containment

technique by stopping the execution of the honeypot as soon as an attacker succeeds in exploiting it. It is important to understand that the memory tainting detection does not support paging and, more importantly, detects only exploitation through code injection techniques. Argos honeypots are thus vulnerable to other kinds of attack, for instance to password brute-force attacks (used for the propagation of some recent worms [URL 12]).

2.1.3.3 Reflection

An interesting containment technique is applied in Potemkin [Vrable 2005]. Potemkin is a honeyfarm of high interaction honeypots developed at UC San Diego. Potemkin achieves an extremely high level of scalability by using resources only on the occurrence of an attack. When an IP is hit by an attack, Potemkin generates a new instance of a high interaction honeypot and assigns it to that IP. The generation is extremely fast and memory efficient thanks to the usage of Copy On Write (COW) techniques. This potentially allows Potemkin to minimize the resource cost of a high interaction honeypot.

The containment policy used in Potemkin is implemented in a central gateway at the border of the honeypot farm. Outbound packets generated by a given honeypot instance are allowed to exit to the Internet only when directed towards the attacker that initiated the conversation. This allows malware propagation strategies based on *phone-home* techniques to work correctly. Any connection attempt towards other hosts is blocked by the Potemkin gateway and *reflected* towards internal entities. Common protocols (DNS) are handled through local proxies able to handle the client requests, while all the other connection attempts are redirected back towards the honeyfarm. A compromised honeypot scanning in the attempt to propagate the infection will thus compromise other honeypots within the honeyfarm. This technique is extremely valuable since it allows a secure containment technique and allows the study of the effectiveness of the malware propagation techniques.

Potemkin's reflection is a very interesting technique for the observation of classical self-propagating worms. With the growth in sophistication of the observed malware, the complete isolation of the infected machine from the rest of the Internet may prevent the observation of the malware behavior. For instance, bots receiving commands from a centralized Command & Control channel would not be able to receive commands from the bot herder. Also, malware that verifies its connectivity to the Internet by checking the reachability of common websites would be reflected to the farm and would easily detect the difference between the farm node and the real Internet host. This may potentially affect the ability of such technique to correctly observe the propagation of such malware samples.

2.2 Data collection infrastructures

Section 2.1 provides an overview on different techniques to extract meaningful information about attack threats. These techniques represent data collection tools that can be used according to different strategies to collect information on attack threats.

2.2.1 Global perspective

The spread of self-propagating worms had great resonance in the world of Internet Security [Song 2001]. Self-propagating worms randomly scan the Internet trying to aggressively compromise the highest number of peers in the least possible time. Worms such as Slammer [URL 38] were so effective and fast in their propagation that they significantly affected the availability of Internet infrastructure, congesting the network links. The main purpose of these threats was fast, complete infection of the Internet. In such a scenario, an increasing interest of the research community led to the development of techniques to monitor these globally expanding threats, to allow early detection and study their propagation.

2.2.1.1 Internet Telescopes

Internet Telescopes observe a single large portion of the unassigned IP space in order to observe *global* events. Moore et al. showed in [Moore 2002] how Internet Telescopes can be compared to an astronomical telescope. Increasing the number of IP addresses being monitored (their "resolution"), Internet Telescopes increase their visibility into fainter, smaller and further objects. Internet Telescopes have been used to detect global threats such as the Witty worm [Shannon 2004] or the evolution of DoS attacks [Moore 2006].

Due to the high amount of IPs involved in this kind of deployment, often Internet Telescopes configure themselves as purely passive sinks for Internet traffic. Examples of such deployments are the CAIDA Internet Telescope [URL 6] and Team Cymru Darknet Project [URL 40].

In order to increase the level of interaction of Internet Telescopes and cope with the scalability problems, different filtering solutions have been proposed in the literature. Pang et al. propose in [Pang 2004] the combination of stateless responders with filtering techniques to drop repeated activities. By allowing, for instance, connections from one attacking IP to only N telescope IPs, these filters significantly reduce the load on the telescope. The reduced load allows the authors to increase the level of interaction and take advantage of iSink [Yegneswaran 2004] and honeyd [Provos 2004] and correctly *distinguish* activities.

A more complex solution is GQ [Cui 2006a, Cui 2006b]. GQ takes advantage of dynamic filters and protocol learning techniques [Cui 2006c] to filter out uninteresting activities and relay the selected activities to farms of fully fledged high interaction honeypots. These techniques are closely related to this work, and share

many similarities but also major differences. We thus postpone their detailed description to Section 4.5.2, where the reader will be able to better understand the challenges underneath this task and the differences between the various approaches.

Finally, Potemkin [Vrable 2005], introduced in Section 2.1.3.3, potentially configures itself as a high-interaction Internet Telescope, being able to theoretically run 1500 high interaction honeypots on a single server. The experimental validation run in [Vrable 2005] focuses on the performance evaluation of the deployment and no experience reports on Potemkin are currently available. No information is thus currently available on the effectiveness of such technique in observing Internet threats or on the nature of the collected data.

An important limitation of the Internet Telescope schema consists in the possible bias derived from the identification of the monitored IP range. Staniford et al. showed in [Staniford 2004] how self-propagating malware could take advantage of *a priori* knowledge of the active IPs to avoid these "black holes" in the IP space and consequently avoid detection.

2.2.1.2 Log aggregation

We have introduced honeypots as the primary way to collect information on malicious activities. Other sources of information exist, such as Intrusion Detection System logs or firewall logs, and can be aggregated to build global pictures of the Internet attacks.

SANS Internet Storm Center (ISC) is a well-known source of information for Internet threats. The Internet Storm Center collects logs from any volunteering contributor and aggregates them in DShield [URL 11], a "Distributed Intrusion Detection System". A free client application allows the upload of logs in a variety of formats, which range from personal firewall solutions to intrusion detection systems such as Snort and commercial firewall appliances. This information is used to build publicly available statistics on the evolution of the *events* on the various ports and can be browsed back in time.

A similar service is offered by Symantec Deepsight [URL 39]. Symantec provides as a free download a client, Deepsight Extractor, able to parse and submit to a central database logs generated by a number of different network appliances and personal firewalls. Contributors are allowed to access a web interface collecting detailed statistics on the submitted logs.

One of the main drawbacks of this kind of log aggregation approaches is the heterogeneity of the different sources of information. There is a considerable semantic difference between the events recorded by different systems. As shown in [Dacier 2004b], the aggregation of logs generated by diverse sources lacks the semantics and the homogeneity necessary to perform correct analyses and inferences.

2.2.2 Observing local activities

After 2003, the proliferation of fast and highly visible self-propagating worms started to decrease and almost came to a stop in 2005. The profile of the attackers changed, and stealthier, profit-driven activities started to spread in the malicious activity scenario. For instance, in [Baecher 2005] it was shown how bot herders often instruct their botnets to scan within specific IP ranges, throttling down the scan frequency. Different research works [Dacier 2004a, Cooke 2004] showed how, consistently with this change in scenario, the Internet was affected by a proliferation of smaller activities highly localized.

In order to tackle this new reality different projects have proposed different data collection strategies allowing to spread the observation perspective over the IP space and characterize the locality of these phenomena.

2.2.2.1 The Leurré.com project

The Leurré.com project is a distributed honeypot deployment result of the work of Pouget et al. [Pouget 2006]. The main purpose of this deployment is the generation of a standard distributed platform to collect *unbiased* information on a specific class of network attacks, server-based scanning attacks. The project aims at deploying equal and thus *comparable* honeypot sensors in different locations of the IP space, and study the differences in the observations in order to ideally build a map of the *"Internet weather"*.

The sensors are deployed on a partnership basis: any entity willing to access all the collected data needs to become part of the project by installing a honeypot sensor. At the moment of writing, the project has deployed over 50 sensors, covering all the 5 continents. Each sensor, based on Honeyd, emulates the presence of 3 IPs and monitors the network interaction of the attacking sources to these IPs through the collection of the full packet traces in *pcap* format. The emulation is purely passive, no application level script is associated with the open ports. The honeypots thus limit their emulation to the establishment of TCP connections but never reply to client requests.

The lack of interactivity with the attackers is a limiting factor for the Leurré.com project. The project addresses this problem in two ways.

Firstly, the collected information is enriched through correlation with other information sources:

- **IP Geolocation.** The project takes advantage of Maxmind [URL 19] to retrieve information on the geographical location of the attackers and the organization responsible for the network range in the registrar database.

- **Passive OS fingerprinting.** Taking advantage of passive OS fingerprinting techniques [URL 48] it is possible to inspect the collected packet traces and infer generic information of the operating system of the attacking client.

- **Reverse DNS resolution.** Reverse DNS resolution information is stored for every attacking source observed by the deployment.

Secondly, the collected information is analyzed through data-mining algorithms [Pouget 2006] that, by looking at the network and temporal characteristics of the observed activities, tries to cluster together activities likely to be connected to the same exploit implementation.

All this information is stored and aggregated in a set of meta-tables organized in such a way to allow its easy usage in data-mining tasks. A detailed overview of the collected information can be found in [Leita 2008c].

2.2.2.2 The Internet Motion Sensor

The Internet Motion Sensor [Bailey 2005] is a distributed Internet Telescope. In order to obtain visibility on localized phenomena, the Internet Motion Sensor spreads its observation points on different network blocks. In [Bailey 2005] the authors describe the Internet Motion Sensor as being composed of 28 network blocks whose size can reach even the size of a /8 network. The Internet Motion Sensor can thus be seen as a hybrid solution, which tries to combine together the observation capabilities of Internet Telescopes with respect to global activities and achieve visibility on localized phenomena through the dispersion of the telescope address blocks along different IP blocks.

Due to the high number of IPs being monitored, the emulation solution used by the Internet Motion Sensor is rather simple: to any incoming SYN packet on any port, the sensors reply with a SYN/ACK packet. According to the authors, this behavior is sufficient to obtain a sufficient characterization of the attack activity. Due to the heavy load implicit in the storage of the network interaction associated with the sensors, the authors employ a caching technique based on the MD5 hash of the application payload. The same hash is used as a signature for a given type of activity, under the implicit assumption that a given network activity exhibits always the same application payload.

The lack of interaction with clients may not suffice though to identify subtle differences among different network activities: for instance, the LSASS exploit mentioned in Section 2.1.2.2 consists of several interactions between the client and the server, the first of which is a benign query that does not reveal yet the real intention of the attacker. The techniques introduced in [Bailey 2005] do not appear to be suitable to correctly handle these more complex activities, in which the attacker exposes its real intentions only after a set of preliminary interactions with the victim.

2.2.2.3 The Honeynet Project

The Honeynet Project [URL 43] has recently started an attempt to standardize their honeypot configurations under a common platform, the Global Distributed

Honeynet (GDH). GDH consists in a CD-ROM that automates the installation and configuration procedure of a honeypot sensor based on some of the technologies presented in Section 2.1, namely Honeywall, Nepenthes, and Sebek-monitored high interaction honeypots. The minimal GDH platform is normally associated with 4 IPs: one IP for the access to the physical host, one for the Honeywall virtual host, one for Nepenthes and one for a Fedora Core 3 Server high interaction honeypot.

Partners hosting GDH honeypots agree to allow the remote collection of the information collected by the honeypot, which is stored in a central server. This information comprises pcap data and Snort logs collected on the Honeywall, and the binary samples collected by Nepenthes. A special tool developed by the UK Honeynet Project, *honeysnap*, parses this information and store it in a database for further analysis.

As GDH is still under development, it is thus difficult to assess its capabilities: at the moment of writing, no result has been presented yet. The effort of the Honeynet Project in standardizing the Honeywall architecture and centralizing the data collection exemplifies though the need to take advantage of standard platforms to generate comparable results, need that was first reflected in the creation of the Leurré.com project seen in Section 2.2.2.1. Differently from the Leurré.com project, the GDH minimal configuration includes a high interaction honeypot contained through Honeywall. This may limit the deployment base for this project: for instance, it is unlikely that any industrial entity would accept to cope with the liability problems inherent in the deployment of such solutions (e.g. usage of the honeypot as a stepping stone to attack others), regardless of the warranties provided by the Honeywall containment.

2.2.2.4 Tunnel-based honeyfarms

All the previous infrastructure aim at physically deploying honeypots in different locations of the IP space. An alternative to this deployment strategy comes with the concept of *honeyfarm*[URL 35]. The main idea shared by all the different implementations is the usage of different types of tunnels to redirect traffic from different locations to a central farm of virtualized hosts acting as high interaction honeypots.

Several different implementations of this concept exist in the literature, such as Collapsar [Jiang 2004], NoAH project's Honey@Home [URL 15] or the Honeynet Project Honeymole [URL 41].

For instance, Collapsar proposes an architecture composed of three parts:

- Multiple traffic redirectors, deployed in different networks, are in charge of capturing all packets and filtering them according to rules set by the system administrator. All the packets are then redirected taking advantage of either the Generic Routing Encapsulation (GRE) tunneling mechanism of a router or using an end-system-based approach.

- A front-end for the Collapsar center, in charge of receiving the encapsulated

packets from the different traffic redirectors and dispatches the packets to a farm of high interaction honeypots. The front-end is also in charge of ensuring the containment of the outgoing activities using host and network monitoring techniques.

- The Collapsar center, a farm of high interaction honeypots running within virtual machines.

The simplification in terms of complexity of the distributed sensors participating to the honeyfarm leads to an easier deployability in terms of complexity of the distributed sensors. This has led projects such as Honey@Home [URL 15] to propose a similar architecture to host honeypot sensors, acting as simple traffic redirectors, on end users machines. The honeyfarm architecture does not solve though the containment issues associated to the usage of high interaction honeypots.

2.2.3 Malware-oriented data collections

We have seen in Section 2.1 how certain honeypot deployments take advantage of different techniques to collect malware samples. The research community has an increasing interest in accessing these malware samples, in order to study the characteristics of these samples, the obfuscation techniques being used, and to develop better ways to detect infections. A number of deployments choose to focus on the sole collection of these samples and exploit them to know more about the attackers.

The mwcollect Alliance [URL 25] takes advantage of Nepenthes honeypots hosted by different contributing partners to collect malware samples. The Nepenthes sensors take advantage of an ad-hoc protocol called *G.O.T.E.K.* to automatically submit any downloaded binary to the alliance database. The alliance provides a very rich dataset of malware samples, which are freely downloadable by any partner hosting a GOTEK-enabled Nepenthes sensor, but does not try to collect any rigorous information on the sources or the victims of the attacks.

A richer and more complete dataset is offered by SRI International through the Cyber-TA project [URL 36]. The dataset is built upon the logging capabilities of BotHunter [Gu 2007] and takes advantage of a set of high interaction honeypots dynamically mapped to the addresses of a /17 network taking advantage of NAT techniques. BotHunter allows the detection of the malware infections and the successive sanitization of the hosts, and provides information on the infection stages in a simple behavioral signature of the malware sample. Once a high interaction honeypot is detected as infected, a forensic analysis takes place before its sanitization. This analysis consists in the extraction of all the filesystem modifications performed by the malware for offline analysis. The containment policy in the deployment is rather loose: the honeypots are free to initiate connections to the outside world until the infection is detected and cleaned. The overall information provided by the deployment is very interesting, but the lack of strict containment limits the spread of the observation points to a single closely monitored network.

2.3 Conclusion

This Chapter has provided an overview on the current state of the art in data collection techniques, overviewing separately the tools that can be used to collect information and the strategies that are currently used to exploit these tools. In both cases, a tradeoff clearly emerges.

High interaction honeypots allow us to retrieve in-depth information on malicious threats, providing the maximum possible level of interactivity. These solutions pose containment problems, which can be addressed and *partially* solved taking advantage of different containment techniques. These containment techniques require the deployment of complex infrastructures, complexity to be added to the resource cost of the usage of real OS implementations. Low interaction honeypots require a significantly lower level of complexity, but fall into another pitfall. For a low interaction honeypot to be a useful data collection tool, it needs to emulate the behavior of a real host and its protocols. Only through application protocol emulation it is possible to lure the attacker into revealing his intent and thus his nature. Existing solutions lack of rigor in accomplishing this task: even the most sophisticated solutions, such as Nepenthes, assume an *a priori* knowledge of network attacks. This *a priori* knowledge implicitly biases any result, since it implicitly makes assumptions on the results *expected* in the collected data.

Existing data collection infrastructures amplify this trade-off. The observation of global threats allows the deployment of centralized solutions, handling wide IP ranges, and employing sophisticated techniques to handle incoming attacks. But when the interest moves to spreading the observation points along the Internet IP space, the cost of this sophistication is not acceptable. We have seen how deployments such as Leurré.com or GDH rely on the volunteer hosting of honeypot platforms. In order to spread as much as possible these observation points, the sensors must be simple, maintainable and must offer strong security guarantees to the hosting partner. The containment techniques available as of today are either too weak, or too complex to fit such requirements. The choice moves then to low interaction techniques, with the previously identified problems in offering a rich and unbiased perspective on the observed attacks.

This work proposes to increase the interaction capabilities of low interaction honeypots taking advantage of protocol learning techniques. We have seen how honeypots such as Honeyd offer a very limited number of responders, especially for binary protocols. We identify the problem in the complexity of the manual generation of scripts emulating the protocol behavior, especially for complex binary protocols such as NetBios, whose specification is not open and that is one of the most heavily targeted protocols for Internet attacks. This work proposes an algorithm to automatically generate such emulators in a protocol agnostic fashion. We want to avoid any *a priori* assumption on the structure of network attacks, with the purpose of obtaining an unbiased and behavior-based perspective on the observed phenomena. The idea of taking advantage of protocol learning techniques is inspired by the Protocol Informatics Project [Beddoe 2005], in which Beddoe pro-

posed the usage of bioinformatics algorithms to help protocol reverse-engineering tasks. The problem of automated protocol reverse engineering was addressed in different ways by different research works. For the sake of clarity, we have decided to postpone their comparison with our work to Section 4.5.

Chapter 3

ScriptGen

Contents

We have seen in the previous Chapter that low interaction honeypots deal with two main issues. On the one hand, building protocol emulators to emulate the presence of a host is a tedious and costly procedure. On the other hand, existing approaches attempting to achieve this goal are biased by a set of *a priori* assumptions.

If we consider the finite state automata ϕ representing the language of protocols such as NetBios, its complexity may make its modeling an extremely tedious

task. We consider here the portion of finite state automata ϕ_{act} effectively traversed during the interaction with network malicious activities. Our work poses its foundations on the assumption $|\phi_{act}| \ll |\phi|$.

This assumption is based on the common knowledge of a strong collaboration in terms of code sharing and exploitation techniques among malware writers. Repositories such as milw0rm [URL 23] and the presence of a number of underground channels among hacker communities (such as the famous 29A Labs [URL 2], which recently ceased its activity) suggest the existence of a strong collaboration among these communities. In spite of the proliferation of different malware variants, we can intuitively expect a controlled number of exploitation vectors that are developed by a few experts and then shared by a bigger number of malware variants developed by less skilled hackers.

Also, the interaction of a vulnerable service with an exploitation script is not supposed to correspond to a complex traversal of the protocol finite state automata. If the interaction is targeting a vulnerability exploitable through code injection techniques, the objective of the attacker will be to reach a given interaction state affected by the vulnerability (for instance, an unchecked buffer). The corresponding exploit interaction is likely to correspond to the shortest path from the root of the automata to the state affected by the vulnerability.

Knowledge-based approaches such as Nepenthes implicitly take advantage of this scenario. Nepenthes, as shown in Section 2.1.2.2, takes advantage of 21 vulnerability modules to download malware. That is, taking advantage of the implementation of only 21 different traversals of the protocol FSMs, Nepenthes is able to collect a variety of different malware variants. The observation capabilities of Nepenthes are constrained by the a-priory choice of these 21 traversals. This work proposes a different approach to the problem aiming at avoiding this *a priori* choice.

The ScriptGen algorithm aims at rebuilding portions of a protocol finite state machine through the observation of samples of network interaction between a client and a server implementing such protocol. In order to maximize the flexibility in handling protocols whose specification is not open, the main constraint of the algorithm relies in its protocol agnostic nature: no assumption is made on the semantic structure of the protocol. The core of the ScriptGen algorithm aims at addressing this constraint and at rebuilding from the interaction samples an approximated knowledge of the protocol semantics.

The task of building a protocol finite state machine representing the protocol interaction can be subdivided into two subtasks. Firstly, we need to extract from the observed samples information on the structure of the protocol messages. Secondly, we need to structure the succession of messages along each TCP session or UDP request-answer tuple. These two subtasks are described respectively in Section 3.1 and Section 3.3. In order to evaluate the characteristics of the resulting algorithms, each subtask description is coupled with experiments assessing the practical impact of the proposed approaches, corresponding to Sections 3.2 and 3.5. The reader should not consider these experiments as an experimental

1	MAIL FROM: <alice@eurecom.fr>
2	MAIL FROM: <bob@orange.fr>
3	MAIL FROM: <carl@free.fr>

Table 3.1: Input example

validation of the approach. A thorough validation will be carried out in Chapter 4, where ScriptGen-based emulators are deployed in a distributed data collection framework, SGNET.

3.1 Region Analysis

The core of the ScriptGen approach consists in the region analysis algorithm. Region analysis is responsible for the reconstruction of a partial semantic structure starting from a set of samples. The input to the region analysis algorithm consists in a set of messages considered semantically *similar*. An input example is provided in Table 3.1 for the case of the *MAIL FROM* command in the SMTP protocol. It is important to understand that this input is free from any semantic knowledge and is seen by the algorithm as a set of unstructured byte streams. From now on, we will use this input example to practically demonstrate the region analysis process.

In order to rebuild semantics from these unstructured sequences of bytes, region analysis takes advantage of bioinformatics algorithms to produce alignments of the samples. The idea builds upon the Protocol Informatics Project by Marshall Beddoe [Beddoe 2005]. Beddoe proposed to exploit global alignment techniques to ease protocol reverse engineering.

Alignment techniques are used in bioinformatics to find overlaps in two distinct sequences of amino acids, in order to identify specific genes. Two different alignment techniques can be identified: *local* and *global* alignment [Gusfield 1997]. Local alignment algorithms aim at identifying the most similar subsequence among two sequences, and thus aim at identifying the similarity among two different evolutionary paths. Global alignment algorithms are used instead to identify alignments from the beginning to the end of a sequence, and are used when two sequences are considered similar.

Beddoe showed in [Beddoe 2005] how the bioinformatics algorithms previously introduced could be of great help to ease the reverse engineering tasks. We propose here to take advantage of bioinformatics algorithms and extend them in order to *automatically* infer from a set of samples a partial notion of its syntax. The region analysis algorithm aims at producing from sample conversations groups of semantic abstractions generalizing them. Each semantic abstraction is composed of one or more *regions*, where each region represents a semantically uniform portion of the protocol structure.

The region analysis algorithm is composed of four building blocks as shown in Figure 3.1. The algorithm receives as input a set of client requests likely to be

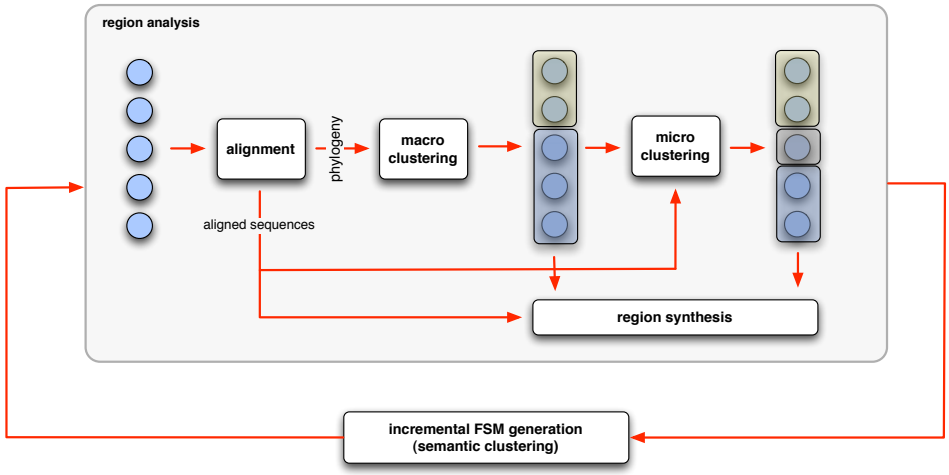

Figure 3.1: The region analysis algorithm

semantically very similar. The algorithm ensuring this "semantic clustering" will be discussed in detail in Section 3.3.

- **Sequence alignment.** We use the Unweighted Pair Group Method with Arithmetic mean (UPGMA) algorithm [Tateno 1982] to build a dendrogram (*phylogenetic tree*) representing the degree of similarity among the different message samples and to construct their corresponding alignment.
- **Macro-clustering.** We analyze the previously generated phylogenetic tree to identify big variations in the structure of the input messages and cluster the samples accordingly.
- **Region synthesis.** For each cluster, we identify in the aligned sequences ranges of subsequent bytes sharing similar characteristics and we assign them to different protocol *regions*. A regular expression is generated representing the different semantic value associated with each region type.
- **Micro-clustering.** We analyze the content associated with mutating fields and search for recurring values to refine the initial clustering decision. We assume in fact that if a mutating field has low variability in its content, it is likely to be associated with a semantic value worth being represented by a separate cluster.

These four functional blocks are detailed in the following Sections.

3.1.1 Sequence alignment

As previously explained, the input samples are seen by ScriptGen as a sequence of bytes. No information is available on the protocol structure or on the different

tokens composing each message. We thus want to use alignment algorithms to identify the fixed portions of the protocol, and build a *skeleton* of the message structure. Alignment algorithms allow us to be robust to misalignments in protocols using variable length fields. For instance, the output of the alignment algorithms introduced in this Section on the sample in Table 3.1 is the following (the character '_' is used to represent a *gap*).

```
MAIL FROM: <alice@eur___ecom.fr>
MAIL FROM: <__bob@_orange___.fr>
MAIL FROM: <_carl@_fr__ee___.fr>
```

In order to infer the shared structure of two *similar* messages we take advantage of global alignment algorithms. The most common algorithm implementing global alignment is the Needleman-Wunsch [Needleman 1970] algorithm. It is a dynamic programming algorithm aiming at achieving the best alignment of two sequences S_1 and S_2 according to a given *scoring function*. For every couple of bytes belonging to the two sequences, a different score is associated with identical values (I_{score}), differing values (D_{score}), or insertion of a gap in one of the two (G_{score}).

As recognized by other works taking advantage of the same algorithm for protocol analysis [Cui 2006c], these scores have a considerable impact on the quality of the alignment when considering a single couple of messages. When dealing with the alignment of more than two messages, the practical experience of the author in [Beddoe 2005] showed that the best results can be obtained by using $I_{score} = 1, D_{score} = 0, G_{score} = 0$. This choice can be considered "greedy": no penalization is given to the insertion of gaps in the sequences or to the presence of mismatching bytes, and the algorithm simply aims at obtaining the maximum possible overlap among the two sequences. This is likely to produce accidental alignments among two messages. Graphically representing gaps with the symbol '_', follows an example of such phenomenon:

```
MAIL FROM: <alice@eur___ecom.fr>
||||||||||||      |  |   |   ||||
MAIL FROM: <__bob@_orange___.fr>
```

By choosing to use *multiple* messages, we enable the algorithm to filter out accidental alignments among a limited number of messages and to isolate the real structural similarities at the cost of an additional computational complexity. Related work carried on in [Cui 2006c] performs different design choices by pursuing a different goal, that is the efficient detection and removal of previously seen activities. By using only two samples of interaction, the method requires a more careful choice in the selection of the alignment parameters.

We present here the validity of the greedy approach in the case of the multiple alignment as a simple intuition. Section 3.2.2 will experimentally investigate the relationship between the number of aligned messages and the correctness of the semantic inference.

The process involved in the alignment of more than two messages is called in bioinformatics *multiple alignment*. The exact solution of the multiple alignment problem is an expensive operation (NP-Complete). For this reason, it is normally performed taking advantage of heuristics able to provide good results at a lower complexity. A common method used in bioinformatics consists in applying the Needleman-Wunsch algorithm multiple times on two samples per time, following a hierarchical structure called *phylogenetic tree*. The phylogenetic tree is a binary tree that groups together samples or group of samples according to their mutual similarity.

We build the phylogenetic tree taking advantage of the Unweighted Pair Group Method with Arithmetic mean (UPGMA) algorithm [Tateno 1982]. The UPGMA clustering is a bottom-up clustering method.

Given two clusters C_i and C_j UPGMA uses a notion of distance d_{ij} defined as:

$$d_{ij} = \frac{1}{|C_i||C_j|} \sum_{p \in C_i, q \in C_j} D_{pq} \tag{3.1}$$

where D_{pq} is the distance between two samples. Figure 3.2 shows a very simple bidimensional case, in which D_{pq} is defined as the Euclidean distance between the two points on the plane.

The algorithm is initialized with the creation of a set of n clusters $C = \{C_1, C_2, ...C_n\}$ containing each of the n elements taken into consideration. The phylogenetic tree is initially composed by n nodes $\{N_1, N_2, ...N_n\}$, associated with each cluster, and constituting the leaves of the tree. The algorithm iterates as follows:

1. Compute the distance d_{ij} among each couple of clusters (C_i, C_j) with $C_i, C_j \in C$. Choose the two clusters (C_u, C_v) that minimize such distance.

2. Merge the two clusters (C_u, C_v) into a single cluster C_{uv} resulting from their union.

3. Generate a node N_{uv} in the phylogenetic tree having as children N_u and N_v. The node is characterized by a *height* defined as $h(N_{uv}) = d_{uv}$

4. If the number of remaining clusters $|C'|$ after the merge is greater than one, reiterate to step 1.

The resulting phylogenetic tree is a binary tree. The height value associated with each node is monotonic with the tree level. That is, the height of the parent node will always be greater than or equal to the height of its children. This property, which will be used by the macro-clustering phase to refine the clustering decision, which can be intuitively understood by looking at the phylogenetic tree resulting from the example of Figure 3.2 and reasoning by contradiction, as follows.

We denote with A, B, C, D the four elements being clustered, with C_{AB} a cluster grouping the elements A and B and with N_{AB} the corresponding node in the phylogenetic tree. We want to show that $h(N_i) \leq h(N_j)$ if N_i is a child of N_j. Let's

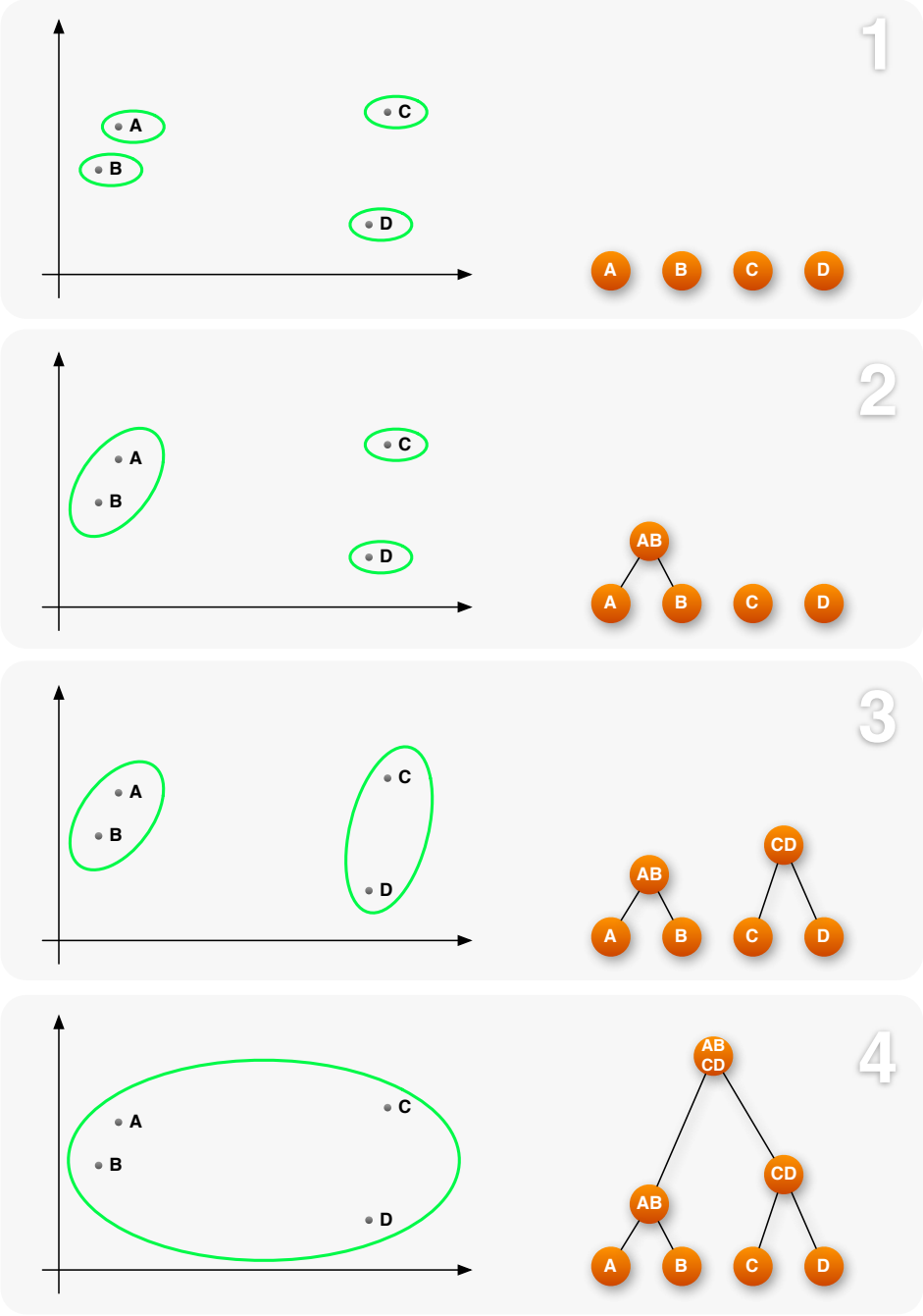

Figure 3.2: UPGMA clustering process

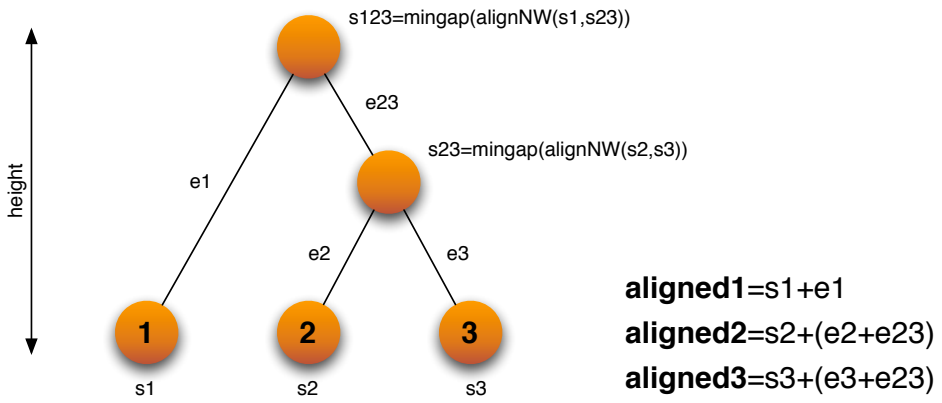

Figure 3.3: Multiple alignment based on phylogenetic tree

assume that, for instance, $h(N_{AB}) > h(N_{ABCD})$. According to the definition of height, this would be equivalent to state that $d_{A,B} > d_{AB,CD}$. That is, the average distance between the elements of C_A and C_B would need to be greater than the average distance between the elements of C_{AB} and C_{CD}. For this to be true, there must be at least a couple of elements belonging respectively to C_{AB} and C_{CD} having smaller mutual distance than the two elements (A, B) belonging to C_A and C_B. At every iteration, the UPGMA algorithm groups together the two clusters having the smallest average distance among their elements. The existance of the cluster C_{AB} implies that the distance between A and B is smaller than the distance among A and any node other than B. The same can be said for node B and any node other than A. It is thus impossible to find a node in C_{CD} having a smaller distance from them.

To apply UPGMA to our conversation samples, we have considered different definitions of distance D_{pq}, such as simple bitwise similarity, or Normalized Compression Distance (NCD). A definition of distance that proved to be particularly suitable to the problem in our early tests can be borrowed again from the bioinformatics scenario, and consists in the usage of local alignment algorithms. We saw for global alignment that the Needleman-Wunsch algorithm is a dynamic algorithm aiming at maximizing a score. The insertion of gaps, the misalignment, and the correct alignment of each byte lead to positive or negative scores defined as parameters to the algorithm. The Smith Waterman [Smith 1981] is a local alignment algorithm; it modifies the Needleman-Wunsch algorithm to take into consideration only segments of the sequence rather than the whole. We take advantage of the output of the scoring function using as parameters $I_{score} = 1, D_{score} = 0, G_{score} = 0$ as a measure of similarity among two sequences.

The generated phylogenetic tree is then used as a skeleton to apply the global alignment. All the leaves of the tree are initialized with the values of the input samples. Taking advantage of a depth-first visit and thus starting from the bottom of the tree, we take advantage of the Needleman-Wunsch algorithm to compute a

value for all the intermediate nodes. Each node is assigned a value generated by the alignment of its children. Among the two aligned sequences resulting from this process, the one with the least number of gaps is chosen for the father.

The result of the alignment can be represented in a set of *edits* to be applied to the original sequence to build the aligned one. No deletion is generated by the alignment process, and the edits solely consist in the list of the positions in which a gap must be added.

For instance, in order to build the aligned sequence `_M__AI_L F_ROM` starting from `MAIL FROM` the following edits must be applied: $edits = \{0, 1, 1, 3, 6\}$. The required edits to align the children of each node are stored in the edges of the phylogenetic tree as shown in Figure 3.3. Once the traversal is completed, the final alignment of each leaf can be reconstructed by combining all the edits stored in each edge of the path between the tree root and the leaf.

Summarizing, the sequence alignment phase takes advantage of existing bioinformatics algorithms to align multiple message samples, inserting *gaps* to maximize overlaps. The multiple alignment is achieved in two steps:

1. **Construction of the phylogenetic tree.** Taking advantage of UPGM clustering, we build a binary tree grouping together the different samples according to their mutual similarity.

2. **Alignment.** The phylogenetic tree previously constructed is used as a skeleton to recursively align all the samples taking advantage of the Needleman-Wunsch algorithm. The edits required to perform each alignment step are stored in the edges of the phylogenetic tree and are used to generate the mutual alignment of the leaves.

3.1.2 Macro-clustering

In order to produce correct results in the semantic abstraction, it is important to group together sequences having similar structure and semantic meaning. Intuitively, the alignment of two completely different messages can lead only to accidental overlaps, and is likely to produce very imprecise results. It is thus important to ensure that the input to the region analysis is composed of semantically similar messages.

We call the process of grouping together messages likely to share the same semantic meaning *semantic clustering*. Two possible approaches can be used to implement this concept:

- Analysis of the *content* of the messages. If two messages are *very different* from each other, they are likely to be associated with different semantics. For instance, the structure of an HTTP GET message is likely to be significantly different from the structure of an HTTP POST message.

- Analysis of the *context* of the messages. Much more can be inferred by looking at the context in which a given message appears. The context comprises both

the position of the message, but also the content of the messages semantically linked to it. For instance, semantically similar client requests are likely to receive very similar server answers.

Both these approaches are implemented in ScriptGen. Macro-clustering is the region analysis component responsible for the analysis of the messages content. The analysis of the context will be addressed instead in Section 3.3, where the refinement algorithm will have scope on the whole conversation.

Macro-clustering aims at the identification of groups of messages that are highly dissimilar and whose alignment should thus be handled separately. This is easily achievable by taking advantage of the characteristics of the phylogenetic tree built during the alignment phase. The phylogenetic tree is defined, in bioinformatics terms, as an evolutionary tree that represents mutations as time goes on. From a clustering perspective, the phylogenetic tree is nothing else than a dendrogram result of a hierarchical clustering technique based on a given notion of distance. Each node of the phylogenetic tree is associated with a *height* that we have shown to be monotonic when moving up in the tree levels. The height of a node represents the level of diversity among the samples belonging to the subtree having that node as root.

The *macro-clustering threshold* W defines the maximum height (i.e. the maximum diversity) of the phylogenetic tree. If the height of the tree root k is greater than W, the tree is cut into two subtrees having as roots the children of k. The condition is recursively verified on the roots of the subtrees and the division continues until the condition is satisfied. Thus macroclustering splits the original phylogenetic tree in a set of subtrees whose root k_i satisfies the condition $height(k_i) < W$. Once the phylogenetic tree is split into these clusters, the multiple alignment does not need to be repeated. The alignment of all the leaves of each subtree can be easily achieved by taking advantage of the edits stored in the edges connecting the new root to each leaf.

The value of the macro-clustering threshold W has thus an important impact on the grouping of the input samples. The impact of such threshold on the quality of the final semantic inferences will be evaluated in Section 3.2.1.

3.1.3 Region synthesis

The region synthesis step aims at analyzing the result of the multiple sequence alignment and inferring a semantic structure for the protocol messages. This structure is built upon the concept of *region*. A region is defined as a set of subsequent bytes in the protocol stream sharing similar characteristics, and likely to correspond to a given protocol field or token.

In order to identify the different protocol regions, a set of characteristics is generated for each byte of the aligned sequences, as shown in Figure 3.4. Taking into consideration a cluster composed of n aligned sequences, we define two characteristics that proved to be the most relevant: byte type and byte variability.

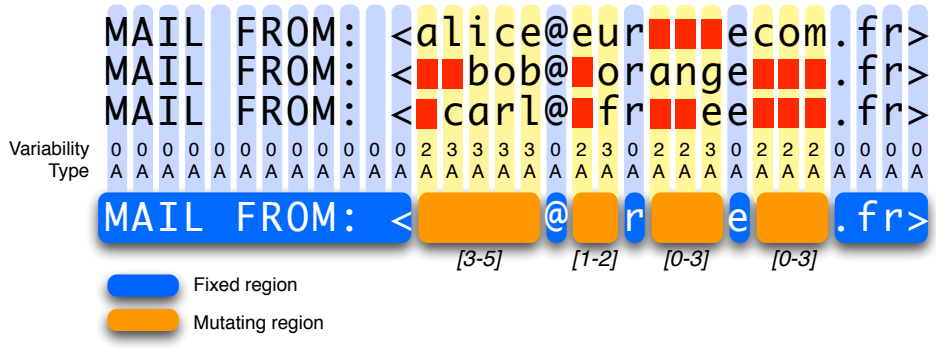

Figure 3.4: Example of region synthesis

Byte type. The type of the byte is defined as the *most frequent* type of content among all the aligned samples, *excluding gaps*. The content can be associated with two different types: ***ASCII***, ***Binary***. The byte belongs to either of the two types depending on whether its value falls within the range of values belonging to the standard ASCII code (values ranging from 0x20 to 0x7E). Such partition of the range of values can lead to ambiguities: a binary byte might be erroneously classified as ASCII. We aim at taking advantage of the statistical variability of the samples to filter out these ambiguities and correctly identifies the byte types.

Byte variability. The byte variability is a measure of the variability of the byte content among the different samples. Considering the set Ω of all the different values assumed by the n aligned samples for a given byte (including gaps), the variability is defined as:

$$v = \frac{|\Omega|}{n} \tag{3.2}$$

For the ease of representation, we show in Figure 3.4 the quantity $vs = v * n = |\Omega|$. The previous variability metric allows the definition of three variability classes:

- **Fixed content.** When $v \simeq 0$.
- **Random content.** When $v \simeq 1$.
- **Mutating content.** When $0 < v < 1$.

We take advantage of these characteristics to define a region as a sequence of bytes which i) contain the same kind of data and ii) have same variability class. A region can be seen as a portion of protocol message which has some homogeneous characteristics and therefore carries the same kind of semantic information (e.g. a variable, an atomic command, white spaces, etc...). Two region types are defined according to the variability class of their bytes:

- **Fixed regions.** Regions characterized by fixed content. Fixed regions represent the structural blocks of the protocol, and are assumed to carry a very strong semantic value.

- **Mutating regions.** Regions characterized by variable content (random or mutating content). In a first approximation, if the content of a region is associated with different values in different samples, it carries a weaker semantic meaning. This approximation will be refined in the micro-clustering phase.

Every region is also associated with a length attribute. While the length of a fixed region corresponds to the length of its content, the length of a mutating region is defined by a range. Such range corresponds to the minimum and maximum length of the portion of samples enclosed in the region, stripped out of gaps (see Figure 3.4). Representing a fixed region as *F("content")* and a mutating region as *M(length)*, we can represent the output of the region synthesis for Figure 3.4 as:

F("MAIL FROM: <")+M(3:5)+F("@")+
M(1:2)+F("r")+M(0:3)+F("e")+M(0:3)+F(".fr>")

This structure can be easily used to produce semantic abstractions of the observed protocol under the form of regular expressions. For instance, the example in Figure 3.4 can be translated to the following grep-like regular expression:

```
(MAIL FROM: <)([[:alnum:]]{3,5})(@)
([[:alnum:]]{1,2})(r)([[:alnum:]]{0,3})(e)([[:alnum:]]{0,3})(.fr>)
```

The example of Figure 3.4 clearly shows the pitfalls inherent in the choice of the initial training set. In the example, a number of erroneous inferences is performed on the semantics of the samples. The generalized representation constrains the username to be between 3 and 5 characters long. Also, it imposes the presence of the characters "r" and "e" at different positions of the domain name. More specifically, an "r" is expected at the second or third character of the domain name, and an "e" is expected between the third and sixth character of the domain name. These wrong inferences are due to the lack of sufficient statistical variability in the training set, leading to deductions that are true for the specific training set being used but wrong for the general semantics of this specific protocol. The collection of a good and *diverse* training set is thus of primary importance for the quality of the semantic inference performed by the region synthesis.

3.1.4 Micro-clustering

The micro-clustering step is a refinement of the macro-clustering output and of the associated region synthesis phase. We saw in the previous Section that fixed and mutating regions are associated with different semantic meanings: fixed regions represent the semantic structure of protocol messages, while mutating regions are associated with mutating fields that carry weak semantic value. While fixed regions are associated with bytes having fixed content, mutating regions are associated with bytes having either random or mutating content. Micro-clustering aims at the differentiation of the semantic value associated with mutating and random content.

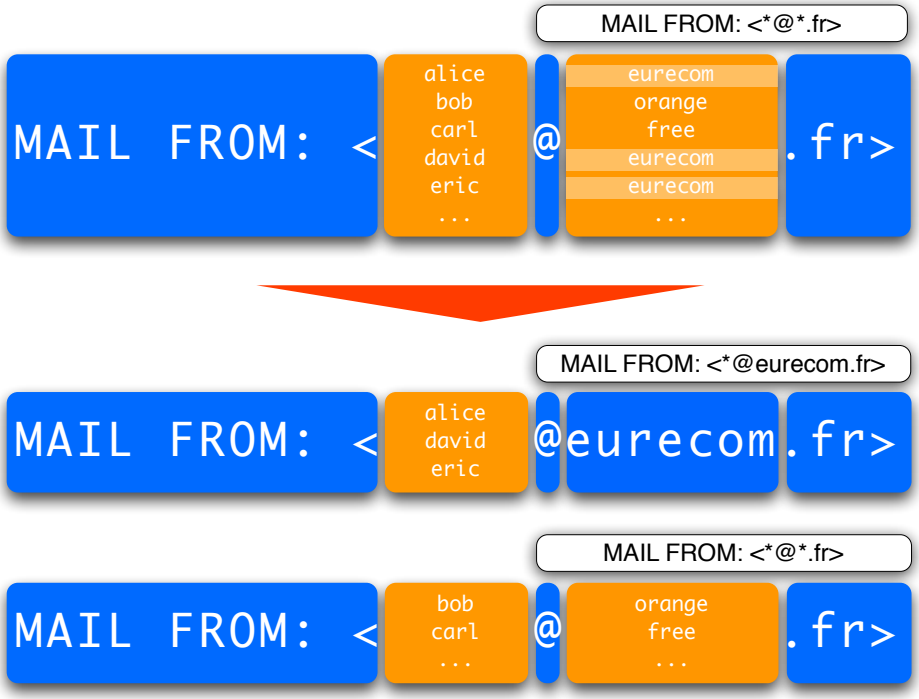

Figure 3.5: Micro-clustering

Random content is likely to be associated with timestamps and cookie fields. The value of these fields does not contribute to the semantics of the message. On the contrary, mutating content is likely to be associated with fields that normally change among a limited number of frequent values, such as bitmasks or flags. If many samples of the training set share the same value within a mutating region, they are likely to have a very specific semantic value, different from the others. As shown in Figure 3.5, micro-clustering creates a separate cluster for this specific value, and thus the corresponding mutating region is converted to a fixed one.

Algorithm 1 microcluster($region_{mutating}$)

1: $hist \Leftarrow$ compute_histogram($region_{mutating}$)
2: **for** *value* in *hist* having $freq/n > w$ **do**
3: splitcluster(*value*)

Micro-clustering operation is schematized in Algorithm 1. For every mutating region identified by the preceding region synthesis step, the histogram of the distribution of its values is computed together with its frequency. All the values having a frequency (normalized to the number of samples) higher than the *micro-clustering threshold* w lead to the generation of a micro-cluster associated with that specific value.

For a given macrocluster, each mutating region can produce a different refine-

ment of the sample grouping according to the identified frequent values. The final result of the micro-clustering refinement is given by the mutual intersection of all the refinements produced by the analysis of each mutating region. This can be better clarified through an example. Let's assume that micro-clustering generated the following clusters from the analysis of the content of two different mutating regions over a set of 9 samples:

$$C_1 = \{\{1,2,3,4\},\{5,6\},\{7,8,9\}\}\ C_2 = \{\{1,2\},\{3,4,5,6,7,8,9\}\}$$

where each number refers to the ID of the sample within the sample set. While the first four samples are associated with the same group by the application of Algorithm 1 on the first region, the analysis of the second region underlines a difference between samples $\{1,2\}$ and samples $\{3,4\}$. The final outcome of the micro-clustering is thus generated by the superposition of these groups:

$$C_{result} = \{\{1,2\},\{3,4\},\{5,6\},\{7,8,9\}\}$$

For each generated group, we re-run the region synthesis in order to reconsider the nature of each region. As a followup to the grouping performed by microclustering, the type of some mutating regions will be reconsidered, as shown in the example of Figure 3.5 for the second mutating region of the first group.

3.2 Threshold sensitivity

We have seen in the previous discussion that the Region Analysis algorithm allows the semantic abstraction of protocol messages taking advantage of a set of samples of protocol interaction. Before showing how this technique can be employed to build Finite State Machines representing the protocol interaction (Section 3.3), we want to devote some space to the analysis of the impact of the region analysis parameters on the quality of the semantic inference.

While the region analysis process is meant to be completely automated and free of any intervention from the human operator, we introduced in the previous Section a set of thresholds and parameters. These thresholds control the impact of the different building blocks on the final result, and thus impact its correctness.

- **Macroclustering threshold (W).** The macroclustering algorithm takes advantage of the dendrogram built during the multiple alignment to group together similar messages believed to belong to the same semantic group. This process is regulated by a threshold W that represents the variability accepted within a given macrocluster. If $W \to 0$, the region synthesis will be performed on many small and specific clusters. If $W \to 1$, the region synthesis is performed on a few big clusters containing a higher amount of variability.

- **Microclustering threshold (w).** Microclustering detects within mutating regions the presence of frequent values that are then used to refine the initial macroclustering decision. The microclustering threshold w defines the frequency for which a given value leads to such a refinement. Defining as n the number of sequences and k the number of samples in which a given value appears, the value is considered frequent if and only if $k/n \geq w$.

- **Number of samples (N).** It represents the number of samples to be used as input to the refinement phase of the Region Analysis algorithm. A tradeoff exists between the complexity of the multiple alignment algorithm and the need to obtain sufficient statistical variability to correctly infer the semantic structure of the protocols. In order to assess the practical usability of the algorithm, we need to explore this trade off.

This Section does not aim at being a benchmark for the quality of the semantic inference produced by the region analysis. We want instead to evaluate its sensitivity to these parameters and evaluate their impact on the interaction among different building blocks. We thus decided to perform our experiments on a simple ad-hoc protocol combining representatives of the three variability types: fixed, random and mutating. The definition of such a protocol allows the execution of the experiment in controlled but still realistic conditions, easily identifying incorrect inferences with respect to the protocol definition.

We take into consideration a simple ASCII protocol, in which each client request is composed of a command and a client identifier in the form of a 16 byte ID. Three different client commands are defined and considered equiprobable: "GET", "STORE", "QUERY".

STORE 'uhbltkahtkgayzpv'
GET 'otezhbbnsrafedew'
QUERY 'ezbesddoptblxpyq'

We performed our experiment on a sample of 100 client requests. Each client request was generated by randomly choosing the type of command and by randomly generating an ID for the user.

3.2.1 Macroclustering and microclustering

To evaluate the effect of the macroclustering and microclustering thresholds on the quality of the semantic inference, we evaluated the result of the Region Analysis for different values of the W and w parameters. We performed the alignment on all the 100 client requests in the sample, and used all the possible combinations of values in the interval $[0 : 1]$ with increments of 0.05. The result is represented graphically in Figure 3.6. Different colors in the area correspond to different number of semantic abstractions.

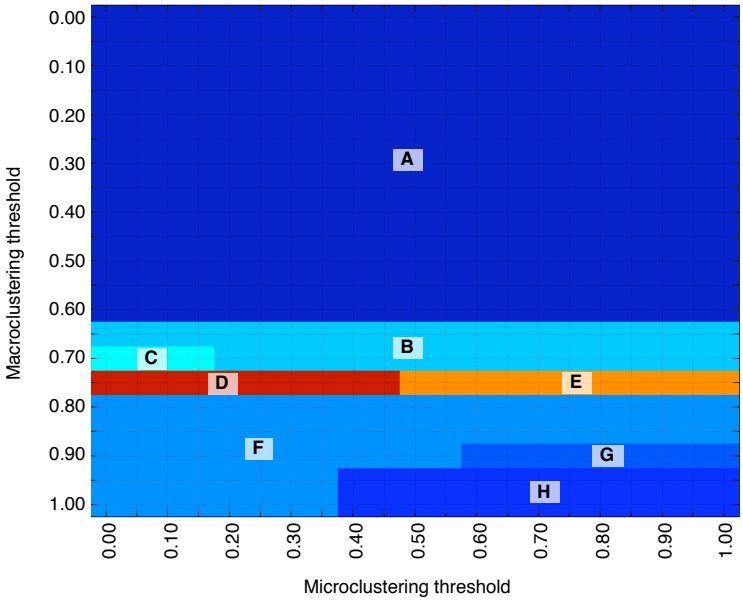

Figure 3.6: Effects of macroclustering and microclustering thresholds

It is possible to identify in Figure 3.6 eight different ranges of values that lead to different semantic inferences, corresponding to eight different areas on the map.

Zone A (0 abstractions). Any combination of values having $W < 0.6$ leads to the same result. The high variability of the samples (due to the presence of a 16 bytes long random ID) prevents macroclustering from forming groups with the required level of similarity, and thus prevents any refinement.

Zone B (4 abstractions), zone C (5 abstractions). For values of W in the interval $]0.6; 0.7]$ macroclustering identifies 4 small groups of samples (each group composed by less than 5 samples) whose user IDs contain similar characters in similar positions. This leads to wrong semantic inferences, such as the following:

$$F(''STORE'x'') + M(1) + F(''j'') + M(7:9) + F(''a'') + M(3:5) + F(''')$$

This semantic inference matches only 2 samples of the whole sample file. The generated inferences are too specific and match only 10 of the 100 selected samples. Zone C corresponds to a microclustering refinement on the clustering decision of Zone B. Such refinement is driven by the repetition of a portion of the user ID in the grouped samples, and is again caused by a lack of variability in the groups generated by macroclustering.

Zone D (15 abstractions), zone E (12 abstractions). The relaxation of the similarity constraint in the macroclustering algorithm allows the increase in the number of groups. The presence of accidental matches in the user IDs leads to an explosion of very specific abstractions, each matching just a few samples. Differently from what we saw for zones B and C, these semantic abstractions are

able to match all the 100 samples. In Zone D, microclustering further contributes to the explosion of the number of abstractions through the production of 3 additional refinements of the initial clustering choice, based again on the accidental match of bytes of the user ID during the alignment phase.

Zone F (3 abstractions), Zone G (2 abstractions), Zone H (1 abstraction). Only with high values of W macroclustering creates groups with enough variability to generate meaningful abstractions. In absence of microclustering ($w \to 1$) high values of W bring to degenerate solutions such as that of zone H, characterized by a single abstraction of the form:

$$M(3:4) + F(“’”) + M(16) + F(“’”)$$

For low values of the microclustering threshold w, the algorithm is able to recover the semantic of the first mutating region, leading to the correct refinement (corresponding to zone F).

It is clear from this experiment that the macroclustering threshold has an enormous influence on the quality of the semantic abstraction. While high values lead to oversimplifications of the protocol semantics, low values lead to too specific and unusable abstractions. While the second condition is not recoverable, the first is recovered by microclustering, which reconsiders the initial clustering choice through an analysis of the mutating region content. For values of $W \geq 0.8$ the combination of macroclustering and microclustering is always able to produce a set of semantic abstractions consistent with the protocol specification (zone F).

We can conclude from this investigation that the ability of macroclustering to group together semantically similar samples is limited. This is due to the fact that the lack of semantic awareness in the notion of distance prevents us from distinguishing variations attributed to semantic differences from variations attributed to the semantics of a given protocol field (such as the user ID in the example). Referring to the previous example, the variability generated by the random cookie fields is indistinguishable from the variability generated by a different type of command. We consider macroclustering reliable only in pinpointing complete structural differences in the messages by using a very high clustering threshold ($W = 0.95$).

As mentioned when introducing the concept of semantic clustering, two possible techniques can be used to group together semantically similar messages. We saw here that the clustering based on the *content* of the messages is unable to fully achieve this goal. Our practical experience showed that the *context* of the messages within the session provides much more reliable information for their classification into semantically similar clusters. This concept will be addressed in depth in Section 3.3.

3.2.2 Number of samples

The quality of the semantic abstractions produced by the Region Analysis algorithm depends on the number of samples on which the alignment is carried on. Increasing

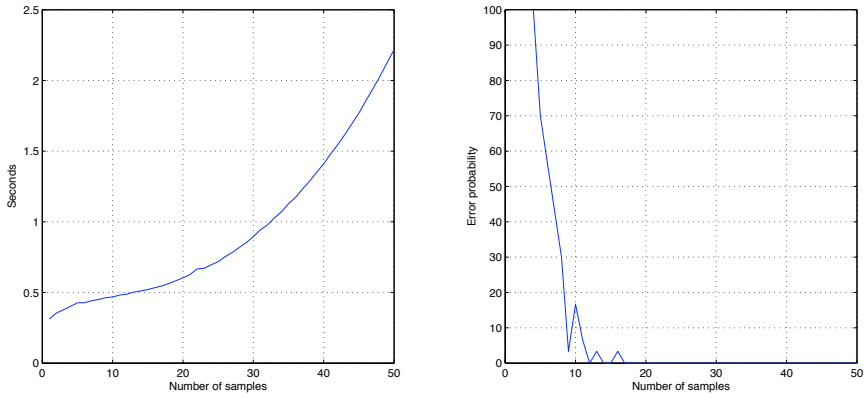

Figure 3.7: Effect of the variation of number of samples

the number of samples increases the statistical variability that can be exploited by the algorithm to correctly infer semantics. Each sample has a cost: firstly, obtaining a sample of protocol interaction may have a cost in terms of required resources or time; secondly, the cost of the multiple alignment depends on the number of samples on which the alignment is performed. For the Region Analysis algorithm to be usable in practice, the number of samples required to produce a correct semantic abstraction must be boundable, and must correspond to a reasonable resource requirement.

In order to investigate the issue, we have repeated the previous experiment fixing the macroclustering and microclustering thresholds ($W = 0.9$ and $w = 0.2$) and randomly choosing subsets of the original sample set with size N varying in the interval $[1 : 50]$. We repeated the selection of the samples and the corresponding alignments 30 times for each value of N.

The results of the experiment are represented in Figure 3.7. On the left graph we represent the average time required by the execution of the whole Region Analysis algorithm on our test system[1]. The right graph evaluates the output of the algorithm through the comparison of the 30 outputs with the expected "correct result". We represent on the Y axis the percentage of runs that led to such a result.

The complexity of the algorithm is exponential with the number of samples. The usage of only 16 samples already leads to correct semantic abstractions, and requires only 0.5 seconds to be performed. Even choosing a very conservative value such as $N = 50$, the algorithm completes execution in less than 2.5 seconds.

These results validate the performance of the region analysis algorithm in an optimal scenario: the samples are manually generated to ensure the maximum degree of variability in the random fields. While the choice of a synthetic sample allowed us to purely evaluate the performance of the algorithm independently

[1]The test was run on a Core 2 Duo 2.4GHz CPU with 2GB RAM.

from the initial sample, two additional degrees of freedom exist in the real-life usage of this algorithm: the *quality* of the initial sample, as well as the complexity of the protocol itself.

For instance, if a protocol field contains the IP address of the server being observed and if the sample is built through the observation of the interaction of a set of clients with a single server, region analysis will erroneously conclude that such a field belongs to a fixed region and will assign to it a strong semantic meaning. Similar problems can be encountered with timestamps: a sequence of UNIX epochs such as

1212669669.38
1212669680.8
1212669692.86
1212669699.82
1212669706.13

would lead to an inference such as *F("1212669")+M(3)+F(".")+M(1:2)* that would fail to match interactions 300 seconds after the last collected sample. It is clear from this consideration that the choice of a *good* training sample is extremely important for the correctness of the semantic inferences performed by region analysis. This challenge is at the foundations of the design of the SGNET deployment (Chapter 4). Through SGNET we will take advantage of a distributed deployment of sensors to maximize the spatial variability of the collected samples and thus try to reduce learning artifacts such as those previously described.

3.3 Building protocol FSMs

The region analysis algorithm is able to automatically infer from the statistical variability of a set of samples an approximation of the corresponding semantics. This approximation is obtained in a totally protocol-agnostic fashion: no assumption is made on the protocol structure, and no *a priori* knowledge is assumed on the protocol semantics. We proposed in [Leita 2006] an *incremental* algorithm taking advantage of the region analysis algorithm to build Finite State Machine representations from samples of protocol interaction.

The input of the algorithm is a flow of samples, generated asynchronously. Each sample is seen as a *conversation*, a sequence of client and server messages generated within a TCP connection or a couple of UDP request/answer messages. Each message of the conversation corresponds to the application-level payload (when present) of the packets composing the network trace, cleaned, in the case of TCP, of artifacts due to packet losses and retransmissions.

The algorithm aims at producing a Finite State Machine representing the protocol dialogue between a client and a server. This representation is derived from that of Deterministic Finite Automaton (DFA) commonly used in theory of computation to represent regular languages. We have modified the original model to represent

the dialogic nature of the represented language. A ScriptGen finite state machine can be formally represented as a tuple $(S, \Sigma_C, \Sigma_S, T, L, s, A)$ consisting of

- a finite set of states (S)
- a finite set called the client alphabet (Σ_C)
- a finite set called the server alphabet (Σ_S)
- a transition function ($T : S \times \Sigma_C \rightarrow S$)
- a labelling function ($L : S \rightarrow \Sigma_S$)
- a start state ($s \in S$)
- a set of accept states ($A \subset S$)

It is clear from this definition that a ScriptGen FSM aims at modeling the conversation from the point of view of the server. Transitions are associated with items of the client alphabet determining the transition upon reception of a client request, while each state is associated with a label representing the corresponding answer to be provided to the client. While the server alphabet consists of a set of byte sequences (the answers to the client requests), the client alphabet is more complex and can consist of semantic generalizations of the input messages generated by the region analysis under the form of regular expressions. The client alphabet is thus composed of complex items, which can be modeled with "lower level" DFAs.

The previous definition can be easily adapted to the representation of the protocol dialogue from a client perspective. As the goal of this work is the generation of emulators for server-side honeypots, we will focus our attention on the server-side FSMs only.

A number of existing honeypot deployments, such as Honeytank [Vanderavero 2004] and iSink [Yegneswaran 2004] maximize scalability by modeling TCP protocols in a stateless way. In most cases, in order to be able to correctly reply to a client request no additional context information is required. Under this perspective, the choice of modeling the application protocol interaction under the form of a finite state machine may seem questionable. The choice of following a stateful approach in the ScriptGen learning is motivated by the need to perform a *semantic clustering* of client requests in 'semantic groups' to be provided to the region analysis algorithm. We want to take advantage of the protocol finite state machine to build a semantic skeleton to be used to group together semantically similar messages. The intuition underneath this need is that the *context* in which a given message is seen has great influence on the semantics of the message itself. For instance, in the SMTP protocol the first client request in the TCP session is always a HELO message, while the QUIT request is likely to be at the end of the conversation. The refinement algorithm proposed here aims at exploiting the state of the connection and the relative positioning of the client requests to generate groups of messages very likely to have similar semantic value.

The refinement algorithm is composed of two separate phases. The first phase exploits the existing FSM skeleton and the concept of *bucket* to semantically cluster the input messages. The second phase applies the region analysis to the generated groups and generates new states and transitions according to the region analysis output.

These two phases are described in Section 3.3.1 and 3.3.2 respectively. A specific problem of the state generation, the state labelling, is separately addressed in Section 3.3.3 for the sake of clarity.

3.3.1 Classification phase

Algorithm 2 classify(*state*, *conversation*)

```
1: request,answer ⇐ pop(conversation)
2: trans ⇐ match(request,transitions(state))
3: if trans ≠ NULL then
4:    label(next(trans)).add_candidate(answer)
5:    classify(next(trans),conversation) {Recursive call}
6: else
7:    bucket(state) ⇐ (request, answer, conversation)
8:    cluster(bucket(state))
```

The classification phase classifies new samples taking advantage of the existing protocol FSM. The classification process is illustrated by Algorithm 2. Starting from the root, we use the sequence of requests in the incoming flow to traverse the existing edges of the state machine, matching the client requests to the existing edges of the FSM. If for a certain state no outgoing edge matches the client request, such request and the remaining training conversation is attached to that state's *bucket*. A bucket is a container for the new conversations; the first message of each conversation is a client request that, through the region analysis step, will lead to the generation of new transitions for the corresponding state.

By construction, the first requests of the conversations attached to a certain bucket share the same type of interaction history: they are all preceded by the same type of requests as it was identified by spanning them over the existing Finite State Machine. We are thus able to group together the messages on which the semantic inference will take place according to their *past* context. In the previously introduced example of the SMTP protocol, probably a HELO message will not be grouped in the same bucket as a QUIT message.

The *future* context is instead analyzed by looking at the partial conversations stored in the buckets. For each message stored in a bucket, we look at the length in bytes of the following server answers in the conversation. We take advantage of this value to *cluster* the content of the bucket and group together client requests belonging to conversations likely to have similar impact on the server. For each conversation C_i in the bucket, we compute its length L_i as sum of the lengths of all

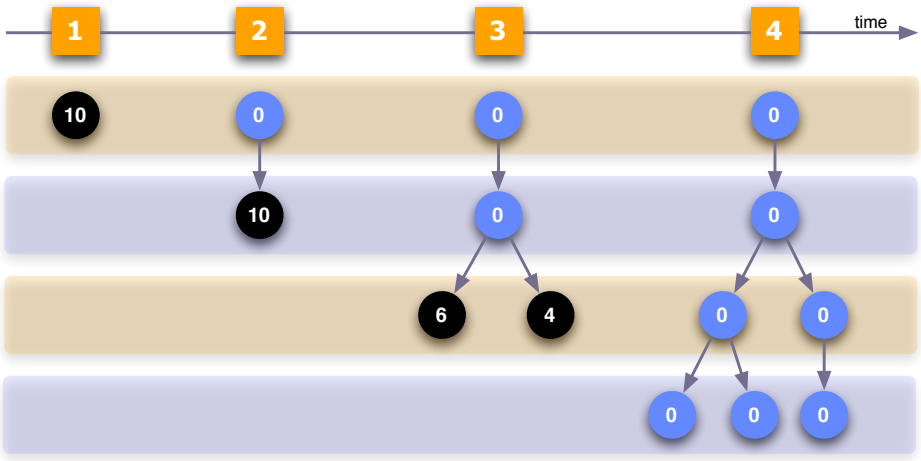

Figure 3.8: Iterative refinement

the future server answers following the unmatched client request.

The employed clustering algorithm is based on two separate phases:

1. We generate clusters grouping together conversations sharing exactly the same value of L_i. If multiple conversations share exactly the same value, they are likely to be associated with exactly the same impact on the server, which generate answers of fixed value.

2. We then build clusters of the remaining conversations keeping the *relative variance* of the lengths under a certain threshold. This second phase is used to handle protocols in which the server answer to semantically identical client requests lead to slightly differing server answers. Given a cluster $C = \{L_1, L_2, ...L_i\}$ we define its relative variance as $rVar(C) = var(C)/mean(C)$. We want in fact to tolerate variations of size proportional to the length of the server answers. The intuition underneath this requirement is that if an answer is short, it probably corresponds to an error code or error message. If an answer is instead very long, it probably corresponds to data content sent by the server to the client, such as an HTML page in the case of the HTTP protocol.

The whole clustering algorithm is sufficiently fast to be repeated every time a new conversation is added to the bucket. The result of such algorithm is the division of the initial bucket into multiple smaller buckets grouping together semantically similar client requests according to their past and future context.

3.3.2 Semantic inference phase

The previous phase has constructed groups of client requests likely to share similar semantic meaning according to the context of the conversation they are into. In the

semantic inference phase, we inspect in a breadth first traversal the buckets of all the FSM nodes generated by the classification phase. If a group contains a sufficient number of samples, the region analysis algorithm is used to infer the semantics of the messages. Each cluster generated as output of the region analysis algorithm leads to the generation of a new transition and of a new corresponding state in the protocol FSM. While each transition is labelled with the regular expression generated by the region analysis, the corresponding state needs to be labelled with the answer to be sent back to the client during emulation. The generation of such answer is not a straightforward process, as detailed in Section 3.3.3.

Upon completion of the breadth first traversal, if new nodes have been generated the classification phase is iteratively applied to the remaining messages of the conversations.

Figure 3.8 shows a simple example of iterative refinement. For each state in the diagram, the label corresponds to the number of training conversations in the bucket. A training set consisting of 10 flows is used to update an empty state machine. Since the state machine is empty, none of the initial client requests contained in the samples will match an existing transition. All the samples are thus initially put in the root bucket (step 1). The semantic inference phase then picks the training samples contained in the bucket, and applies the region analysis algorithm to the samples. Region analysis generates a different transition for each cluster of sample client requests believed to have a similar semantic meaning. In this first step, a single transition is generated. After the semantic inference phase, the classification phase is then triggered and the training flows are matched with the newly created transition. Since the state machine is still incomplete, the training samples do not find a match in the following state, and are thus stored in the corresponding bucket for the next refinement iteration (step 2). The process repeats until the semantic inference phase is not able to generate other transitions: this happens at step 4, in which the sample flows do not contain any further interaction between the attacking source and the server (client closes connection after having sent 3 requests to the server). The state machine is then complete.

3.3.3 Responding to clients

The update phase previously introduced updates a FSM with the information provided by a conversation sample. We take advantage of the region analysis algorithm to infer a partial semantic structure for the client requests, but while this structure is sufficient to build regular expressions able to identify the nature of an incoming client request, the generation of a correct answer theoretically requires full knowledge of the protocol semantic. We take advantage of a heuristic, which proved to be acceptable in most cases: randomly choose one of the answers given by the server in one of the conversation samples reaching the state. Such an approximation is acceptable for many protocols, but fails to correctly handle complex phases that proved to be critical in the early experimental validation.

During the first experiments with honeypots taking advantage of ScriptGen

FSMs to interact with clients [Leita 2005], we encountered peculiar results when handling interactions on complex protocols such as NetBios. In order to study the behavior of ScriptGen-enabled honeypots, we deployed a first experimental host in the same network as a high interaction honeypot and observed its behavior over a period of two months. More specifically, we focused on the activity generated by attacking IP addresses observed by the two technologies approximately in the same period. On port 139 TCP, a big discrepancy was detected between the two honeypot technologies. While clients were carrying on conversations with high interaction honeypots composed of more than 6 exchanges of client request and server answers, the same clients were leaving the conversation with the ScriptGen honeypot after receiving the second server answer.

An explanation for this fact can be found an in depth analysis of the nature of the interactions on port 139 TCP. Port 139 TCP is associated with NetBT, the NetBios implementation running over TCP/IP. This port was historically associated with the Server Message Block protocol used by Microsoft systems to access file and printer shares, and has been replaced in Windows 2000/XP by port 445, which allows direct access to the SMB protocol without the additional NetBT layer. The first step of any conversation on port 139 corresponds to a session establishment phase, consisting of a session request in which the client identifies itself and the corresponding answer from the server. Only after this preliminary phase the additional SMB layer carries on the more complex interaction. In our experiments, the honeypots have been able to correctly carry on the session establishment with the client, but failed in correctly handling the SMB protocol.

The heuristic previously introduced for the generation of server answers is based on the assumption that for any client request of the same type the server answer does not change. While this is true in many cases, such as the NetBT session establishment, we can identify two main classes of protocol fields that contradict this assumption.

- **Timestamps.** If the server answer contains information on the freshness of information under the form of timestamps, the value of such timestamps will vary for two equal requests performed at different times. While such fields violate the assumptions of our heuristic, such violation does not normally lead to emulation problems. Even if the answer given by a ScriptGen-based honeypot uses always the same value for this timestamp, in most protocols such timestamp will be accepted by the client and interpreted as a simple time skew in the server clock. Nonetheless exceptions exist to this assumption: protocols such as Kerberos [URL 17] are sensitive to the relative time differences.

- **Cookie fields.** In many protocols, one of the two peers involved in the conversation chooses a cookie value to be put in the message. This value can be incremental, randomly generated or derived from the internal system clock. For a correct protocol interaction, the other party must reuse such

identifier or function of it in the next interaction step. For instance, in the SMB protocol the client reports its process ID in a specific protocol field, and the server answer is valid only if the same value is used in its header.

We initially assumed that the simplistic implementation of attack scripts would not have been sensitive to an incorrect handling of cookie fields. The analysis of the information collected in [Leita 2005] contradicted our assumptions and led us to the conclusion that a correct handling of the cookie fields is necessary to carry on a correct conversation with the attacking clients. In [Leita 2006] we introduced a more sophisticated handling of the server answers in order to infer content dependencies.

In order to identify content dependencies, it is necessary to correlate the content of client requests with the content of the following server answers for all the conversations of the training sample. We can exploit the statistical diversity of the samples to reliably infer content dependencies and filter out casual matches. The process is composed of two separate steps: *link generation* and *consolidation*.

3.3.3.1 Link generation

The link generation algorithm takes into consideration each request contained in the training set, enriched by the output of region analysis, and correlates it with all the following server answers contained in the corresponding training conversation.

For each sample request, the algorithm correlates each byte belonging to mutating regions with the server answers using a correlation function. In the most simple case, the correlation function returns 1 if the bytes match, and returns 0 if the bytes differ. For each encountered match, the algorithm tries to maximize the number of consecutive correlated bytes starting from a minimum of two.

Let's consider here an example. For a certain set of messages, the region analysis generated the following output, in which the regions have been numbered for convenience:

$$F_1(\text{“PATH:/”})+M_1(1{:}10)+F_2(\text{“ FILE: ”})+M_2(1{:}10)+F_3(\text{“.”})+M_3(1{:}3)$$

Two conversations belonging to the training set are considered. Each conversation is composed of a client request, modelled by the previous sequence of regions, and the corresponding answer from the server:

1. R1: `PATH: /root FILE: bash.rc`
   A1: `FILE: bash.rc OWNER: root MODIFIED: 22:13`

2. R2: `PATH: /images/.hidden FILE: pic123.jpg`
   A2: `FILE: pic123.jpg OWNER: user MODIFIED: 12:34`

The server answer has thus a *content dependency* with the previous client request. The algorithm should thus model the first answer as:

FILE: $L(M_2).L(M_3)$ OWNER: root MODIFIED: 22:13

where the notation $L(M_2)$ represents a content dynamically generated from M_2. Such a model will still introduce approximations during the emulation. Using the above answer for all the semantically similar client requests, all the queried objects will be reported as owned by root and modified at 22:13.

In the context of this work, we call the pointers to previous content in the client requests *links*. Two issues need to be addressed in the practical implementation of the link generation algorithm.

Firstly, the practical identification of the content of a mutating region for a given client request is not straightforward. For this reason, the identification of the content target of a link is more sophisticated. Instead of referring to the mutating region containing the content, we refer to the message boundaries (Start Of Message, SOM and EndOfMessage, EOM) or to *significant* fixed regions acting as boundaries for the mutating content. A significant fixed region is a region that has always an unambiguous position within the requests of the sample set. For instance, in the example F_3 is not significant: for the request R2, it is matched in two different positions. With respect to R1 and R2, F_1 and F_2 are instead significant. In the link generation process the region F_3 is thus not considered and is merged with the two mutating regions M_2 and M_3 in a single mutating area whose content can be referenced as $L(F_2 + 0 : EOM - 0)$ (a byte offset can be added).

Secondly, running the link generation process on each sequence is likely to generate a number of accidental correlations. Referring to the two conversations in the example, the following links would be generated in the two answers A1 and A2:

FILE: $L_1(F_2 : EOM)$ OWNER: $L_2(F_1 : F_2)$ MODIFIED: 22:13
FILE: $L_1(F_2 : EOM)$ OWNER: user MODIFIED: $L_3(F_2 + 3 : EOM - 5)$:13

The link L_2 is generated by the accidental coincidence of the directory name and the user name in the first conversation, while link L_3 is generated by the appearance of a substring of the filename in the modification time of the second conversation. We need a method to filter out these matches and leave only those that appear in the majority of the training conversation. This is achieved by the consolidation phase.

3.3.3.2 Consolidation

The input to this consolidation phase is a set of "proposals" for the content of a server answer generated by the previous link generation. The algorithm takes into consideration each byte and compares the content of each proposal for that byte. This content can be either a link or the value of the answer in the original training file. The most recurring content is put in the consolidated answer, while the other ones are discarded. All the proposals having a content for that byte differing from

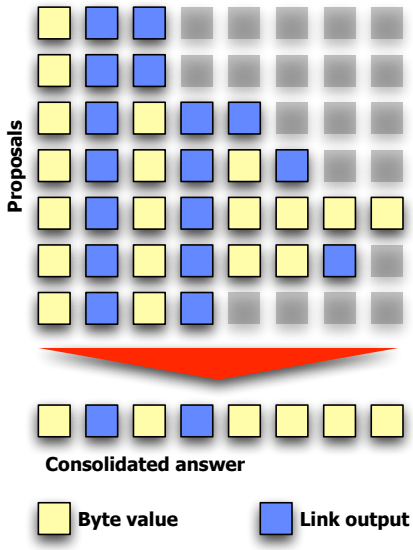

Figure 3.9: Consolidation

the chosen one will not be taken into consideration any more for the remaining bytes.

Figure 3.9 represents the consolidation behavior in a very pessimistic case. In this case, the number of misleading links is as high as the number of proposals. The algorithm is such that the consolidated answer will always be equal to at least one of the proposals. Also, increasing the number of training samples will increase the number of proposals, therefore increasing the robustness to misleading links. The number of valid proposals at the end of the algorithm can be considered as the confidence level for the correctness of the consolidated answer.

During emulation, the link information is used to transform the referenced content of the requests and provide the content for the server answers. Using the significant fixed regions as markers, and offsets to specify relative positions, it is possible to correctly retrieve variable length values in most cases. The success of the algorithm depends in fact on the presence of a sufficient number of significant fixed regions.

What has been stated herein with reference to simple equality relations can be extended to other types of relations, such as incrementing counters, by simply defining different types of links. We left this task for future work, as experimentation did not highlight the need for it as far as we can tell.

3.4 Emulation

Taking advantage of the resulting FSMs to emulate the protocol interaction is straightforward, and consists in determining the future state from the current state

according to the incoming client requests. Once a new state is reached, its label (and the associated links) is used to respond back to the client.

The emulation of the protocol interaction with clients presents two challenges that are worth being described more in depth: determining the correct future state according to the incoming client request and deciding whether the connection needs to be terminated. These two challenges are discussed in the rest of this Section.

3.4.1 Choosing the future state

For a given state, the choice of the correct future state according to the incoming client request can be performed in two ways: by tolerating imperfect matches or by requiring perfect matches only.

One may choose to be as robust as possible to new activities or to imprecise choices in the generation of the regions by tolerating imperfect matches. Even if the regular expression generated by the region synthesis does not perfectly match an incoming request, the corresponding transition can be chosen as long as it leads to the maximum possible overlap of the content of the fixed regions with that of the incoming client request. This choice although leads to two major drawbacks. First of all, tolerating imperfect matches between the incoming request and the known transitions might lead to the choice of a wrong transition generating a completely wrong answer, corrupting the conversation. Also, distinguishing imperfect matches from new activities becomes impossible.

For the above reasons, we choose to take into consideration only *perfect* matches. Even in this case, the way in which region analysis builds the semantic abstractions does not ensure that an incoming message will be matched univocally by one transition. For instance, microclustering produces refinements to the output of the macroclustering, generating specifications of a more generic transition. Both the microclustered transition and the generic one will match a given incoming request. It is thus important to take advantage of a *matching function*.Given an incoming message M and a set of transitions $S = T_i, i \in [0 : k]$, a matching function associates each transition to a matching score $k = f(M, T_i)$. The transition with the highest matching score is chosen to determine the future state.

Algorithm 3 *score* = matching_function(M,T_i)

```
1: if not(matches(M,T_i)) then
2:    score ⇐ −inf
3: else
4:    for region in get_regions(T_i) do
5:       if region is fixed then
6:          score ⇐ score+ len(region)
7:       else if region is mutating then
8:          score ⇐ score − 10
```

The matching function used in this work is detailed by Algorithm 3. The

purpose of this matching function is to ensure that, among multiple candidates, the most specific transition is chosen. With specificity of a transition we refer to the amount of bytes of the incoming request that are considered as fixed according to the semantics defined by the transition. If a transition matches the incoming message, the scoring function gives a positive score of 1 to each byte matched by a fixed region, and adds a penalty of -10 for every mutating region matching the message, regardless of its size. The scoring function introduces thus two additional tuning parameters, that weight the penalties associated to the different types of regions.

3.4.2 Final states

Each FSM representation is associated with a single *start state* that, in case of TCP protocols, is univocally associated with the beginning of the session. In the formal definition of the FSM in Section 3.3 we also defined a set of accept states A corresponding to the final states of the protocol interaction. The characteristics of the TCP protocol allow us to distinguish among two classes of accept states: client-side and server-side. The emulation of the two cases has different characteristics.

In most protocols, the termination of the session is initiated by the client. For instance, in the SMTP protocol the client actively closes the TCP connection as soon as the email data is delivered. States corresponding to client-side termination are not necessarily leaves of the ScriptGen FSM representation. For instance, an activity searching for SMTP open relays is likely to terminate the connection with the server immediately after having received an answer to the RCPT TO command, traversing only a portion of the traversal normally associated with the SMTP activity. The ScriptGen-based emulator does not need to actively handle these cases: when the client terminates the connection, the emulator reacts to it by simply removing the corresponding contextual information (e.g. pointer to the current state, link values for the content dependencies, ...).

In some protocols or in case of error conditions, the connection is instead terminated by the server. States corresponding to server-side termination are by construction leaves of the ScriptGen FSM representation. Every time in which these states are reached, the emulator actively terminates the connection preventing the client from continuing the interaction.

3.5 Lab-based experimentation

The practical applicability of the ScriptGen approach to the learning of Internet attacks will be explored in depth in Chapter 4 through the implementation of the technique in a distributed honeypot deployment, SGNET. While this large scale experimentation will allow us to have quantitative information on the behavior of the ScriptGen approach in handling the multiplicity of Internet attacks, it is interesting to evaluate here the quality of the semantic abstraction performed by

Name	Disclosure	Affected port	Latest affected configuration
MS04-011 (LSASS)	13/4/2004	139/445 TCP	Windows XP SP1
MS02-056 (MS SQL Server Hello)	6/8/2002	1433 TCP	MS SQL Server 2000 SP1
MS02-045 (SMB Nuke)	22/8/2002	139 TCP	Windows XP SP0
MS06-040 (Netapi)	8/8/2006	139/445 TCP	Windows XP SP1

Table 3.2: Analyzed vulnerabilities

ScriptGen on a small number of significant examples taking advantage of lab-generated datasets.

The experiments proposed here focus on the nature of the semantic abstraction and on its ability to handle future instances of the attack, and consider solely the specific network interaction required to exploit the vulnerability. The emulation of the interactions consequent to a successful exploitation of a victim host will be extensively addressed in Chapter 4.

Among the exploits taken into consideration during the evaluation of ScriptGen's semantic inference, we selected a subset of them associated with the four vulnerabilities represented in Table 3.2.

We have considered public domain implementations of exploits hitting these vulnerabilities, taking advantage of the milw0rm repository [URL 23] and of the latest version of the Metasploit Framework [URL 44], which was shown to be responsible for a non-negligible portion of the Internet attacks in [Ramirez-Silva 2007].

We have chosen these vulnerabilities to investigate the behavior of the ScriptGen semantic inference in handling protocol interactions of different complexities. The MS04-011 vulnerability is investigated taking advantage of a very simple exploit tool. The MS02-056 vulnerability is used to show the ability of the semantic inference to handle semantically structured client requests. The MS02-045 vulnerability is demonstrated taking advantage of a more sophisticated client that performs basic checks on the correctness of the received answers and generates content dependencies among messages. The MS06-040 vulnerability is finally used to show the difference between a very simple proof of concept code obtained from milw0rm and a more sophisticated implementation part of the Metasploit Framework.

The training sample was generated by repeating each exploit every 90 seconds against a vulnerable host for 100 times. We have tried to evaluate in the experiment the impact of variability of certain fields on the protocol learning. To do so, we have forced the reboot of the system at each exploitation attempt. This ensures to maintain of the application entropy (e.g. process IDs), entropy that would have been otherwise be significantly reduced by reverting the system to a pre-computed snapshot at each instance of the experiment.

We have chosen instead of not taking into consideration the variability of IP addresses within the protocol stream. The client and the server are always associated to the same IP address through all the iterations. As previously discussed, the problem of the variability of the IP addresses will be addressed more in depth

in the next Chapter, where the samples will be generated by the operation of a distributed deployment of honeypot sensors monitoring different networks of the IP space.

We took advantage of the generated sample to produce a FSM representing the interaction, and we took advantage of this FSM in a simple emulator, taking advantage of it to interact with clients. We then observed the interaction of the protocol emulator with the exploit tool, observing both the concordance of the network interaction with the expected behavior by manually analyzing the network traces and the output of the attack tool itself.

3.5.1 MS04-011 Vulnerability (LSASS)

The MS04-11 vulnerability, disclosed on the 13th of April 2004, was brought to the attention of the press for its exploitation in the spread of the Sasser worm. The vulnerability allows a stack-based overflow in certain Active Directory service functions in LSASRV.DLL of the Local Authority Subsystem Service (LSASS) in different versions of Microsoft Windows. In order to reproduce the exploitation of this vulnerability, we took advantage of a proof-of-concept exploit developed by a Russian hacker called *HouseOfDabus* [URL 22].

In order to generate the interaction sample, we tried to reproduce the *modus operandi* of an attacker manually using the exploit. We first executed the exploit in test mode: in this mode, the tool attempts to authenticate with the SMB service using a *Session Setup AndX* request. In case of successful authentication, the server response contains the OS version. The knowledge of the OS version is required for the correct choice of the offset to be used in the exploitation phase. Once the OS version is identified, we re-run the tool to perform the actual exploit. As previously explained, we do not take into consideration here the network interaction following the successful exploit. The emulation of such interaction generates in fact dependencies between two separate TCP sessions, and thus between two separate FSM representations. We will investigate the feasibility of representing these dependencies in Section 4.3.

Figure 3.10 shows the FSM representation generated by the ScriptGen algorithm for the generated sample. In the schema, each state is labelled with its frequency, which is the number of samples traversing it. Each transition is labelled with a numeric identifier that helps to uniquely represent traversals. For instance, the whole traversal of the FSM in Figure 3.10 can be represented as *111111111*. Buckets containing samples not yet parsed by the region analysis algorithm are represented as squares and labelled with the number of enclosed samples.

The FSM was generated using only 10 samples for the Region Analysis algorithm. No difference is encountered by using a higher number of samples for the semantic abstraction. Some interesting facts can be derived from the analysis of the generated FSM.

Firstly, the interaction involved in the preliminary OS version test is actually a subset of the exploitation phase. The overlap of the samples generated by the two

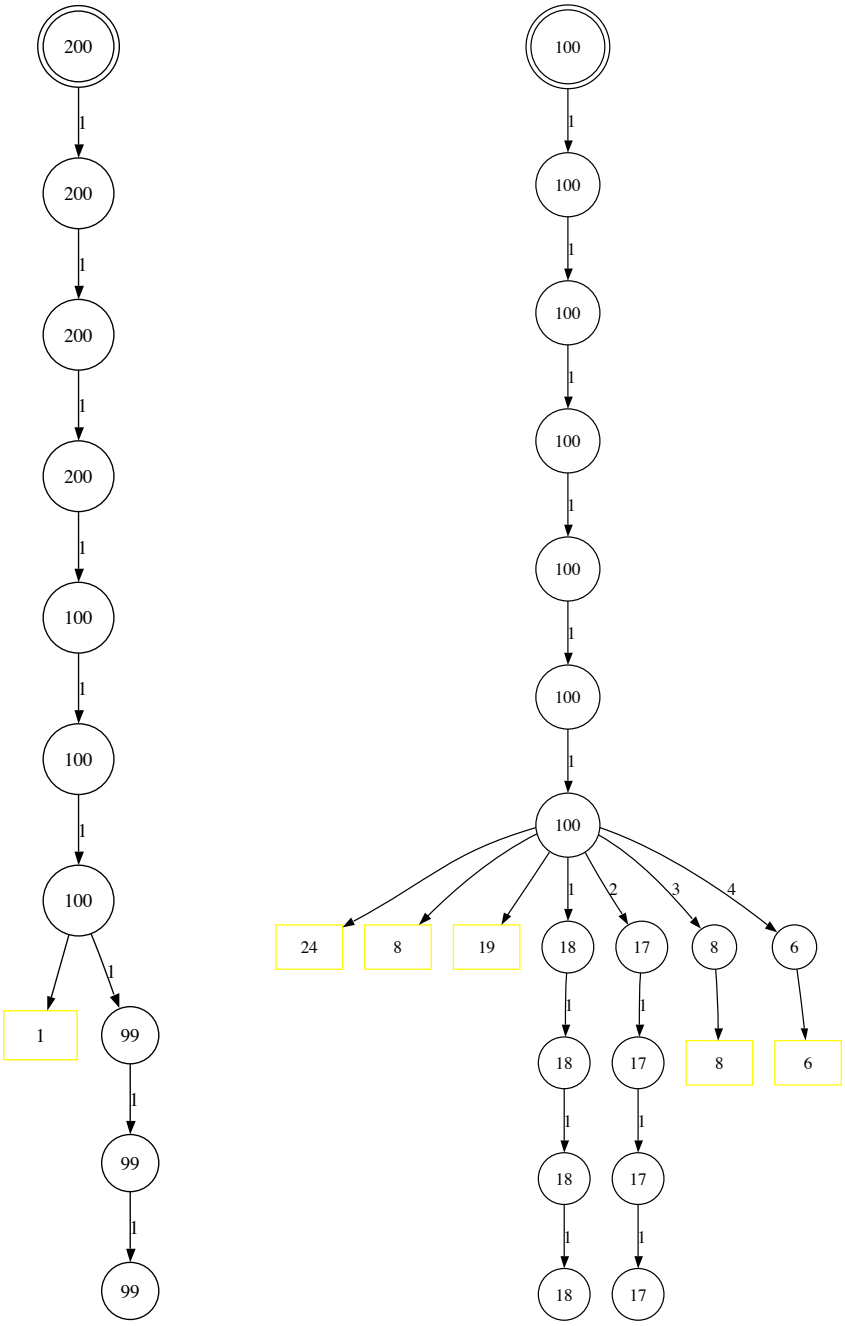

Figure 3.10: FSMs for MS04-011 (left) and MS06-040 (right)

activities results in fact in a single path, in which the traversal *111* (corresponding to the OS version test) was traversed twice the times of the traversal *111111111* (corresponding to the real exploit). This is confirmed by an analysis of the exploit code: if the OS version test is enabled, the program quits after having sent the first three client requests of the exploit script and having retrieved the OS version. This is thus an example of client-side session termination whose corresponding state is not a leaf of the FSM, as previously explained in Section 3.4.2.

Secondly, the protocol interaction generated by this exploit script is extremely simple. The output of the region analysis corresponds almost always to a single fixed region corresponding to each client request. The only exception is the 7th client request, which is composed by two fixed regions separated by a one byte mutating region. An interpretation for this surprising result is given by the analysis of the exploit source code. All the client requests are hard-coded in the exploit code under the form of a binary blob. Also the typically mutating fields of the SMB protocol, such as the process ID previously encountered in this chapter, are always assigned the same value.

An interesting fact that appears in Figure 3.10 is the presence of a single sample in a bucket at node *111111*. In one of the iterations of the sample generation, the LSASS service crashed prematurely and closed the connection instead of replying to the client request as it was supposed to. The ScriptGen algorithm interpreted the lack of answer to the client request as a semantic difference, which led to the decision of putting the sample in a different bucket than the other semantically similar client requests. Depending on the technique used to generate the samples, we can expect the generation of a considerable number of *stale* buckets, composed of only a few samples erroneously attributed to a different semantic branch because of erroneous behavior of one of the two parties involved in the conversation. We can easily address the problem tracking the date of last modification of a bucket in order to detect unused small buckets likely to correspond to these situations.

The exploit considered here proved to be extremely simple and with an extremely low degree of sophistication. The semantic abstraction generated by ScriptGen easily emulated the protocol interaction during the experiment, perfectly replicating the behavior of the real server used to build the samples. An attacking client using this exploit tool would have been unable to recognize the presence of a honeypot emulating the exploit.

3.5.2 MS02-056 Vulnerability (MS SQL Server Hello)

The MS02-056 Vulnerability, disclosed in August 2002, affects all the versions of Microsoft SQL Server 2000 and Microsoft SQL Desktop Engine 2000 previous to the Service Pack 3. By sending a specially crafted login request to TCP port 1433, an unauthenticated remote attacker can overflow a buffer and gain system level privileges on the victim. The protocol adopted by SQL Server on port 1433 is the Tabular Data Stream (TDS) protocol, initially designed and developed by Sybase Inc. in 1984, and later adopted by Microsoft. In order to reproduce the exploitation

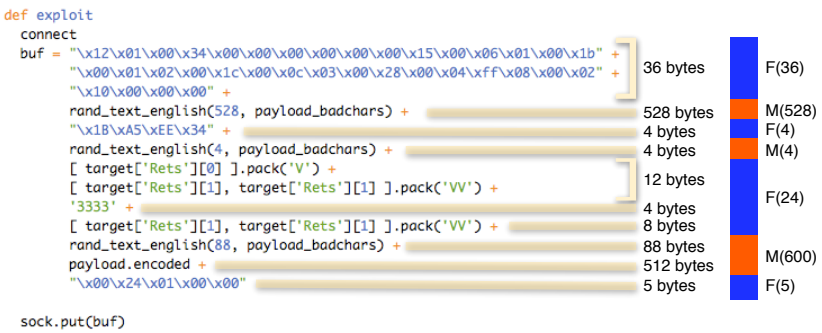

Figure 3.11: Metasploit implementation of the MS02-056 exploit

of this vulnerability we took advantage of the module *windows/mssql/ms02_056_hello* in Metasploit 3.1. The module was run taking advantage of the *windows/exec* payload, which forces the victim system to execute a command already present in the filesystem of the host, in our case the *shutdown* command to reboot the OS.

Differently from the previous exploit, consisting of several exchanges of client and server messages in order to lead the vulnerable system to the failure state, the vulnerability exploited in this case lies in the very first authentication request sent by the client. The exploitation of the vulnerability crashes the server, which never answers to the request. The corresponding FSM generated by ScriptGen reflects this condition, and is composed of a single transition leading to a state with empty label. Differently from the previous case, the output of the region analysis for the transition is more complex:

Type	**Size (bytes)**	**Content**
Fixed	36	*12 01 00 34 00 00 00 00 ...*
Mutating	528	
Fixed	4	*1B A5 EE 34*
Mutating	4	
Fixed	24	*BA 8A B6 42 50 1E D0 42 ...*
Mutating	600	
Fixed	5	*00 24 01 00 00*

This output reflects perfectly the implementation of the exploit. Figure 3.11 shows the implementation of the MS02-056 exploit used in the experiment, and its correspondance with the output of region synthesis. This validates once more the ability of the region synthesis to correctly infer the semantically important portions of the protocol, generating a FSM that is robust to the randomly generated parts such as in this example. The emulator interacting with the exploit tool taking advantage of the generated FSM was able to correctly handle and emulate any future instance of the attack.

3.5.3 MS02-045 Vulnerability (SMB Nuke)

An interesting vulnerability is MS02-045. Differently from the other vulnerabilities analyzed in this work, this vulnerability cannot be used to force the execution of code on the remote host. By sending a specially crafted packet request on port 139, the attacker can mount a denial of service attack on the target machine and crash its OS. We reproduced the attack taking advantage of a proof of concept tool developed by Frederic Deletang in 2002 [URL 10]. After every attack, the system was crashing with a blue screen that led to the automatic reboot of the host, achieving the same result obtained in the previous exploits through the execution of a shutdown command on the victim.

Also in this case, ScriptGen has been able to infer a correct representation of the FSM using only 10 samples of interaction as input to the region analysis algorithm. No difference was detected when running the algorithm over more samples. The result of the algorithm is a FSM composed of a single path composed of 6 states. An interesting difference with respect to the previous LSASS exploit is the complexity of the exploit implementation. While in the LSASS exploit the exploit script did not parse the server answers and produced the client requests from a set of predefined binary blobs, this exploit implements a parser for the SMB protocol that is used for both checking the client requests and the server answers. Mutating fields in the SMB header such as the process ID are randomly generated by the exploit, and the more realistic behavior of the client makes the ScriptGen learning more challenging, since the complexities inherent with the SMB protocol start to appear.

The presence of cookie fields in the SMB protocol leads to the generation of content dependencies between client requests and server answers. We can observe the correct inference of *links* between client request and the successive answer to handle the process ID of the SMB protocol, which are then correctly used during the emulation of the exploit.

3.5.4 MS06-040 Vulnerability (NetApi)

The MS06-040 vulnerability is a relatively recent vulnerability linked to the NetApi32 CanonicalizePathName() function. The attacker is able to exploit the vulnerability using the NetpwPathCanonicalize RPC call on the SMB service through interaction on TCP ports 139 or 445. The wide amount of software configurations vulnerable to exploitation of this vulnerability (ranging from Windows 2000 SP0-SP4+, to Windows XP SP0-SP1, to Windows 2003 SP0) makes this vulnerability extremely interesting to the more recent threats. We selected two different exploit implementations, with different degrees of complexity.

The first exploit implementation is a proof of concept C program written by the Rootshell Security Group and publicly available on milw0rm [URL 21]. This exploit follows the lines of the LSASS exploit previously analyzed, and is composed of a list of client requests stored in the form of binary blobs that are sent one after the other to the victim host. Any server answer received during the interaction

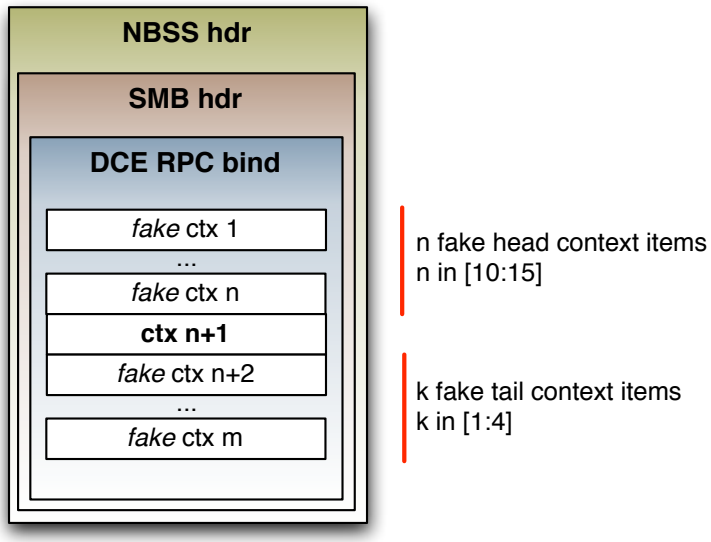

Figure 3.12: Metasploit multi-context bind call

is simply discarded by the script. The resulting FSM generated by ScriptGen is extremely simple, and is composed of a single path of 16 states. All the transitions inferred by ScriptGen are composed of a single fixed region: all the client requests sent by the clients are constant strings that never change across two different runs of the exploit.

A more challenging exploit implementation can be found within the Metasploit 3.1 framework. We took advantage of the *windows/smb/ms06_040_netapi* module in conjunction with the *windows/exec* payload to exploit the vulnerable server and force the reboot of the host as in previous experiments. The results of the ScriptGen semantic abstraction when using these samples have been much different from the expectations, as shown in Figure 3.10 on page 54. The 7th client request leads to an unexpected number of transitions. The bucket-level clustering in fact puts the different samples in multiple clusters characterized by different lengths of the server answers. Only two of these clusters lead to a correct semantic inference, which leads to the generation of two transitions matching 18 and 17 samples respectively.

The reason for this result lies in the high level of sophistication of the exploit implementation. The Metasploit framework implements an almost full-fledged SMB client that tries to maximize the realism of the network interaction in order to challenge the detection of the exploits for Intrusion Detection Systems. One of the features implemented by the framework consists in the usage of *multi-context bind calls* when binding itself to a given RPC service. The 7th client request in the network interaction is in fact a DCERPC bind request, in which the exploit tries to bind to a given RPC service. The structure of the bind request generated by the Metasploit framework is schematized in Figure 3.12. The purpose of the bind request is to associate the client to one or more UUID identifiers corresponding to the

required services. Metasploit hides the required UUID (in this case 4b324fc8-1670-01d3-1278-5a47bf6ee188, corresponding to the Windows Server service) among a set of dummy contexts associated with randomly generated UUID identifiers. A number $n \in [10 : 15]$ of contexts precedes the "real" context, while a number $k \in [1 : 4]$ of contexts follows it. This behavior not only leads to a client request of random length and random structure, but also leads to a server answer with similar characteristics: the server answer will contain the list of answers, one for each of these contexts.

The usage of such obfuscated bind calls fully explains the behavior of the ScriptGen learning when facing those requests. Since the server answers have different length, the client requests leading to each length are grouped in different buckets since a different semantic value is assumed. This choice is correct: a given combination of values for (n, k) leads to a different semantic structure that requires a different answer. We can identify 24 different combinations of value, which should theoretically lead to 24 different branches. The insufficient amount of samples in the experiment leads to the generation of only two of them.

The results of the emulation of the protocol using the generated FSM are consistent with the assumptions. During 100 exploitation attempts on the ScriptGen emulator, 19 of them traverse the path *1111111111* and lead to the correct emulation of the protocol, and a corresponding success message in the exploit tool. Path *1111112111* leads instead to an error message, saying that the server answer is shorter than expected. This is due to the fact that the bucket level clustering grouped together samples having *similar* values of (n, k). We can assume that with a bigger training sample this would not have happened.

The complexity and the sophistication of the Metasploit framework challenge the learning capabilities of the ScriptGen approach. Still, even in these unfavorable conditions we have been able to correctly emulate 20% of the attacks, and detect a missing transition in 63% of them. Only 17% of these attacks failed due to an incorrect grouping of the samples performed by the bucket-level clustering. This incorrect grouping was due to the requirement of only 10 samples for the region synthesis, and driven by the lack of a sufficient number of samples in the experiment.

Such a case shows the limited capability of the ScriptGen approach to handle correctly variability in the protocol structure. While the alignment algorithms allow to take into consideration the variability in the different protocol fields, the random composition of such fields into variable structures cannot be easily modelled by ScriptGen. Thanks to the semantic clustering, each different structure is clustered in a different bucket and will eventually need to a distinct transition per structure, with a significantly increased resource consumption and learning time.

To isolate the problem and validate the capability of ScriptGen of otherwise handling correctly the network activity, we tried to disable the bind call obfuscation and repeat the experiment in these more favorable (even thought unrealistic) conditions. ScriptGen has then been able to correctly generate a single-path FSM correctly modelling the protocol interaction, and we have been able to use this FSM

to correctly emulate the interaction with the Metasploit module.

3.5.5 Comparison with Nepenthes

In order to have an idea on the usefulness of the ScriptGen ability to automatically learn the protocol interaction, we have compared the exploit emulation obtained in the previous analysis with that obtained by Nepenthes [Baecher 2006] and its python counterpart Amun [URL 13] in handling the previously analyzed exploits. We consider in fact these two tools as the most advanced and most interactive low-interaction honeypots freely available in the current state of the art.

When running the LSASS exploit, both Nepenthes and Amun are able to lead the exploit into sending all its client requests. Even if the honeypots do not emulate correctly the SMB protocol and simply send back random content, we have seen that the exploit implementation does not perform any check on the server answers. The level of interaction provided by the honeypots is thus sufficient to carry on the conversation.

Things get worse with respect to the MSSQL Hello exploit. Neither Nepenthes nor Amun provide emulation for the MS02-056 vulnerability, and do not listen on port 1433 TCP. An attempt to exploit a Nepenthes or Amun honeypot simply leads to an error message notifying of the fact that the connection was refused.

Focusing on the capture of self-propagating malware, the MS02-045 vulnerability is not taken into consideration by either of the approaches. Since they do provide emulation of the SMB protocol to emulate vulnerabilities such as the LSASS exploit, the SMB Nuke exploit is able to connect and start the protocol interaction with the honeypots. But as soon as the first server answer is received by the exploit script, the exploit code tries unsuccessfully to parse it and quits.

Finally, the MS06-040 vulnerability is not supported by Nepenthes but is supported by Amun, which ships with an increased number of vulnerability modules that have not been back-ported to Nepenthes. The support for the vulnerability is again implemented in an extremely simplistic way, which is enough to handle the extremely simple milw0rm proof of concept exploit. When facing the highly sophisticated SMB client shipped in the Metasploit framework, an anomaly is detected at the very first packet exchange:

```
[-] Exploit failed: Login Failed: The SMB response packet was
                                  invalid
```

Surprisingly enough, the honeypot does not seem to detect the failed exploitation attempt and does not provide any feedback to the user about the event in its logs. The maintainers have been notified of the issue.

3.6 Conclusion

We have proposed in this Chapter a novel application of protocol informatics algorithms in order to build semantic-aware representations of the protocol interaction.

We started from a set of samples generated by the interaction of a real client with a real server implementing that protocol and we inferred from it an FSM representation of the protocol language. The learning algorithm is totally automated and protocol agnostic: no assumption is made about the semantics of the protocol, and the behavior of the algorithm is based on a set of thresholds whose influence has been investigated in this Chapter. We have shown how ScriptGen is able to correctly learn and model the network interaction for complex binary protocols. ScriptGen has been able to take advantage of a very small number of interaction samples (10 samples proved to be enough in most cases) to "reverse" the behavior of exploit implementations of varying complexity and outperform the exploit emulation capabilities of widely used honeypots such as Nepenthes.

While ScriptGen consists in a very important step towards the increase of the level of interaction of low-interaction honeypots, it is not enough alone to collect meaningful data on attack threats. In order to correctly emulate the attack trace associated with the propagation of modern malware, we need to "understand" and emulate code injection attacks. Differently from Nepenthes vulnerability modules, ScriptGen FSM paths presented in this Chapter do not provide any information on the behavior on the attack *after* the successful exploitation of a vulnerable function. We will show in the next Chapter how we have been able to exploit the characteristics of ScriptGen to overcome these limitations and build a distributed framework for the collection of in depth information on attack threats.

Chapter 4

SGNET: deploying ScriptGen

Contents

In the previous Chapter we presented ScriptGen, an automated algorithm to abstract semantics from samples of network interaction and represent it under the form of Finite State Machines. ScriptGen learning can be used to successfully model the interaction involved in attack tools of various complexity, and allows the automated generation of responders for protocols. In this Chapter we show how we have been able to exploit the characteristics of ScriptGen to build a distributed framework for the collection of data on Internet threats, called SGNET.

SGNET main characteristics can be summarized in the following points:

- **Protocol agnosticism.** Differently from existing low interaction honeypot solutions, we want to avoid any *a priori* assumption on the structure of the network interactions that SGNET honeypots are going to face.

- **0-day reactiveness.** We want to leverage the ScriptGen characteristics to detect the presence of previously unknown activities and react to them by refining the FSM knowledge.
- **Code injection dissection.** We want to take advantage of the increased interaction level to study code injection attacks, retrieve in-depth information on the structure of the attack trace, and download malware samples.

4.1 Introduction to code injection attacks

In Chapter 3 we have evaluated the ability of ScriptGen emulators to handle the network interaction associated with exploitation tools. The network interaction evaluated in that context is only a portion of the attack trace involved in a code injection attack. In order to clearly dissect the code injection attack trace, we reuse a model introduced by Crandall et al. in [Crandall 2005]. The *epsilon-gamma-pi* model was introduced by the authors to describe the content of a code-injection attack as being made of three parts.

Exploit (ε). A set of network bytes being mapped onto data which is used for conditional control flow decisions. This consists in the set of client requests that the attacker needs to perform to lead the vulnerable service to the control flow hijacking step.

Bogus control data (γ). A set of network bytes being mapped onto control data that hijacks the control flow trace and redirects it to someplace else.

Payload (π). A set of network bytes to which the attacker redirects the vulnerable application control flow through the usage of ε and γ.

The payload that can be embedded directly in the network conversation with the vulnerable service (commonly called shellcode) is usually limited to some hundreds of bytes, or even less. It is often difficult to code in this limited amount of space complex behaviors. For this reason it is normally used to force the victim to download from a remote location a larger amount of data: the malware. In the context of this work, we propose an extension of the original epsilon-gamma-pi model in order to differentiate the shellcode π from the downloaded malware μ. A code injection attack can be characterized as a tuple $(\varepsilon, \gamma, \pi, \mu)$.

It is important to understand that the epsilon-gamma-pi-mu model used in this work does not exhaustively characterize all possible interactions that could take place in the case of a code injection attack. In fact, such a model implies as a final result of a code injection the upload and execution of a single malware to the victim. According to our experience, this is unlikely to be the case in real world. It is relatively common for a malware sample (what we could call stage-1 malware) to download from the Internet further "stages" upon execution. Such stages can consist of keyloggers, or even updates for the bot itself. While this work will not take into consideration directly this additional interaction, we will show how we are able to collect partial information on these interactions taking advantage of the behavioral information offered from external information sources in Section 5.1.2.

4.2 Handling 0-day attacks

The ScriptGen emulation as defined in the previous Chapter is composed of two distinct phases: a first phase takes advantage of a set of samples to generate a FSM representation of the interaction, and a second phase takes advantage of the generated FSM to emulate the protocol. This structure implicitly assumes a static view on Internet attacks, in which the emulation phase faces attacks that have already been seen in the previous training phase.

This situation is unrealistic. We expect Internet attacks to evolve with time, in terms of discovery of new vulnerabilities and thus 0-day exploits, but also in terms of variations of known exploits through different implementations. We need a mechanism to react to activities never observed before and dynamically refine the FSM knowledge, closing the loop between the training and the emulation phase. We achieved this with a *proxying algorithm* initially proposed in [Leita 2006] and then refined in [Leita 2008a].

4.2.1 Detecting new activities

A protocol emulator taking advantage of ScriptGen FSM handles activities by traversing the protocol Finite State Machine according to the requests received from the client. When facing an activity that does not belong yet to the FSM knowledge, the emulator will experience a deviation at a certain FSM node: a given client request will not be matched by any transition.

A simple demonstration of the capability of ScriptGen to detect *different* activities can be done using any of the attack tools investigated in Section 3.5. We set up an emulator taking advantage of the FSM knowledge generated by the interaction of the MS06-040 exploit implemented in Metasploit 3.1 with a vulnerable Windows 2000 configuration. We already showed in Section 3.5.4 that such FSM is able to correctly handle future instances of the same exploit when disabling the bind call obfuscation technique implemented in Metasploit. In order to evaluate the ability of the emulator to recognize a *different* activity, we took advantage of the FSM to handle a different exploit on the same port. We chose the MS05-039 vulnerability (CVE 2005-1983), implemented in the Metasploit 3.1 module *windows/smb/ms05_039_pnp*. This vulnerability, famous for being exploited by the Zotob worm, involves an unchecked buffer in the Windows Plug and Play service that can be exploited through remote interaction on the Windows SMB protocol (ports 139/445).

When running the MS05-039 exploit against a FSM trained with samples of the MS06-040 exploit, the ScriptGen-based emulator correctly handled the first 5 client requests generated by the exploit tool. An in-depth inspection of the packets and of the Metasploit code revealed that these initial requests correspond to the SMB session establishment. Metasploit 3.x implements a simple SMB client that takes care of common tasks and that is shared by all the exploit tools involving the SMB protocol. The first part of the conversation is thus actually the result of the execution of a shared code and this explains the ability of the FSM to handle the

interaction. The 6th request generated by the MS05-039 exploit is not matched by any transition in the ScriptGen FSM. The request is a SMB NT_CREATE_ANDX request to access the pipe *\browser*. Interestingly enough, this request does not actually consist in a deviation from the behavior of the MS06-040 exploit: also the other requests access to the same pipe using an NT_CREATE_ANDX request. But while the MS05-039 implementation names the pipe "*\browser*", the MS06-040 implementation uses capital letters. The service implementation is case insensitive and thus has identical behavior in the two cases, while the FSM model of the protocol is sensitive to this difference and considers the change of case of the parameter a deviation.

We repeated the experiment modifying the case of the pipe name, and the request was handled correctly by the FSM emulator. A deviation from the FSM model was then detected at the 6th client request generated by the exploit. Such request corresponds to a DCE RPC bind request, and contains as UUID the value 8d9f4e40-a03d-11ce-8f69-08003e30051b, corresponding to the Plug and Play service targeted by the vulnerability. This is indeed a deviation from the behavior of the MS06-040 exploit, which binds to the UUID 4b324fc8-1670-01d3-1278-5a47bf6ee188, corresponding to the Windows Server service.

This exemplifies the ability of the ScriptGen FSMs to identify deviations in the protocol interaction associated with new attacks (such as 0-day) but also to variations of known attacks generated by different implementations or different configurations of the exploit tool. It is important to underline that ScriptGen does not blindly detect *any* variation in the client requests: thanks to the semantic inference of the region analysis algorithm, ScriptGen is able to identify variations that do not *normally* happen. Referring to the previous experiment, the MS06-040 exploit implementation chooses a different SMB Process ID at every instance of the attack. The FSM model *expects* that field to be mutating, and its variation is a *characteristic* of the protocol interaction. But all the MS06-040 exploits perform a DCE RPC bind to the same resource identifier, so when an attack is encountered using a different value for that field the attack is considered *different*.

4.2.2 Learning new activities

Once a new activity is recognized as such, the emulator is unable to carry on the conversation taking advantage of the existing FSM knowledge. Since the activity is unknown, it is of extreme interest to observe the continuation of such conversation. One possible solution would be to terminate the connection and redirect future activities generated by the same attacking source to a high interaction honeypot able to handle the activity. In the case in which the attacker tried again to repeat the same attack on the same target, the attack would be redirected to the high interaction honeypot and this would generate a new sample for the FSM refinement. In practice, we consider this event unlikely to happen in a short time: in the case of a worm propagating randomly over the whole IP space with a scanning frequency of 100 hosts per second, in average it would take 497 days for a honeypot IP to be

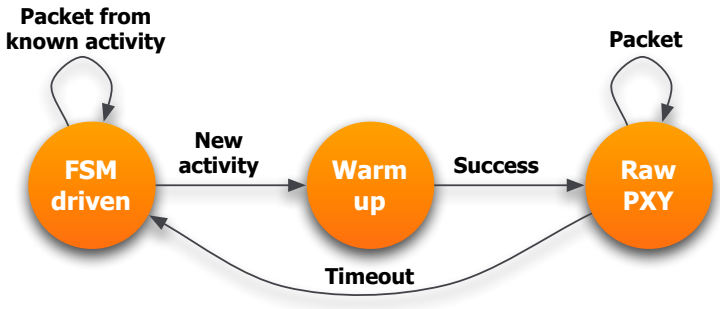

Figure 4.1: Proxying phases

hit a second time by the same attacker. While in many other propagation strategies such number could be significantly lower, this example is used as a worst case to motivate the usage of alternative techniques.

The proxying algorithm is inspired by an idea initially suggested by Cui in [Cui 2006c] and then independently expanded in the context of this work. The algorithm allows us to dynamically react to the detection of a new activity. Once a conversation within a TCP session is detected as being new with respect to the current FSM knowledge, we are able to rely on a high interaction honeypot acting as an *oracle* and providing the answers to the client requests that are out of the FSM knowledge. This transition is performed on the fly and in an almost completely transparent way from the point of view of the attacking client.

As illustrated in Figure 4.1, the operation of a protocol emulator implementing the proxying algorithm can be articulated over three different states.

- **FSM driven operation.** The emulator takes advantage of the FSM knowledge to handle the conversation with the attacker. While interacting with the attacking client, the emulator caches the ordered list of messages $[M_1, M_2, ...M_n]$ generated by the attacker.

- **Warm up phase.** Once a new activity is detected, the emulator is assigned a high interaction honeypot able to help in handling the activity. Initially, the emulator and the high interaction honeypot are not in sync: while we assume the high interaction honeypot to be in a "clean" state, the emulator has already interacted with the attacking client and this interaction has modified its internal state (for instance, one or more TCP connections are established between the attacker and the sensor). In the warm up phase, the sensor must then sync the honeypot state to its internal state. This is accomplished by *replaying* all the messages $[M_1, M_2, ...M_n]$ generated by the given attacker during the FSM driven operation.

- **Raw proxying operation.** During this phase, the emulator acts as a proxy between the attacker and the initialized high interaction host. All the messages

$[M_{n+1}, M_{n+2}, ...M_{n+k}]$ generated by the attacker in this phase are relayed by the emulator to the high interaction host, and all the corresponding replies $[R_{n+1}, R_{n+2}, ...R_{n+k}]$ generated by the host are sent back to the attacker by the emulator.

During the above interaction, the set of messages $[M_1, ...M_{n+k}]$ relayed to the high interaction host and the corresponding messages $[R_1, ..., R_{n+k}]$ generated as response constitute a new sample of interaction that can be used for the FSM learning. The combination of an emulator implementing the above proxying algorithm with a high interaction host allows the automated generation of new samples of interaction for unknown activities. These samples can be used to refine the existing FSM knowledge and generate a new FSM path as soon as the corresponding bucket is filled with enough samples. The proxying algorithm thus "closes the loop" between the refinement and the emulation phase, interleaving the two processes.

An important decision in the context of the proxying algorithm is the exact definition of a message. We have seen in the previous Section that ScriptGen FSMs model the protocol conversation at application level, which is focusing on application level payloads. The underlying TCP/IP interaction in the FSM driven operation can be easily handled using the standard TCP/IP stack of the operating system, greatly simplifying the complexity of the emulator. This would lead to the straightforward choice of following the same approach also in the proxying algorithm, defining a message as an application level payload and discarding the TCP/IP information.

Application level proxying has an important shortcoming: it does not preserve message boundaries. Working at application level only, there is no way to distinguish two separate requests 500 bytes long sent in fast succession from a single request 1000 bytes long. We do want to preserve this difference for two reasons.

Firstly, the warm up phase must reproduce on the vulnerable host exactly the same effects that the attack would have had without the intermediation of the protocol emulator. We identified a number of cases in which the loss of message boundaries in SMB exploits led to unexpected behaviors. Message boundaries, as well as TCP PUSH flags, need to be preserved in order to correctly replay the attack.

Secondly, the conversation sample generated by the proxying algorithm is used to refine the protocol FSM. In this context, ambiguities in message boundaries are not acceptable. Let's consider the following scenario. An exploit sends two separate packets 500 bytes long in rapid succession at a given point of the conversation. If no server answer is associated to the first 500 bytes payload, the boundary between the two requests is ambiguous and solely depends on timing issues. For instance, when the activity is unknown, proxying may introduce a small processing delay. As a result of this delay, the two separate requests could be buffered together and seen by the received as a single 1000 bytes message. This would lead to the generation of a single transition expecting a 1000 bytes message with characteristics given by the concatenation of the two messages. In different conditions

such as during FSM driven emulation, the received might be faster in reacting to new messages in its buffer. The two messages may then be received distinctly and would not match any more the previously generated transition. By preserving the whole packet encapsulation, we provide a simple way to correctly preserve message boundaries and avoid ambiguities.

The previous reasons led us to choose a different approach to handle the proxying phase, which allowed the preservation of packet boundaries and TCP/IP flags, such as the TCP PUSH flag. This is accomplished by working at IP level and considering as message M a TCP/IP packet comprising the original TCP and IP header. This allows maintaining the packet boundaries and all the TCP/IP flags, but introduces some complexity drawbacks in replaying a conversation at such low level. The proxying algorithm configures itself as a TCP session hijack, with all the complications that this implies.

While solutions such as TCP Opera [Hong 2005] are now part of the state of the art and successfully address the problem, at the time of building the system none of them were available yet. We thus developed an ad-hoc solution. Thanks to a set of design decisions performed in the development of the communication channel between the emulator and the high interaction host, we have been able to greatly simplify the problem of replaying the conversation by the emulator to the high interaction host. As it will be explained in Section 4.3.3, the communication channel allowing the interaction between the emulator and the high interaction host is based on a TCP-based protocol called Peiros. We can thus take advantage of the characteristics of these solutions to simplify the problem of protocol replay:

- Differently from the connection between the attacker and the emulator, the connection between the emulator and the high interaction host maintains packet ordering and has zero loss rate. In fact, the underlying TCP protocol used to tunnel the packets transparently handles any network issue that may arise on the path between the emulator and the high interaction host. This ensures that blindly replaying the packets exactly in the same order on the two systems will produce exactly the same effects.

- The emulator and the high interaction host are under our *partial* control. This is needed to ensure that the TCP/IP stack of the two hosts will follow exactly the same choices in advertising their state (window scale TCP option, ability to use of TCP SACKs, maximum advertised window, ability to handle TCP timestamps, TCP Maximum Segment Size, ...). While this is possible manipulating the configuration of the two hosts, nothing can be done to control their entropy, and thus, for instance, the choice of the initial sequence number ISN when establishing a TCP connection.

The above characteristics greatly simplify the problem of the initialization of the high interaction host. The lack of control on the entropy of the high interaction host still needs to be addressed. That is, the initial TCP sequence number ISN_{emu} chosen by the emulator when establishing a connection with the attacker will be

different from the initial sequence number ISN_{host} chosen by the emulator when replaying to it the same connection attempt. The emulator must exploit its position of *man-in-the-middle* to compute the difference $d = ISN_{emu} - ISN_{host}$ and use it to "convert" acknowledgement numbers in client requests sent to the high interaction host and sequence numbers in the corresponding answers. A similar approach is also applied to TCP timestamps when present.

Concluding, low level proxying allows the dynamic initialization of a high interaction host upon detection of a new kind of activity. Working at IP level maximizes the reliability of the replay and, thanks to a set of assumptions compatible with the specific scenario, does not impact significantly the complexity of the algorithm.

4.3 Collection of information on code injection attacks

Section 3.5 showed with lab-based experiments how ScriptGen is able to correctly infer the semantics of the interaction generated by several tools exploiting different vulnerabilities. Such validation was focusing solely on the network conversation leading to a vulnerable state and not on the network interaction following a successful exploitation. In the epsilon-gamma-pi-mu model introduced in Section 4.1, this corresponds only to the *epsilon* dimension, the exploit. The observed interaction already contains the payload *pi* and the control flow redirection *gamma*, but no information is known on their location in the protocol stream or on their behavior.

When dealing with a limited number of highly popular worms, we can identify a strong correlation between the observed exploit, the corresponding injected payload and the uploaded malware (the self-replicating worm itself). With the spread of worms such as Blaster [URL 8], there was a certain degree of correlation between the different attack dimensions. Thanks to this correlation, retrieving information about a subset of them was enough to characterize and uniquely identify the attack [Bailey 2005]. This situation is changing. Taking advantage of the many freely available tools such as Metasploit [URL 44, Ramirez-Silva 2007], even unexperienced users can easily generate shellcodes with personalized behavior and reuse existing exploit code. This allows them to generate new combinations along all the four dimensions, weakening the correlation between them. This motivates our need to observe and retrieve information about the full attack trace.

The network interaction following a successful code injection is a function of the content of the injected payload π. Such payload may force the victim to open a TCP port on an arbitrary port number and save the content pushed by the attacker on a file. Or it may force the victim to perform an FTP download and actively retrieve a malware file from an FTP repository. The emulation of this phase is extremely valuable: the ability to successfully carry on its emulation gives access to the malware μ. In most cases, code injection attacks are generated by self propagating malware aiming at generating a copy of itself on the victim host. In these cases the access to the uploaded malware sample is thus equivalent to the access to a copy of

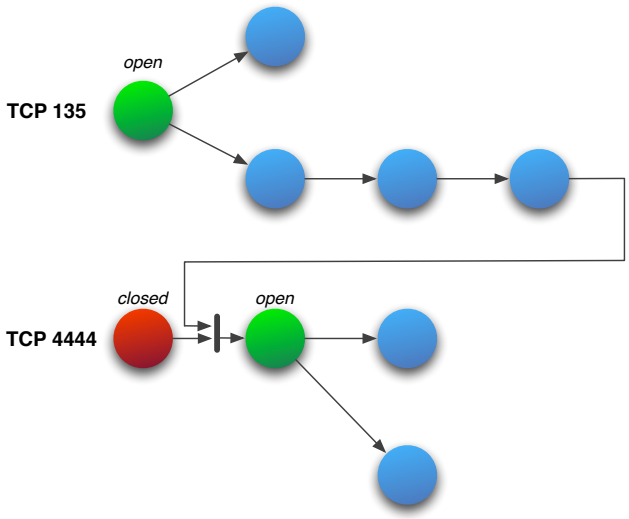

Figure 4.2: Inter-protocol dependencies

the attacker itself, and is a source of extremely valuable information on the nature of the attack.

4.3.1 Inter-protocol dependencies

We proposed an initial solution to address the problem in [Leita 2006]. Despite of the fact that, normally, the scope of a ScriptGen FSM is restricted to a single TCP session, we proposed to infer inter-protocol dependencies, mainly consisting in dependency links among states of different FSMs. Figure 4.2 shows an example of such dependencies: a successful exploit on port 135 opens a normally closed port (TCP port 4444). We proposed in [Leita 2006] a simple algorithm to infer dependencies through the analysis of the learning samples. The following cases were considered:

- Bind ports. A common action performed by a class of shellcodes consists in forcing the victim to open a normally closed port. This port can be used either to interact with a process, such as *cmd.exe* in Windows, or to directly push a malware file to the victim. When observing a successful connection on a previously closed port, we assume this state modification to be a consequence of the last client request sent by the attacker and we model the dependency accordingly.

- Protocol interleaving. Some protocols generate dependencies among two different streams. A classic example is the FTP protocol: commands on the control stream lead to downloads on the data stream. By looking at the interleaving of messages belonging to two data streams, it is possible to infer

dependencies that are common to all the samples, and that are thus likely to be a characteristic of the protocol.

This approach can be extended to other cases, corresponding to the possible actions commonly associated with the execution of the payload π.

The previously introduced solution is rendered impractical by the employment of polymorphism in the shellcode code. The idea of taking advantage of polymorphic techniques to hide the presence of a shellcode in a network stream was probably introduced at a Defcon 9 talk by a hacker named K2 in 2001 [URL 16]. The tool ADMmutate presented at the talk provided a simple API to transform a payload π_1 into a payload π_2 having the same effect when executed but with a totally different content. Two distinct executions of the mutation code bring to two totally different payloads, and this allows attackers to generate payloads that are extremely difficult to detect with classical intrusion detection systems such as Snort [Roesch 1999]. Experimental work such as [Polychronakis 2007] reported the extensive usage of such techniques by real world attacks.

The basic assumption required for inter-protocol dependencies to correctly model the shellcode execution is the presence of a single effect associated with each exploit state. Referring to the example of Figure 4.2, we can imagine that while the represented exploit state on port 135 leads a state modification on port 4444, a different state associated in the FSM will involve, for instance, port 5000. While such an approach would work correctly when facing non-polymorphic payloads, the usage of polymorphic shellcodes breaks the previous assumption. When employing polymorphism, two distinct attacks leading to the same action use two totally different payloads. ScriptGen would interpret such variation as a characteristic of the protocol, and would thus build a single transition with a mutating region matching all the polymorphic shellcodes embedded in a given exploit. It would then be impossible to associate a single characteristic action to the traversal of that path.

Our early experiments led us to conclude that network-centric perspective followed by inter-protocol dependencies does not cope with the complexity of a code injection attack.

4.3.2 Dissection of the epsilon-gamma-pi-mu model

Our experience with inter-protocol dependencies made us realize that the network-centric approach followed by ScriptGen, while suitable for the exploit emulation, was not suitable to handle the later stages of a code injection attack. We thus tried to dissect the attack trace into the various stages of the epsilon-gamma-pi-mu model, and handle each stage in the most suitable way [Leita 2008a]. We introduced three functional entities associated with the different phases of a code injection attack: sensor, sample factory, and shellcode handler. These three components are the main building blocks for the SGNET deployment.

The SGNET sensor is a small daemon to be deployed on a low-end host in

different Internet locations and in charge of handling the network interaction with the attacker. From the point of view of the attacker, the sensor handles the network interaction during the whole attack trace. Nonetheless the packets generated by the sensor can be generated by other entities and relayed to the sensor, which blindly injects the packets into the network acting as a proxy.

4.3.2.1 Epsilon: exploit phase

The exploit phase can be handled using the ScriptGen FSM knowledge if the activity is known. Traversing a FSM to handle an activity is an extremely inexpensive activity, and can be performed locally by the sensor. Sensors are thus completely autonomous in handling generic scanning activities and exploitation attempts as long as the activity falls within the FSM knowledge.

If an activity falls outside the FSM knowledge, the sensor needs to rely on an external oracle taking advantage of the proxying algorithm introduced in Section 4.2. In the context of SGNET, we call this entity the *sample factory*. As we have seen, the interaction of the sensor with the sample factory allows the generation of new samples for unknown activities that can be used to refine the FSM knowledge.

4.3.2.2 Gamma: the control flow hijack

The final objective of the exploit is to take advantage of a vulnerability to redirect the control flow of the victim towards an executable code (the shellcode π) injected by the attacker into the victim memory. In order to correctly understand the behavior of the shellcode, we need to detect when an attacker is attempting to redirect the control flow and we need to identify the target of this redirection within the protocol stream.

In order to accomplish the objective, we took advantage of memory tainting techniques in the sample factory. Argos [Portokalidis 2006] is a freely available tool that takes advantage of *qemu*, a fast x86 emulator [Bellard 2005], to implement memory tainting. Keeping track of the memory locations whose content derives from packets coming from the network, it is able to detect the moment in which this data is used in an *illegal* way. For instance, tainting allows to detect when network data is being pointed by the virtual processor Instruction Pointer, and is thus going to be executed by the victim. We reused the work of Portokalidis et al. in Argos in order to provide to the sensors not only the network interaction needed to emulate the exploit, but also to provide host-based information on the presence of code injections (γ).

The additional information provided by the Argos-based sample factories can be summarized into two points:

- Detection of a successful control flow hijack. This allows tagging all those states of the protocol FSMs that are the final stage of a successful exploit. Also, it allows the reliable detection of all the successful exploitations of

Figure 4.3: Shellcode identification

the high interaction hosts and the stop of their execution as soon as the attacker successfully takes control of them. This reduces the security concerns involved in giving access to attackers to vulnerable systems.

- Information on the position of the payload in the protocol stream. We have modified Argos in order to receive information on the position of the first byte B_i of the protocol stream that the victim tries to execute as a result of a successful control flow redirection.

In order to infer from the latter information the position and content of the payload *pi*, we take advantage of simple heuristics. Ideally, the payload π should be located at bytes $[B_i, .., B_i + k]$ following the first byte being executed by the victim (Figure 4.3 A). While this happened to be the case for a number of exploits in our experiments, we identified a number of cases in which the payload identified by this heuristic was made of only a few bytes. These bytes turned out to be composed of only NOP instructions and a jump instruction to a different memory location. We then realized that the structure of these payloads was similar to that represented in Figure 4.3 B.

Algorithm 4 get_payload(*stream*,*i*)

```
1: l ⇐ len(stream)
2: π ⇐ stream [ i,l ]
3: while not(valid(π)) and i > 0 do
4:     i ⇐ i − 1
5:     π ⇐ stream [ i,l ]
```

We revised the previous heuristic as shown in Algorithm 4. Assuming to have a method to check the validity of a given payload π (discussed in the next Subsection), we backtrack from the initial choice until a valid payload is identified. This very simple heuristic proved to be sufficient to handle the majority of payloads involved in code injection attacks.

We take advantage of the information on the payload location to tag the FSM transitions during the incremental refinement, specifying the position of the shellcode in the corresponding client request. When the final state is reached, the shellcode is reconstructed as concatenation of all the segments (see Figure 4.4).

The proposed heuristic implicitly assumes that the mapping between the protocol stream and the victim's memory is "linear". That is, we assume that subsequent bytes of the shellcode in memory will also be subsequent in the protocol stream.

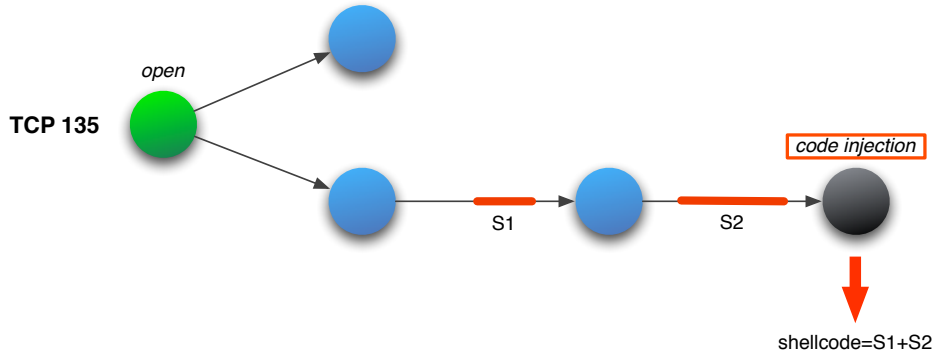

Figure 4.4: Shellcode reconstruction

While we saw that this assumption holds for most of the observed exploits, we also identified cases that break this assumption.

- **Unicode encoded strings.** Some protocols such as SMB encode the string fields according to the unicode UTF-16 encoding. When running the LSASS exploit [URL 20] as implemented in [URL 22] and targeting Windows 2000 systems, we saw that the argument of the DsRoleUpgradeDownlevelServer call, used to overflow the victim's buffer and push the payload π, is encoded in such a way. Each byte of the shellcode is encoded in the protocol stream over 16 bits, where the most significant byte is set to 0. The vulnerable service will decode such unicode string and load the result in memory. The payload executed in memory by the victim is thus different from that extracted from the protocol stream.

- **Protocol headers.** In some cases, the payload π spans not only over multiple TCP packets, but also over multiple application protocol payloads. For instance, in the previous LSASS exploit, the shellcode spans over two DCE/RPC fragments F_1 and F_2, each of which is prepended by an SMB header H_1 and H_2 in the application level payload. When taking advantage of the previous heuristics to reconstruct the shellcode, we have no way to recognize the SMB header H_2 from the second fragment of the payload F_2. The generated shellcode will thus be polluted by the presence of such header: $\pi = F_1 + H_2 + F_2$.

In both these cases, the shellcode sample identified by the previous heuristic will be corrupted. Our practical experience with the emulation of these samples taking advantage of the techniques proposed in the next Section showed that this corruption did not prevent a correct emulation. Heuristics were in fact already present in the shellcode emulator to detect Unicode encodings and correctly repair them. Also, the usage of regular expressions to recognize the shellcode behavior allowed being resistant to the pollution of the sample generated by additional protocol headers. The proposed heuristic proved thus to be sufficient at the col-

lection of shellcodes of a sufficient quality to allow their emulation despite of the introduced anomalies.

4.3.2.3 Pi: the shellcode execution

In the case of a successful code injection attack, the sensor relies on a *shellcode handler* for the emulation of the payload *pi*. The shellcode handler is thus an entity that, given as input a payload and the network characteristics of the sensor, produces the required network interaction to emulate the payload execution. Similarly to the sample factory, the shellcode handler behaves as an *oracle* to emulate an unknown network interaction in proxying mode. But differently from the sample factory, the interaction generated by the shellcode handler is *not* used as a sample for the FSM learning.

Different approaches exist in the literature to recognize the presence of a shellcode within a protocol stream. Some approaches aim at recognizing peculiar characteristics of the payload: for instance, detecting the presence of *sledges* before the executable payload [Toth 2002, Akritidis 2005]. Some aim at detecting the presence of executable code by checking the correctness of its control or data flow [Chinchani 2005, Wang 2006]. Others aim at detecting decryption routines for polymorphic shellcodes emulating their execution [Polychronakis 2006, Polychronakis 2007].

Despite the considerable attention to the problem of the detection of executable code in the network flows, little has been done to address the problem of its emulation. Polychronakis et al. in [Polychronakis 2006, Polychronakis 2007] detect the presence of shellcode in the network streams taking advantage of a CPU emulator. While the objective of the method is to detect the shellcode, they achieve this objective through the execution of the decryption procedure of the polymorphic code, which unpacks the real shellcode payload. The authors do not attempt to execute the payload: the payload is in fact likely to contain system calls (for instance, socket operations) that would require a full virtualization environment to be executed. Another relatively recent open-source project is now trying to accomplish shellcode profiling using a CPU emulator [URL 3]. While these solutions may prove to be effective and useful in the future, at the time of the implementation of our system they were not yet mature or available.

We thus followed a different direction and took advantage of the shellcode engine used by Nepenthes [Baecher 2006]. The Nepenthes architecture is detailed in Section 2.1.2.2, page 8. We modified Nepenthes in order to bypass its vulnerability modules and directly provide shellcode samples to its shellcode engine. By doing so, we effectively bypass the most important drawback of Nepenthes, namely its limited number of handcrafted vulnerability modules, and take full advantage of its shellcode emulator and download modules. We built a shellcode handler able to recognize valid shellcode and emulate its behavior. Still, its usage has shortcomings with respect to full shellcode emulation. Nepenthes' approach is knowledge-based also in the shellcode emulation: it employs a set of signatures

(mainly, regular expressions) to recognize a set of known decryption routines and unpack simple polymorphic shellcodes. Then, it takes advantage of a set of heuristics to reconstruct from the unpacked payload a URL corresponding to the action to be taken to download the malware sample. SGNET can thus potentially identify payloads that the shellcode handler is unable to emulate.

4.3.2.4 Mu: the malware behavior

The shellcode handler allows us to understand the behavior of a shellcode by mainly converting the payload π into a generic URI representing the download action that is required to download malware. The shellcode handler is then able to emulate this behavior taking advantage of Nepenthes download modules, and ultimately is able to collect malware samples. The malware μ is a valuable source of information on the root cause of the observed activities. In self-propagating malware such as worms, the downladed malware corresponds to an exact copy of the attacker. This might not be the case in more sophisticated malware such as bots: a botnet may potentially push to the victim a malware that is different from itself. Still, we can take advantage of the malware to retrieve rich information regarding the goals of the attacking client.

The characterization and analysis of malware samples is an extremely active research area. In order to have information on the collected sample, we can identify two main areas of interest:

- **AV signatures.** The antivirus (AV) community is very active in the process of collecting malware samples and classifying them according to a set of signatures. It is thus of great interest when receiving a new malware sample to retrieve information from the various AV products on their ability to recognize the sample, and on the name associated with it by the vendors when available.

- **Behavioral analysis.** The entirety of the malware collected so far by SGNET consists of binary executable code for the Windows operating system. It is thus of interest to retrieve information about the actions performed by the malware once executed on a real system. A number of existing sandbox implementations exist that perform this task: Anubis [Bayer 2005], CWSandbox [URL 9] and Norman Sandbox are among the best known ones.

- **Structural information and packers.** The samples Windows executables and are thus structured according to the Portable Executable (PE) format. The PE information provides all the information required to the OS loader to manage the executable code. This information can be easily retrieved from the malware, and it is possible to use public signature databases [URL 28] to associate such structure to the output of well known packing utilities. These packing utilities are in fact often used by attackers to make the malware detection more challenging for AV products, to prevent the use of debuggers,

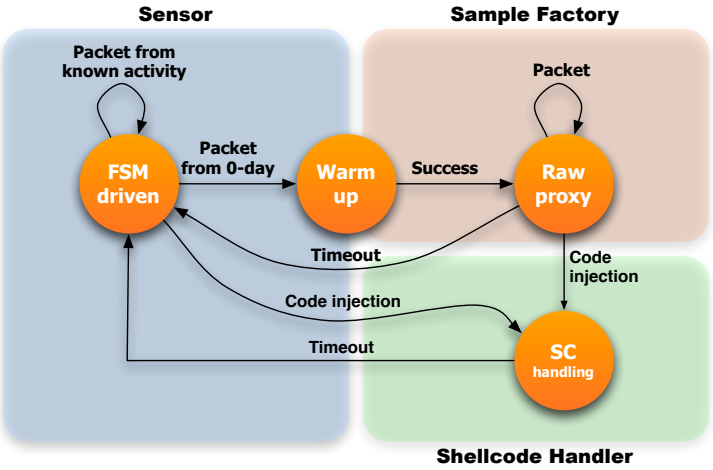

Figure 4.5: SGNET phases

and often to prevent execution in sandbox environments. Information on the packer used by a given malware can thus be very valuable in understanding the level of sophistication of the sample.

The above information can be retrieved relying on existing tools and techniques developed by other research teams. Most of the information on the collected malware samples is thus retrieved through information sharing with other projects. A detailed overview on the information enrichment performed over the data collected by SGNET is postponed to Section 5.1.

4.3.2.5 Interaction

Summarizing, the interaction with an attacker during the emulation of a code injection attack evolves through different states represented in Figure 4.5. As it is clear from the diagram, SGNET extends the proxying phases introduced in Figure 4.1 on page 67 with an additional phase, which exploits the shellcode handler to emulate the collected shellcodes.

4.3.3 The architecture

The resulting architecture of the SGNET deployment is illustrated in Figure 4.6. The sensors are deployed along the IP space, and due to the small resource requirements of the ScriptGen approach they can be easily deployed on low-end machines. The sample factories are deployed in a central farm on one or more high-end hosts, together with the shellcode handlers. It is very important to understand that this solution substantially differs from the classical honeyfarms where the farm is invoked for each and every attack not implemented by, for instance, a honeyd script. In SGNET the high interaction nodes (Argos machines) are only used when

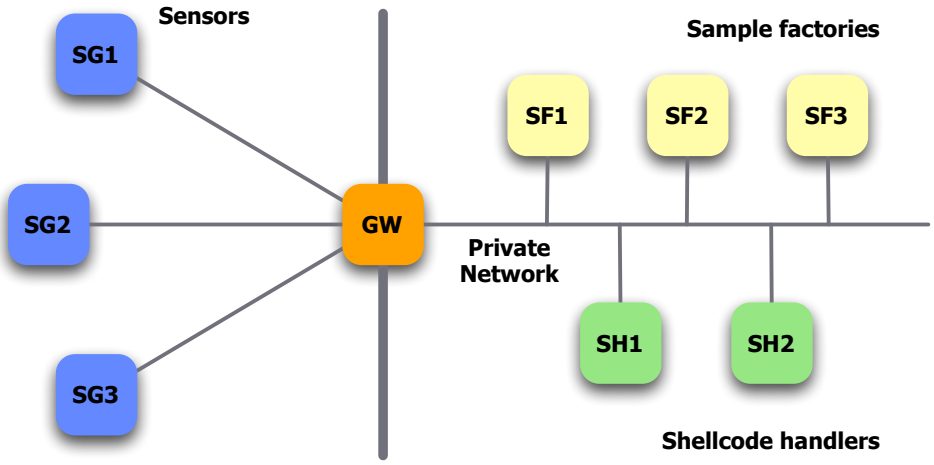

Figure 4.6: SGNET architecture

new attacks are observed. Upon collection of enough samples of attack activity the ScriptGen FSMs will be refined and pushed to all the sensors, and the activity will be handled locally. As we will see in Section 4.4.2 , this has a significant impact on the dimensioning of the farm.

The different SGNET entities are coordinated through an ad-hoc HTTP-like protocol named Peiros. The goals of this protocol are threefold.

Firstly, it must provide primitives to the sensors to send service requests to the other entities, asking for the instantiation of an oracle or submitting a payload to a shellcode handler.

Secondly, it must allow the exploitation of the distributed deployment of the sensors in order to maximize the statistical variability of the collected samples. That is, it must allow sharing the collected samples in order to perform a centralized refinement. Also, it must allow the coordination of the distribution of the refined FSMs in order to make sure that all the deployed sensors share the same knowledge of the protocols.

Finally, it must allow the tunnelling of packets between the various entities, avoiding the need to modify packets endpoint addresses or modify the routers configuration in order to deliver tunneled packets to their destination. Delivering tunneled packets as application payloads over a normal TCP stream allows great flexibility and transparency in the placement of the various entities.

The interested reader can find in Appendix A an in-depth specification of the Peiros protocol.

An additional component is introduced in the architecture, which is the *SGNET gateway*. The gateway is the core of the whole SGNET architecture. The gateway is the coordinator for every SGNET sensor. Each sensor on startup takes advantage of the Peiros protocol to connect to the gateway, and retrieve its own configuration. Centralizing the configuration details greatly simplifies the administrative tasks.

Also, the gateway acts as an application level proxy for the Peiros protocol. SGNET sensors send service requests to the gateway, which transparently dispatches them to a free sample factory or shellcode handler. The gateway acts then as a load balancer for the various SGNET entities, using a simple round robin scheduling policy. Finally, the gateway centralizes the collection of samples generated by the various sensors and refines the ScriptGen FSMs. As soon as a new refinement is produced, the gateway is able to push the update to all the sensors, taking advantage of the Peiros protocol. All the sensors active at a given moment will thus have the same protocol knowledge, with some approximation due to network latency and retransmissions. It is important to understand that the architecture shown in Figure 4.6 is an abstraction of the composition of the different entities. The Peiros protocol can be easily extended to allow more complex configurations, for instance with multiple gateways to improve the system availability and scalability.

4.3.4 The SGNET deployment

In order to validate the proposed architecture and study the behavior of ScriptGen when facing real attacks we have developed and deployed a prototype, which has been running and collecting data over a period of 8 months.

The SGNET deployment is open to any institution interested in joining the project. The spirit of the deployment is very similar to that of the Leurre.com project (see Section 2.2.2.1 on page 18). Any industrial or research institution interested in accessing the information collected by the deployment is welcome to become a partner. In order to become a partner, the institution needs to deploy an SGNET sensor and agree to sign a Non-Disclosure Agreement protecting the information involving the sources and the targets of the attacks.

At the time of writing, the deployment consists of 22 sensors, beta testers, deployed in different locations of Europe, America, Asia and Australia.

All the sensors in the deployment share the same network setup: they are assigned 4 IPs, one of which is used for the remote control of the honeypot and the Peiros communication of the sensor daemon with the central entities. The other three IPs are emulated by the sensor and used to collect information on the attacking clients. In the context of this work, we have decided to assign to all the IPs a single profile, associated with a well defined setup in the sample factory. This profile corresponds to an unpatched Windows 2000 Professional OS, with IIS 5.0 installed. While more various configurations are possible and will probably be used in the future, we preferred here to concentrate on the learning of a single profile assigning to it all the IP addresses currently available.

The 22 sensors provide thus a total of 66 honeypot IPs that are deployed over 18 different class A networks over the Internet. Each of these IPs contributed to the collection of data in diverse proportions as visible from Table 4.1. Table 4.1 represents the daily rate of successful connection attempts handled by the honeypots. The majority of the interactions observed by the deployment and analyzed within this work involved typical Windows TCP ports, mainly ports 139

Env	Port 139	Port 445	Port 135	Port 80	Others	Total
25	656.9	151.0	430.9	18.0	39.9	1296.7
15	411.4	431.6	335.8	17.9	75.5	1272.1
5	821.9	223.5	113.9	26.9	44.6	1230.8
17	719.8	241.6	50.9	6.9	18.1	1037.3
26	575.3	28.4	351.2	3.8	26.5	985.2
19	673.0	74.4	42.8	162.0	14.5	966.7
9	174.8	93.7	29.4	11.8	11.8	321.5
16	102.8	105.0	37.1	1.0	6.5	252.4
10	27.6	135.0	22.7	28.0	20.3	233.6
2	137.1	22.8	1.5	11.7	48.0	221.1
6	38.4	40.6	27.7	20.6	6.7	133.9
14	6.0	3.7	32.8	50.4	21.0	113.9
12	40.9	34.0	0.5	1.4	3.7	80.5
22	0.0	0.0	0.0	6.5	72.5	79.0
11	0.0	0.2	0.0	60.6	9.5	70.3
8	38.8	8.4	6.8	4.8	4.5	63.3
20	6.3	9.8	0.1	10.0	7.4	33.7
23	0.0	0.0	0.0	23.5	0.9	24.4
3	0.0	0.0	0.0	17.8	5.8	23.6
18	0.0	0.0	0.0	4.0	6.1	10.2
7	0.0	0.0	0.2	2.9	3.0	6.2
13	0.0	0.0	0.0	0.2	0.2	0.4
Total	4431.2	1603.7	1484.4	490.6	447.0	

Table 4.1: Daily rate of TCP sessions hitting each sensor

TCP, 445 TCP and 135 TCP. It is possible to see from Table 4.1 that the different sensors part of the SGNET deployment are characterized by very different volumes of inbound traffic. Some sensors, such as sensor 22, are not hit at all by any activity involving the Windows ports: a possible explanation for this behavior can be found in the presence of packet filters in front of the honeypots, filters out of the control of the hosting organization. Other differences cannot be instead be explained other than with differences in the distribution of the activities over the Internet IP space. For instance, sensor 19 and sensor 25 are hit by approximately the same volume of traffic on port 445, but differ of one order of magnitude in the volume of traffic observed on port 80.

The access to the SGNET dataset can be performed in different ways. Among them, it is worth mentioning a Python-based programming API that allows easy access to all the information without requiring in depth-knowledge of the underlying SQL schema. All the results presented in this work were obtained through this programming interface. The interested reader can find in Appendix B a documentation of the underlying ideas and concepts.

4.4 SGNET behavior

The deployment of the SGNET infrastructure represents the first deployment of ScriptGen based honeypots on the Internet. We have evaluated the learning and emulation capabilities of ScriptGen when facing a short list of exploits, but many questions have been left unanswered until now.

- **Does ScriptGen improve scalability?** In theory, the proxying algorithm introduced in this Chapter relies on high interaction solutions only to face new activities. We do not know if such an approach will be effective, which is if new activities will be learned fast enough to reduce the load on the sample factories.

- **What kind of activities does ScriptGen successfully learn?** We expect the observed activities to be very diverse in their characteristics. Large scale high breadth phenomena are likely to be paired with short duration, localized ones. While many samples of activity can be collected in the first case, the latter case may cease before having given the time to ScriptGen to learn the activity.

- **How will the FSM size evolve?** While we have seen that ScriptGen is effective in learning a single exploit configuration, we have no quantitative information on the variety of the exploitation and scanning techniques practically employed by Internet attackers. We thus have no *a priori* information on the evolution of the size of the ScriptGen FSMs and thus on the feasibility of using this technique in practice.

The information collected by the SGNET deployment in almost 8 months of observation can provide answers to these questions.

In the following pages, Figures 4.7, 4.11 and 4.13 provide a high level overview on the behavior of the SGNET deployment from three different perspectives. Respectively, they provide an overview on the evolution of the size of the FSM knowledge, on its ability to offload the sample factories and on its effectiveness in emulating code injection attacks and collect malware. In the plots, the vertical dashed lines correspond to the installation of new SGNET sensors.

4.4.1 Evolution of the FSM size

Figure 4.7 shows the growth of distinct FSM traversals in all the ScriptGen FSMs. The number of distinct traversals in the ScriptGen FSMs increases approximately linearly with time, with a relatively low growth rate. In the 8 months of observation, only 438 distinct FSM traversals were generated. These traversals mainly target port 445 (32%), port 135 (23%), port 139 (21%) and port 80 (13%). In Figure 4.7, it is possible to see that the creation of new traversals goes in parallel with the *death* of a number of them.

We define the death of an activity as the last date in which the activity was observed. Such a definition generates an ambiguity deriving from the limited nature of the observation interval T. Modelling an activity A as a periodic process, we can define its period π_A as the average number of days spacing two subsequent instances of such activity. Upon observation of an event $E_A1 \in T$ its death can be assessed only through the analysis of the events in the successive π_A days. Otherwise, it is impossible to discriminate the death of the event from the case in which event E_A2 following E_A1 does not belong to T. This ambiguity leads to a sharp increase in the number of deaths when approaching the end of the interval T. In order to cope with this problem, we defined the death of the traversals over an interval T' prolonged of one month with respect to the observation interval considered in this work. This has reduced the ambiguity for all the activities A having a $\pi_A < 30$.

We can identify three main reasons for the death of a traversal: deprecation, high specificity, and low activity frequency.

With *deprecation* we refer to the phenomenon for which the creation of a new FSM path "deprecates" an existing one. We have seen in Section 3.4.1 on page 50 that the choice of the future state among all the outgoing transitions of the current state depends on a matching function that assigns a score to each transition matching the incoming message. In special conditions, a newly generated transition T_n may obtain a higher score than a previous transition T_o for the incoming requests that were previously traversing T_o. In order to investigate the deprecation phenomenon, we searched for all the traversals generated in the period immediately successive to the death of a traversal on the same port. We detected 12 such cases, 5 of which were generated by the creation of a single path on port 80. A manual inspection of this path disclosed the following set of regions associated with the first transition of the conversation:

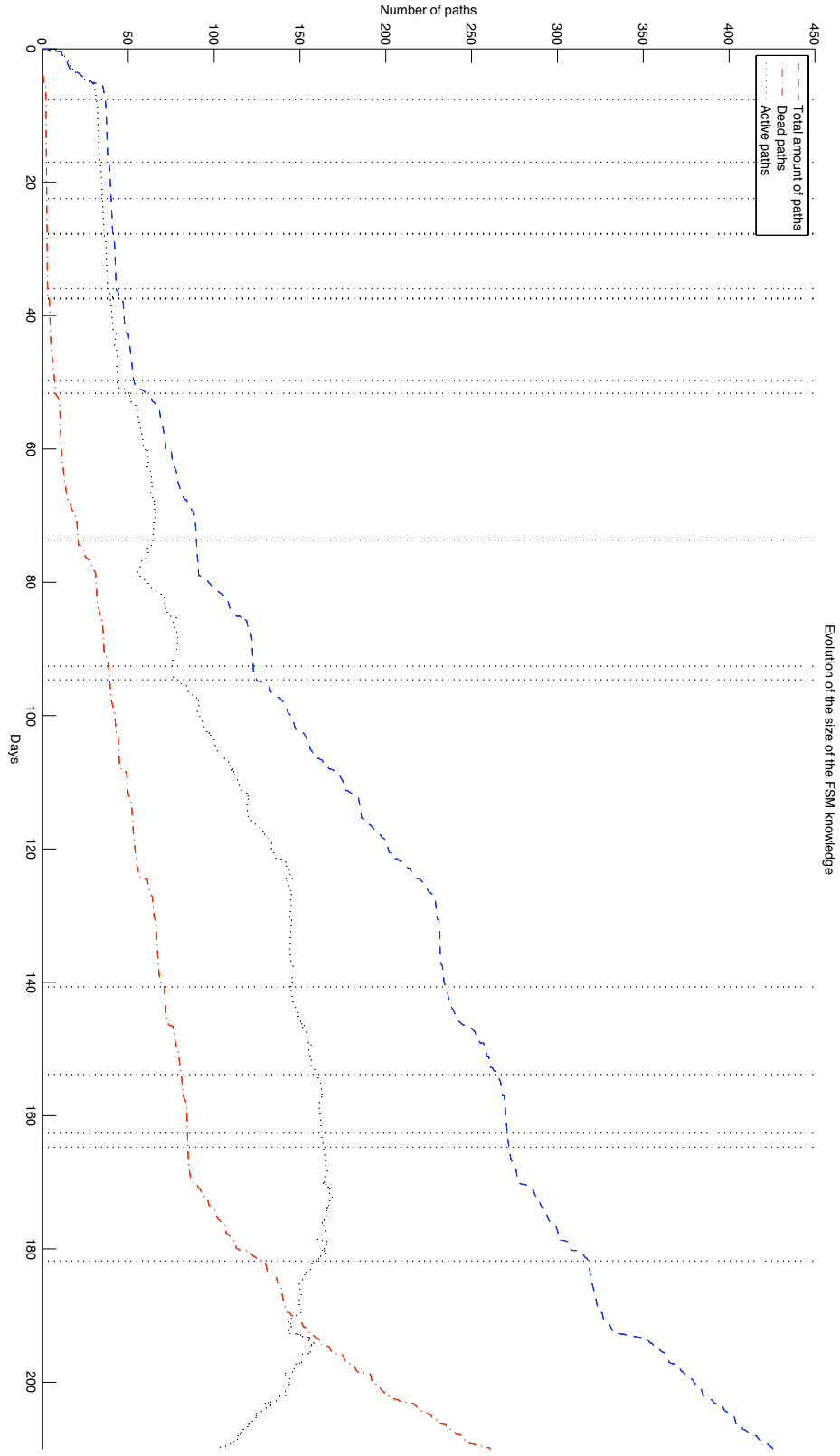

Figure 4.7: Evolution of the FSM knowledge size

$$M(1) + F(\text{“}E\text{”}) + M(0:7) + F(\text{“}T\text{”}) + M(0:12) + F(\text{“ ”}) + M(0:17) + ...$$

Such a path was generated by the incorrect grouping of a set of HTTP HEAD requests with two malformed HTTP GET requests, simply containing the payload `'GET '`. In normal conditions, these two types of payloads should have been separated by the semantic clustering. The HTTP HEAD requests in fact lead to an error response from the server, while the malformed GET requests lead to an abrupt connection termination from the server. A temporary problem in the experimental setup led to the accidental grouping of these two requests. The *pollution* of the sample set led to incorrect inferences resulting from the alignment of these two payloads:

```
'GE______T_____________ '
  |       |              |
'HEAD / HTTP/1.0..Host: X.Y.Z.K....'
```

The generated transition not only matches HTTP HEAD requests, but also any HTTP GET request. 5 existing traversals were deprecated by the generation of this path. Since the answer provided to the client when following this new traversal is inconsistent with the content that should be provided to an HTTP GET, the generation of this path diminished the ability of SGNET to observe activities. This shows the danger inherent in the effect of deprecation, but it also shows that deprecation can be detected by looking at abrupt changes in path trends.

A second cause for the death of a traversal is its high specificity. When collecting samples to be used for the refinement, no constraint is posed on its *source*. A traversal can thus be generated as a consequence of the repetition of the same attack launched against, or from, a single IP address. The corresponding traversal is thus likely to contain information specific to the victim and/or the attacker. Looking at the top part of Figure 4.8, we can see the Cumulative Distribution Functions on the number of attackers and victims traversing each FSM traversal. Approximately 25% of the traversals are generated by a single attacker, and 22% involve a single victim IP. In fact, 89 traversals (20%) are associated with a single attacker and a single victim IP. 77 of them had a duration of a single day, and died the same day of their birth.

The problem of highly specific paths can easily be fixed by defining constraints on the acceptance of samples in a bucket. For instance, we could prevent the acceptance of more than k samples generated by the same attacking IP. We consciously decided not to do that. The intuition underlying this decision is that if, for instance, a single attacker contributes in an anomalous way to the generation of samples, then it's a good idea to assign to it a separate traversal identifier and consider it as *different* from others. The absence of constraints on the diversity of the samples entering in the buckets allows us to easily pinpoint highly localized activities.

Finally, a last contributor to the death of traversals is the bias introduced by bursty activities. Figure 4.9 provides a measure of the duration of each traversal.

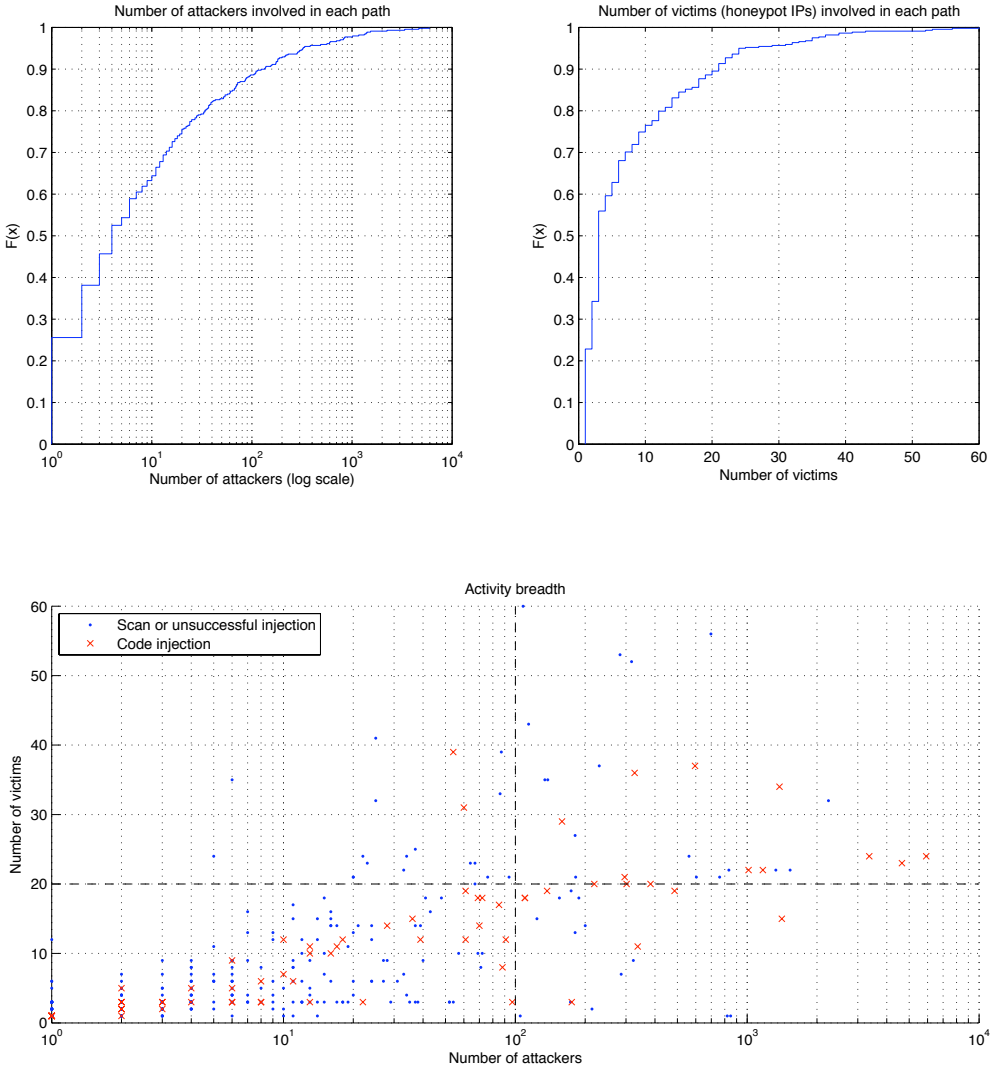

Figure 4.8: Dimension of the activities in terms of attackers and victims

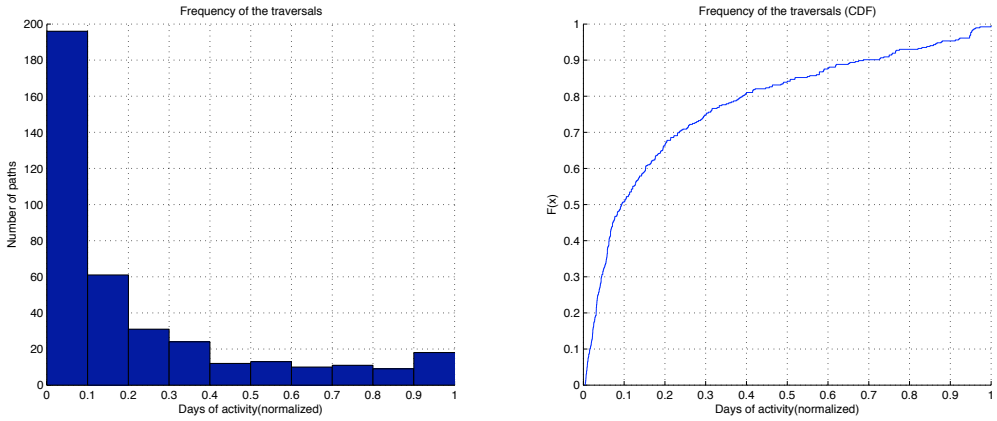

Figure 4.9: Average duration of a FSM path

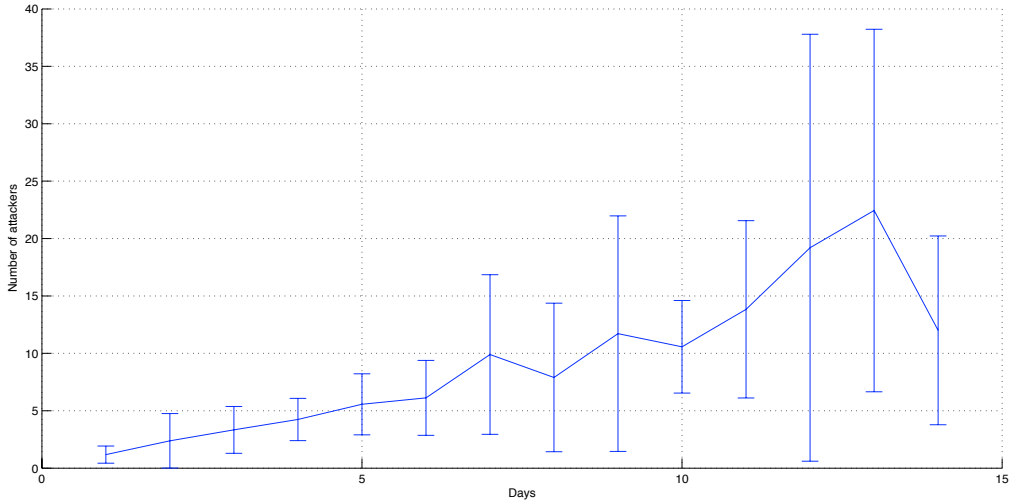

Figure 4.10: Relationship between duration and number of attackers (rare activities)

For every traversal, we have computed the ratio between the number of days in which the traversal was hit and the total number of days in which the traversal was present in the FSM knowledge. This measure is meant to discriminate between rare activities observed a limited number of days in their total lifetime from activities that appear daily in a constant fashion. Figure 4.9 shows the histogram and the cumulative distribution function for such parameter. 50% of the traversals have been observed only in less than 10% of their lifetime. Half of the traversals can thus be attributed to very bursty or rare activities rather than regularly scanning ones. The burstiness of these traversals is evident when looking at the number of days in absolute terms. 159 paths (36%) were traversed during only one day; 72 (16%) were traversed during 2 days; 44 (10%) were traversed during 3 days. As shown in Figure 4.10, these bursty activities are also associated to a small number of attacking sources: activities lasting less than 5 days are most of the times associated to less than 5 attacking clients.

Despite the death of a considerable number of traversals in the 8 months of observations for the reasons previously explained, an interesting fact comes to our attention. The lower part of Figure 4.8 shows an overview on the nature of the 438 paths generated in the 8 months of observation. Plotting these traversals according to the number of attackers and victims involved in each activity, we can have a visual picture of the dimensions of the phenomenon involved in each path. Dividing the area of the graph in 4 sub-areas, the bottom-left part includes the vast majority of generated paths. Out of 438 traversals, 48 traversals corresponding to successful code injections and 320 traversals corresponding to other activities involve less than 100 attackers and less than 20 target IPs (i.e. less than 7 honeypot sensors). 83% of the FSM knowledge is thus used to handle highly localized and *small sized* activities. In the 8 months observation period, this knowledge was used to handle only 41,018 traversals, which is only 6% of the total amount of traversals observed in SGNET. Focusing on the top-right area of the graph, 27 traversals (6% of the FSM knowledge) handled a total of 501,734 attack instances that is 77% of the total amount of attacks observed by SGNET.

This shows a an interesting fact on SGNET but also reflects an important characteristic of the scenario of malicious activities observable on the Internet. We can identify a small set of highly visible activities, involving a high number of attackers, and probably randomly scanning the whole IP space. These activities constitute by far the majority of malicious activities: according to our observations, 77% of the traffic observed by our honeypots is associated with just a few FSM traversals. The continuous growth of the FSM knowledge in the 8 months under analysis is instead associated with small bursts of activity, of short duration, and involving small numbers of attackers and victims. We don't expect the total number of created FSM traversals to stop increasing. As Figure 4.7 suggests, the steady state is likely to correspond to the continuous creation of new paths and the contemporary death of other paths involving short term localized activities that vanish shortly after having appeared.

These numbers ensure our method will continue to be feasible in the future

Period (days)	avg daily attackers	avg daily attackers (proxied)
1-60	82.41	31.03
61-150	166.40	39.76
151-210	171.66	25.51

Table 4.2: Average number of attackers handled by the deployment

regardless of the growth in terms of absolute number of paths. Given the ephemeral nature of the activities we are observing, the total number of active traversals in the FSM knowledge tends to stabilize and is far from raising scalability concerns (the maximum number of active traversals, generated by the overlap of the processes of birth and death, is 169 over the 8 months of observation). Finally, ScriptGen proved its ability to correctly learn not only high visibility activities, but also the bursty localized activities that consist of the bottom left area of Figure 4.8. This property combined with the capabilities of the SGNET distributed deployment gives the SGNET dataset a potentially very interesting insight on Internet threats.

4.4.2 FSM learning and sample factories

Figure 4.11 gives an overview of the ability of the FSM knowledge to offload the sample factories from handling previously encountered activities. The plot compares the daily number of attacking sources that were handled solely by the FSM knowledge with that of those attacking sources that exhibited a new behavior and thus required the intervention of a sample factory.

Table 4.2 numerically describes what can be already seen graphically from Figure 4.11. The gradual increase in the number of honeypots handled by the system increases the amount of traffic that is handled by the deployment: in the first 60 days of the observation period, the deployment observes an average of 82.41 attackers per day. In the last 60 days of the observation period, the number of average daily attackers grows to 171.66 per day. Despite of the fact that the average number of attackers is almost doubled in the last period, the load on the sample factories tends to decrease.

The reduction of load on the sample factories despite the global increase of honeypots and the corresponding increase in number of observed attackers shows that SGNET is able to scale. That is, the knowledge acquired by SGNET with the initial set of sensors is successfully reused to handle similar activities hitting newly installed sensors. This analysis shows that the learning is successful in preventing future usage of the sample factory for the same type for activities, but does not provide yet any information on the *quality* of the protocol emulation, that will be addressed in the next Section.

While SGNET is effective in reusing existing traversals to handle activities common to multiple platforms, we have seen that most of the traversals in the FSM knowledge handle very localized activities. On this basis, we would expect the

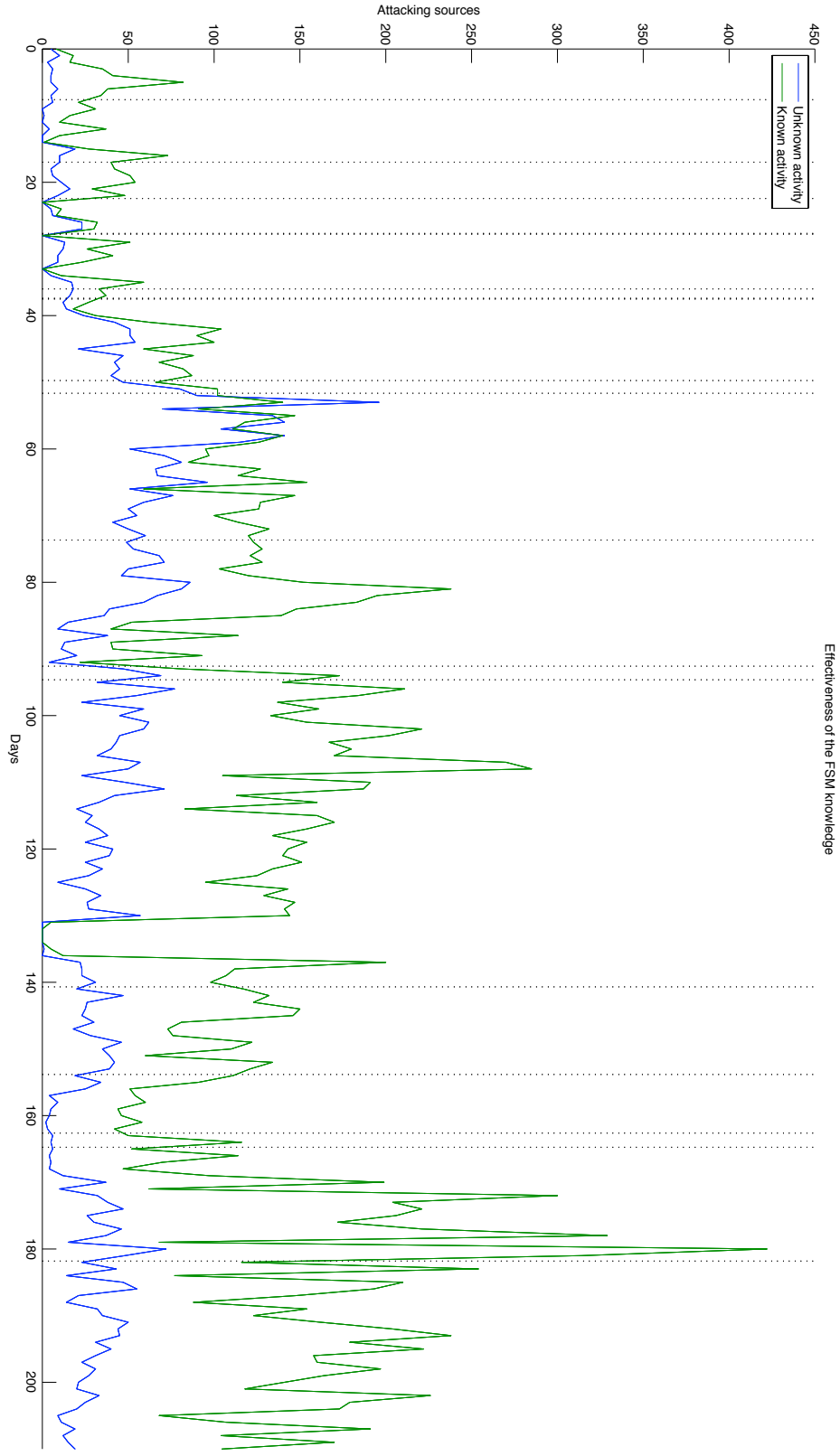

Figure 4.11: Ability of the FSM knowledge to offload the sample factories

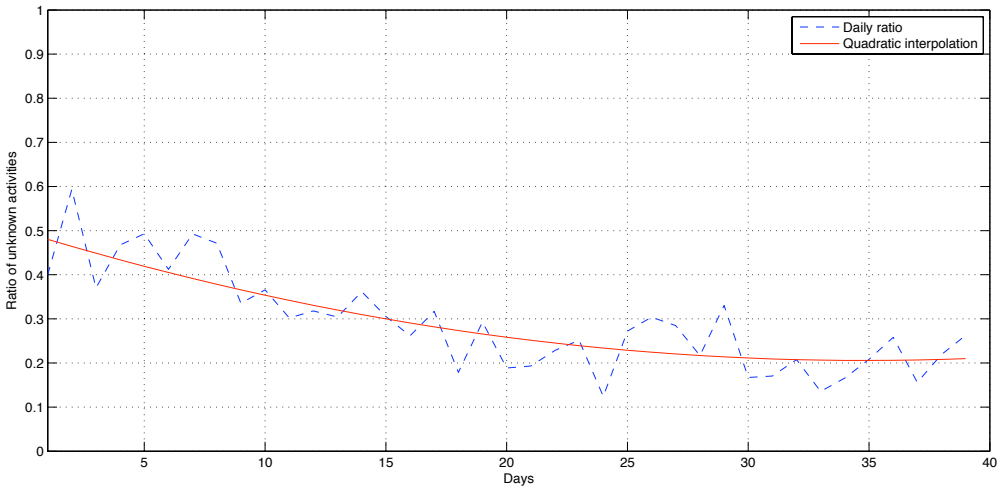

Figure 4.12: Effect of the installation of new sensors

installation of a new platform to have an impact on the load of the sample factory. In fact, installing a new platform should allow the observation of a considerable amount of localized activities hitting a network location on which SGNET had previously no knowledge. In order to verify this intuition, we have aligned the information on each installed platform according to the installation date in Figure 4.12. Day 0 on the X axis corresponds to the installation date of each platform, and the graph shows the resulting cumulative ratio of unknown activities in the first 40 days of lifetime. In the first 20 days of lifetime of a newly installed sensor, the sensor affects the sample factories according to the previously exposed intuition. After this period, the load seems to converge to a stable value.

4.4.3 Ability to emulate code injections

We have investigated until now the ability of SGNET to scale to multiple honeypots by taking advantage of the shared FSM model. The scalability investigation carried on so far needs to be complemented by an evaluation of the correctness of the generated traversals. A single traversal matching any inbound client request would be scalable, but would be unable to carry on correct conversations with the attacking clients.

In order to evaluate the ability of the SGNET infrastructure to handle code injection attacks, we focus here on the evolution of the size of the malware collection generated by SGNET. The malware download is the last stage of a code injection attack: a success of this stage is thus considered a reliable indication of the correct emulation of the whole attack trace. Figure 4.13 shows the growth of the number of samples collected by the SGNET infrastructure, both in absolute terms and in

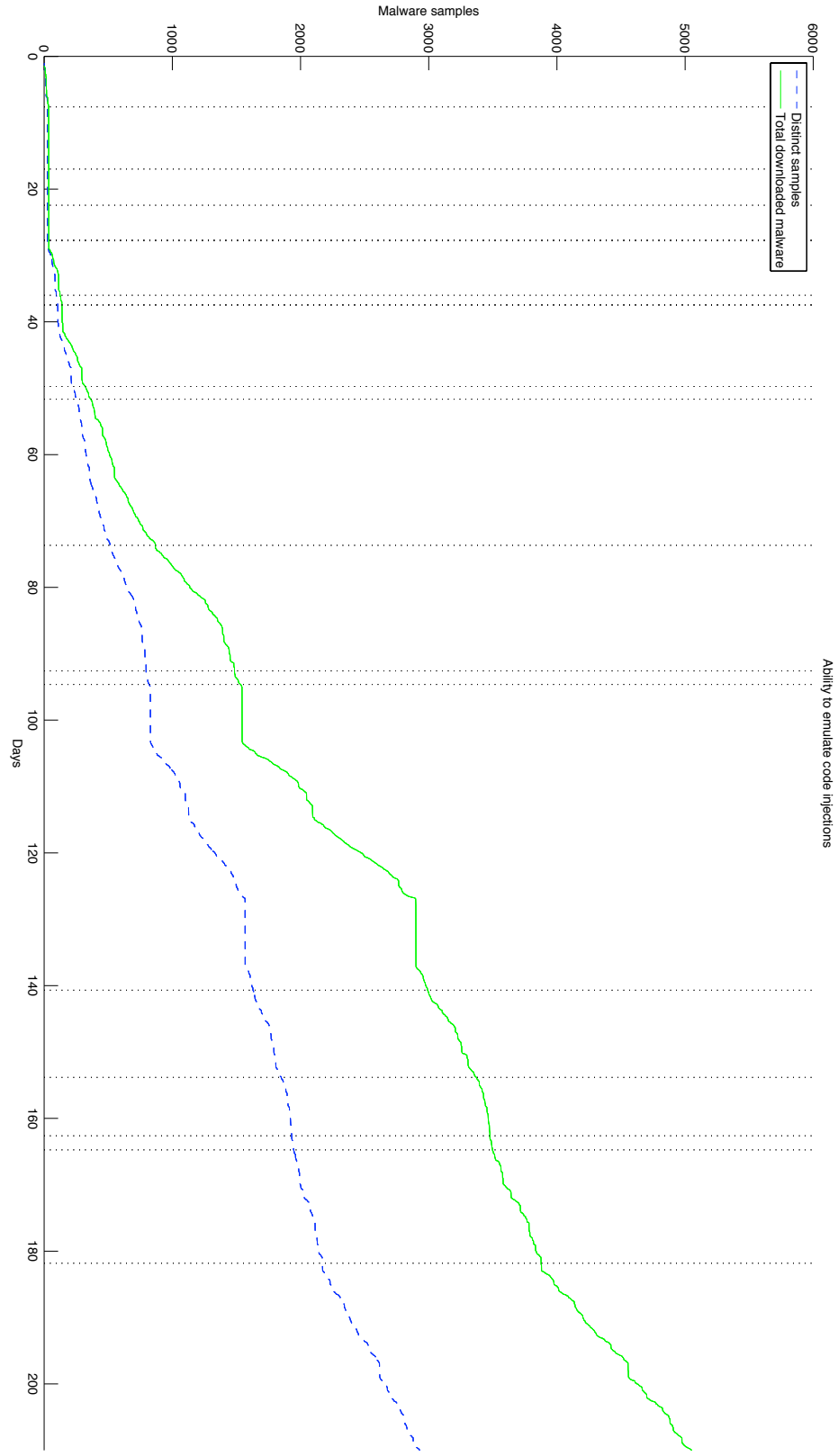

Figure 4.13: Amount of malware collected by the deployment

Sensor ID	Collected samples	Lifetime	Samples per day
19	943	109	8.65
5	1593	211	7.54
26	133	20	6.65
25	259	42	6.17
17	869	147	5.91
15	496	160	3.10
9	364	175	2.08
8	68	38	1.79
2	332	204	1.63
16	187	147	1.27
14	68	60	1.13
6	3	40	0.08
7	7	206	0.03
10	3	125	0.02

Table 4.3: Samples collected by each sensor

terms of distinct MD5s.

Two interesting facts can be deduced from the observation of Figure 4.13.

Firstly, the ratio between the number of malware downloads and the number of distinct samples is constant. This may look counter-intuitive: one would expect the number of different malware samples to converge to a number corresponding to the number of different malware variants currently visible by the deployment. A closer examination reveals that 2521 samples, so 49% of the total, were downloaded a single time along the observation period. This phenomenon is due partly to the extensive usage of polymorphism in modern malware, and partly to the corruption of malware samples in the download phase. Both of these phenomena will be addressed more thoroughly in Section 5.1.

Secondly, differently from the growth in FSM paths, the growth of collected samples seems to be heavily influenced by the addition of some sensors. Table 4.3 provides information on the relationship between the number of collected samples and the days of activity for each of the sensors. Looking at the average number of samples per day, it is clear that the contribution of the different sensors to the collection of malware is extremely diversified. Different sensors are hit by different amounts of code injection attacks and thus contribute in different rates to the growth of the sample set.

4.5 Comparison with similar work

The structure of the SGNET deployment as presented in this work has a number of similarities with existing infrastructures and analysis methods. We preferred to postpone their in depth analysis up to this point in order to be able to make specific

comparisons between our infrastructure and existing work.

4.5.1 Honeyfarms

When considering the general structure of SGNET, it is possible to identify some similarities with the concept of *honeyfarm*[URL 35]. The main concept underneath the definition of honeyfarm is the usage of different types of tunnels to redirect traffic from different locations to a central farm of virtualized hosts acting as high interaction honeypots. Many different implementations of this concept exist in the literature, such as Collapsar [Jiang 2004] or the Honeynet Project Honeymole [URL 41]. The SGNET architecture differs from these approaches in three main points.

Scalability. While in normal honeyfarms all the traffic targeting the monitored addresses is constantly relayed to the virtualized hosts, in SGNET the allocation of a virtualized host to handle an activity is a *rare* event triggered by the observation of a new kind of activity. We saw in Section 4.4.2 that in the 8 months of observation the utilization of the sample factories to handle new activities was reduced, despite of the fact that the number of sensors participating to the collection of data was more than doubled. This gain is obtained through the usage of the ScriptGen FSM knowledge as an evolving *filter*, which handles autonomously all the activities that are already known and thus offloads the sample factories.

Reproducibility. A considerable amount of exploits is effective only when run for the first time on a vulnerable system. That is, many exploits modify the system state. When providing a virtualized host to multiple attackers, this factor needs to be taken into account. With reproducibility we refer to the ability of a system to provide to all the attacking clients exactly the same state, and thus behave in the same way when facing multiple instances of the same attack. Differently from Collapsar and Honeymole, SGNET sets up tunneling towards a high interaction host on a per-attacker basis. That is, each attacker generating an activity that is not part of the FSM knowledge is assigned to a clean snapshot of the vulnerable system to which he is given exclusive access. Such a policy is unfeasible in honeyfarms due to the excessive load, which becomes proportional to the number of attackers witnessed by the system. A similar policy is instead implemented in Potemkin [Vrable 2005], where Copy On Write techniques allow a significant reduction of the instantiation cost of each host. In SGNET instead we successfully implement such a policy thanks to the offloading achieved by the ScriptGen FSMs, which handle at the *border* of the infrastructure the majority of the attacks.

Containment. When handling farms of virtualized hosts, a trade-off exists between the quality of the observations and the security of the system. A virtualized host can potentially be attacked and compromised, and eventually used as a stepping stone by attackers to compromise other systems. It is thus important to carefully define the policy according to which the network interaction of the farm can propagate to the Internet. For instance, it is advisable to allow the farm to perform DNS requests but at the same time it is essential to block any exploit

attempt towards other hosts. The knowledge based approach used by Honeynet Project Honeynets or by Collapsar needs constant maintenance in order to avoid misjudgements in the containment decisions. Taking advantage of memory tainting techniques, SGNET employs a behavior-based containment in the detection of code injections that allows the termination of the host execution as soon as an attacker successfully hijacks the control flow.

4.5.2 Learning the protocol behavior

The idea of automatically learning the protocol behavior to build honeypot responders was first introduced by Chowdhary et al. in [Chowdhary 2004]. In this work the authors proposed to automatically infer the evolution of the protocol state from a set of conversations taking advantage of the input of a human expert to define the protocol syntax. For instance, the approach would automatically infer a model of the HTTP interaction starting from a definition of the syntax of its various message primitives. Differently from ScriptGen, the authors mainly focused on the evolution of the protocol state during, for instance, the human interaction of an attacker interacting with an FTP server and browsing the different directories. Our work aimed instead at the syntactic inference of the protocol structure and on the inference of information of its semantics, with special attention to automated and deterministic attack scripts. Our inference of the evolution of the protocol state is thus less sophisticated, but is enough to suit our requirements.

The idea of using bioinformatics algorithms to perform protocol analysis was initially predated by Marshall Beddoe in the Protocol Informatics Project [Beddoe 2005]. The Protocol Informatics project does not aim at automated protocol analysis; the objective of this work consists instead in the detection of relevant fields in order to help manual reverse protocol engineering. Two different research teams took advantage of such algorithms to perform automated protocol analysis. This resulted in our work on ScriptGen [Leita 2005] on the one hand, and Weidong Cui's work on RolePlayer [Cui 2006c, Cui 2006a] on the other. Both ScriptGen and Roleplayer aim at the automated analysis of the protocol format in order to *replay* conversations. While ScriptGen focuses only on server-side honeypots, RolePlayer is a generic replayer that can reproduce both the server side and the client side of a network conversation. While RolePlayer and ScriptGen share many similarities, they are also characterized by two differences.

Firstly, ScriptGen's region analysis bases the semantic inference on statistical characteristics of a sufficiently large and diverse number of samples. The alignment algorithms used by ScriptGen are normally performed of a big amount of samples considered to be *similar* by the semantic clustering process described in Section 3.3. On the contrary, RolePlayer leverages two conversation samples semantically similar but syntactically different to perform the inference.

Secondly, ScriptGen aims at achieving as much as possible the protocol agnosticism by minimizing the assumptions on the protocol structure and by trying to take advantage of a limited number of very generic heuristics (e.g. intra protocol

dependencies for the cookie fields). RolePlayer instead leverages external information on the samples (e.g. IP address of the attacker/victim, hostname of the victim, ...) and heuristics to recognize this information within the protocol.

RolePlayer was used in a high interaction honeyfarm called GQ [Cui 2006a, Cui 2006b]. While the structure and the purpose of GQ are profoundly different from those of SGNET, many similarities can be identified between the usage of ScriptGen within SGNET and the usage of RolePlayer within GQ. RolePlayer acts in fact as a filter to offload the telescope from known activities, and implements a proxying algorithm similar to that proposed in this work to react to unknown ones. The differences among the two systems are manifold.

- Firstly, ScriptGen and RolePlayer are used in the two deployments with different goals. In GQ, RolePlayer is mainly used as an efficient filter to offload a high interaction honeyfarm. The analysis of the propagation techniques used by the malware, of its propagation time and other statistics reported in [Cui 2006a] is based on the observations of the underlying honeyfarm. In SGNET instead the contribution of the honeyfarm to the data collection task is rather limited. The sample factories are solely instrumental to the generation of information used by ScriptGen for the incremental learning. In SGNET the ScriptGen-based emulation and the resulting classification based on FSM traversals has instead a central role in the generation of the dataset. The interaction of the sensors with the other components of the infrastructure is used to generate detailed information on all the different phases of the code injection attack. We will show in the next Chapter the usefulness of this information in studying the propagation techniques employed by self-propagating malware.

- Secondly, due to its different structure, SGNET employs ScriptGen in a distributed fashion: many distinct sensors deployed in different locations contribute to the shared knowledge. This is compatible with the requirements of ScriptGen, which needs the collection of many diverse samples in order to make meaningful semantic inferences.

- Finally, ScriptGen is used within SGNET for the sole emulation of the exploitation phase. We propose in SGNET an alternation of different components in charge of the emulation of the different phases of a code injection attack. Such "dissection" is used to extract as much information as possible on each observed event along the various dimensions of the epsilon-gamma-pi-mu model. In GQ, RolePlayer is instead in charge of the emulation of the whole attack trace for known activities and, as previously said, does not take active part into the collection of information on the observed events.

Small et al. in [Small 2008] recently presented an alternative technique to automatically build protocol responders without taking advantage of bioinformatics techniques. Their work focuses on ASCII-based protocols (mainly, HTTP) and

takes advantage of a set of pre-processed traces to build protocol responders for honeypots. Taking advantage of clustering and language analysis techniques, the authors propose a method to build responders for the HTTP protocol and use them to monitor the spread of *search worms* [Provos 2006]. Differently from ScriptGen and RolePlayer, this method specifically targets a set of protocols and thus does not aim at being protocol agnostic.

4.5.3 Automated protocol reverse engineering

While ScriptGen and other related work aim at the inference of the protocol syntax only up to the point of the successful replay, a considerable amount of recent work has gone further and has addressed the automated reverse engineering of the protocol structure to produce syntactic/semantic information on the different protocol fields.

Cui et al. in Discoverer [Cui 2007] extended the previous work on RolePlayer. While Discoverer shares with RolePlayer some of the heuristics used for the detection of certain classes of fields, it profoundly differs in the methodology. While ScriptGen and RolePlayer take advantage of alignment techniques on the raw stream bytes in order to detect overlaps and approximations of the different protocol tokens, Discoverer initially tokenizes the protocol stream taking advantage of a set of heuristics. The alignment is used in Discoverer to detect similarities among the different token patterns, and thus join together erroneously tokenized messages.

A considerable number of recent research work takes advantage of static and/or dynamic analysis techniques to look at the way the application parses the network input. While the first approaches were not scalable and could be applied only on very simple synthetic protocols [Newsome 2006], more recent approaches were successfully tested on real-world protocols and on binary file structures [Caballero 2007, Lin 2008b, Lin 2008a, Wondracek 2008, Cui 2008]. For instance, [Wondracek 2008] and [Cui 2008] take advantage of memory tainting techniques applied on multiple messages of the same type to correctly identify the different protocol tokens and assign to each of them a semantic description (e.g. identification of mandatory fields, optional fields, constraints on their values, ...).

The objective of automated protocol reverse engineering goes much beyond the initial objective for which region analysis was conceived. ScriptGen does not aim in fact at fully reconstruct the protocol syntax, but aims instead at understanding the structure of the specific interaction of the observed exploits. We can say that the objective of the ScriptGen FSM abstraction is not that of reverse engineering the *protocol*. ScriptGen aims at reverse engineering the interaction generated by a specific implementation of an exploit. For instance, if ScriptGen observes an attack taking advantage of always the same value in a cookie field, it will incorporate it in a fixed region and will assign it the same semantics of protocol keywords (i.e. the "HTTP" string in an HTTP GET request). This apparently limiting choice allows ScriptGen to distinguish this specific exploit from another implementation that, for

instance, takes advantage of a random value for that field.

4.6 Conclusion

In this Chapter we have presented SGNET, a distributed honeypot framework for the collection of quantitative data on code injection attacks. We have proposed a method to exploit ScriptGen's characteristics and build a system able to incrementally refine its knowledge on network attacks. We showed how it is possible to incorporate into the FSM knowledge information on the structure of code injection attacks supplied by Argos memory tainting techniques. The resulting honeypot emulators become able to correctly observe exploits without any *a priori* knowledge on their structure, following the protocol agnostic approach at the foundations of ScriptGen. We show how we are able to couple the exploit emulation with the Nepenthes shellcode handler, and correctly download malware.

Through the implementation of a prototype of this infrastructure, we provide in this Chapter a detailed analysis of the behavior of ScriptGen when facing real-world attacks. We show that ScriptGen approach is scalable, both in terms of FSM complexity and in terms of load on the sample factories.

While the total number of generated traversals continues to increase over time, the creation of new traversals goes in parallel with the death of old ones. Many of the activities observed by SGNET are in fact ephemeral, and have a very short duration in time. Considering this process of continuous birth and death of traversals, the net number of traversals required to handle the network interaction of all the honeypots in a given timeframe is generally below 200.

The process continuous birth of traversals generates a continuous load on the sample factories. This load does not significantly increase with the increase in number of sensors deployed in the architecture. The reasons for this achievement are twofold. Firstly, the ScriptGen semantic abstraction and the shared semantic knowledge characteristic of SGNET allow the reuse of the same FSM traversal to handle the same type of activity on different sensors. Secondly, the impact of the installation of a new sensor on the observation of localized activities to which the deployment was previously blind is quickly "absorbed" by the system. The FSM learning is normally able to acquire the knowledge required for the normal interaction of a new sensor in a period of approximately 20 days.

The results of the analysis of ScriptGen behavior offer a preliminary insight on the characteristics of the threats observed by the SGNET deployment. More specifically, Figure 4.8 on page 86 shows an interesting picture on the breadth of the network activities. SGNET allows us to depict a scenario in which a limited number of global high-visibility activities is interleaved with a proliferation of diverse, ephemeral activities that are highly localized. It is difficult for the moment to make inferences on the nature of these activities. In the next Chapter, we will present a simple methodology to classify these activities and associate them to contextual information useful to gather intelligence on their ultimate root cause.

Chapter 5

Exploring the epsilon-gamma-pi-mu space

Contents

Chapter 4 introduced SGNET, a distributed honeypot framework based on the ScriptGen learning techniques and used to collect unbiased and rich information on Internet threats. We took advantage of the dataset generated by 8 months of operation of a prototype implementation to validate the ScriptGen approach.

Until now, we have not been able to make any inference on the nature of the observed activities. What kind of malware did SGNET download? How does it propagate? How are the different traversals exploited to spread malware? Is a traversal uniquely associated with a given malware activity? We have started Chapter 4 conjecturing on the increased degrees of freedom in the epsilon-gamma-pi-mu space, and we have built upon this conjecture many of the design choices of the SGNET architecture. The information collected by the real-world operation of the deployment can allow us to evaluate this conjecture.

In this Chapter we introduce a simple data mining technique to categorize the code injection attacks observed by SGNET along the dimensions of the epsilon-gamma-pi-mu space. For this categorization to be possible, the raw dataset generated by the SGNET interaction needs to be enriched with additional information. Little information is available, for instance, on the nature of each malware sample downloaded by the deployment, or on the nature of the observed exploits. In Section 5.1 we present an information enrichment framework that takes advantage of various tools to enrich the raw SGNET dataset with additional perspectives on the observed events. In Section 5.2 we will bridge these different perspectives in a comprehensive classification along the various dimensions of the epsilon-gamma-pi-mu space.

5.1 Information enrichment

The raw SGNET data collected by the deployment activity is processed and enriched on a regular basis by an information enrichment framework [Leita 2008b]. We have built around the SGNET dataset a framework allowing to process the information collected by the deployment and enrich it by taking advantage of different analysis tools and information sources. This process includes sources and techniques inherited from previous work in the Leurré.com project [Pouget 2006] and extends it with additional ones, specific to the SGNET additional data on code injection attack. The interested reader can find a complete overview of these techniques in [Leita 2008c]. In the context of this work, we selected three of the most relevant and interesting ones.

5.1.1 Labelling traversals with knowledge based signatures

In Section 4.4 we saw how SGNET has been able to correctly learn activities and synthesize the different interactions into a set of traversals of the FSMs. While a limited number of traversals was associated with *global* activities, we have been able to see a proliferation of localized activities with very short duration. The total number of traversals generated by SGNET in the 8 months of observation is 438. 70 of these traversals have been tagged as successful code injection attacks by the sample factories. Still, we have no information on the nature of these traversals. From the way they are constructed, we know that many of them are highly specific to a specific honeypot sensor. Also, the same exploit run with different parameters (i.e. different offset in the control flow hijack) is likely to produce different traversals. While the presence of this high amount of specific paths tells us a lot about the diversity of the activities on the Internet, we are not able to know in practice how many different types of exploits are causing these traversals.

In order to have information on the nature of the activities that we are observing, we took advantage of the output of an Intrusion Detection System, Snort [Roesch 1999]. We took advantage of the Sourcefire VRT Certified Rules released on the 15th of July 2008 and of Snort 2.8.1. For every traversal we collected a sample of TCP sessions collected by all available sensors that observed the activity. We limited the maximum size of the sample set to 1000 TCP sessions, which is 1000 traversals. If more samples were available we randomly selected them among the pool of available ones. We then parsed the output of Snort for each of these samples and we defined *sets* of alert identifiers (called Snort IDs) generated by each attack instance. We considered as a representative set for each traversal the most frequent set observed among the samples.

Table 5.1 provides an overview on the characteristics of the different groups of traversals associated with the same alert set:

- **References:** when applicable, references to the full vulnerability report.

Alerts	Ports	References	NP	NIJP	N1	Q1	Q2	Q3	Q4
WEB-PHP test.php access	80		4	0	48380	3035	45345	0	0
WEB-IIS WEBDAV nessus safe scan attempt WEB-MISC WebDAV search access	80	MS03-007	3	0	1321	0	0	0	1321
WEB-IIS cmd.exe access	80		1	0	709	709	0	0	0
WEB-MISC cross site scripting attempt	80		1	0	594	0	594	0	0
WEB-IIS view source via translate header	80	MS00-058	6	0	2520	745	0	1775	0
TELNET Solaris login environment variable authentication bypass attempt	23	CVE 2007-0882	6	0	98	22	76	0	0
SHELLCODE x86 NOOP SHELLCODE x86 inc ebx NOOP SPECIFIC-THREATS ASN.1 constructed bit string	445	MS04-007	25	9	25215	1265	0	23950	0
SHELLCODE x86 NOOP NETBIOS SMB-DS srvsvc NetrPathCanonicalize WriteAndX unicode little endian overflow attempt	445	MS06-040	3	2	11	11	0	0	0
SHELLCODE x86 NOOP NETBIOS SMB srvsvc NetrPathCanonicalize WriteAndX unicode little endian overflow attempt	139	MS06-040	3	0	9	9	0	0	0
NETBIOS SMB-DS srvsvc NetrPathCanonicalize little endian overflow attempt SHELLCODE x86 NOOP	445	MS06-040	1	0	28	28	0	0	0
NETBIOS SMB srvsvc NetrPathCanonicalize little endian overflow attempt SHELLCODE x86 NOOP	139	MS06-040	1	0	52	52	0	0	0
SHELLCODE x86 0x90 unicode NOOP NETBIOS SMB-DS lsass DsRolerUpgradeDownlevelServer WriteAndX unicode little endian overflow attempt NETBIOS SMB-DS lsass DsRolerUpgradeDownlevelServer unicode little endian overflow attempt	445	MS04-011	8	0	84	84	0	0	0
SHELLCODE x86 0x90 unicode NOOP NETBIOS SMB-DS lsass DsRolerUpgradeDownlevelServer WriteAndX unicode little endian overflow attempt	445	MS04-011	4	0	18	18	0	0	0
SHELLCODE x86 NOOP NETBIOS SMB-DS lsass DsRolerUpgradeDownlevelServer unicode little endian overflow attempt	445	MS04-011	1	0	10	10	0	0	0
SHELLCODE x86 0x90 unicode NOOP NETBIOS DCERPC NCACN-IP-TCP lsass DsRolerUpgradeDownlevelServer little endian overflow attempt	135	MS04-011	1	0	470	0	0	0	470
SHELLCODE x86 NOOP NETBIOS DCERPC NCACN-IP-TCP ISystemActivator RemoteCreateInstance little endian attempt	445	MS03-026	20	11	7432	938	304	5423	767
NETBIOS DCERPC NCACN-IP-TCP ISystemActivator RemoteCreateInstance little endian attempt	135	MS03-026	3	0	681	64	617	0	0
NETBIOS DCERPC invalid bind attempt	135	MS03-026	1	0	6650	0	0	6650	0
SHELLCODE x86 NOOP	139		1	0	2526	0	0	2526	0
SHELLCODE x86 NOOP	135,445		6	2	17382	129	91	17162	0
SHELLCODE x86 NOOP SHELLCODE x86 inc ebx NOOP	139,445		47	41	96587	1448	0	85028	10111
Unknown activities			297	0	442736	33398	5606	362562	41170

Table 5.1: Groups of traversals according to the generated alerts

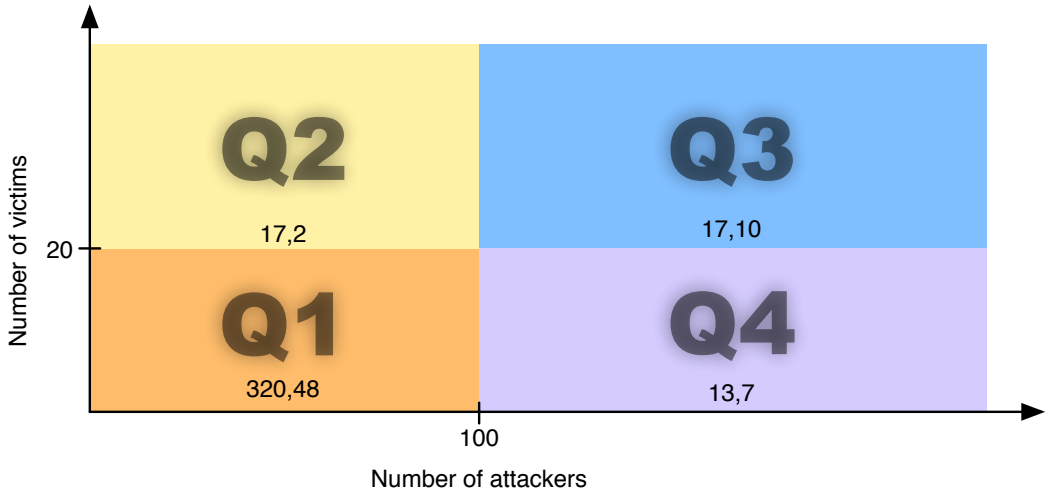

Figure 5.1: Locality of traversals

- **NP:** total number of distinct FSM traversals that raised this alert set.
- **NIJP:** total number of FSM traversals that raised this alert set and that are tagged by the sample factory information as successful code injection attacks.
- **NI:** total number of attack instances hitting one of the traversals in the group.

In order to retrieve information on the scale of the observed activities and compare it with the nature of the alert, we reuse a concept introduced in Figure 4.8 on page 86. We showed how the different traversals could be classified in 4 different types according to their breadth in terms of number of attackers and of victims involved in the activity. This concept is summarized in Figure 5.1 where for each of the 4 quadrants of the graph we provide a comma-separated tuple representing the number of traversals and the number of traversals leading to successful code injections.

- **Q1:** activities involving a low number of attackers and victims, thus highly localized. The majority of the traversals generated by SGNET are included in this area, even if this area accounts for less than 6% of the total amount of attacks observed in SGNET in the observed period.
- **Q2:** activities involving a low number of attackers hitting a large number of victims. We can hypothesize that these activities correspond to high breadth scanning activities performed by individual attackers.
- **Q3:** global activities involving a large number of attackers and victims. These activities are likely to be associated with global phenomena such as the spread of worms or big botnets.

- **Q4:** activities involving a high number of attackers but focused on a small number of victims. This kind of coordination may be explained by the presence of botnets coordinated through C&C channels. This would explain the presence of a high number of attackers focusing their scanning activity on a very localized portion of the IP space.

While the attribution of the different quarters of Figure 5.1 to the different types of activity is purely a conjecture and, at this moment, cannot be verified methodically, it give hints on the nature of the observed activities.

Looking at table 5.1, we can see that the 438 distinct traversals could be grouped in just 22 groups. An interesting and surprising fact can be immediately spotted by the last lines of table 5.1: the biggest group, accounting for a total of 442736 attack instances (67.9 % of the total) is associated with an empty alert set. Moreover, other three groups are solely associated with alerts belonging to the *shellcode.rules* set of rules, which searches for common repetitions of NOOP instructions into protocol payloads. Due to the high number of false positives associated with these rules, they are normally disabled. Out of the 652296 attack instances observed by SGNET in the considered period, Snort generates meaningful alerts only for 93065 of them, that is, only 14%. Among the undetected activities, a considerable number of instances is associated with a single traversal, hit 303812 times and belonging to quarter Q3, corresponding to the brute force attack on the SMB shares run by the Allaple worm as part of its propagation strategy (see [URL 12]). Also, 43 of the traversals that are not associated with any meaningful alert have been tagged by SGNET as successful code injection attacks. This clearly shows a major deficiency of Snort in correctly identifying malicious traffic.

A considerable number of alert groups, associated to well-known vulnerabilities exploitable with code injection attacks, are not associated in practice to any successful code injection. A possible explanation for this behavior might be insufficient interactivity of the honeypots, leading the attackers to prematurely abandon the conversation. Manual analysis of the conversation samples associated to these alert groups ruled out this hypothesis. The exploitation attempts for these vulnerabilities are fully emulated by SGNET honeypots, but the Argos memory tainting did not flag such activities as successful in hijacking the control flow. The most likely cause for this behavior is that the exploitation attempts were tuned for a different OS configuration than the profile used in the experimentation.

Looking at the relationship between alert sets and traversals, we can see that in most cases multiple traversals have been associated with the same alert set. Attack instances hitting different FSM traversals are associated with differences in the protocol interaction with varying semantic relevance. Two attack instances classified in the same way by Snort are thus often considered different by ScriptGen. By looking at Snort rules, we can see that Snort often aims at detecting the necessary condition for an attack to take place, focusing on conditions that do not normally happen in a benign application. For instance, Snort detects attempts to use the Windows DCERPC protocol to bind to vulnerable functions that are not normally

invoked remotely. Since ScriptGen traversals classify activities according to the entirety of the network interaction, they are likely to identify differences among activities that Snort is unable to spot.

Looking more in detail at the different groups associated with each alert set, we can identify some interesting facts.

Firstly, all the alert groups on port 80 are not associated with code injection attacks but are associated with application-level threats. For instance, the most popular alert group consists in the "test.php access" [URL 34]. Certain PHP based applications can provide useful information to the attacker through a "test.php" script accessible in the web server root. This activity is likely to be associated with scanning attempts in order to discover the presence of such PHP applications on the Internet. As a corroboration to this hypothesis, most of the activities associated with this class of attack belong to the group Q2. Such observation is compatible with the hypothesis of a small number of sources taking advantage of automated tools to discover vulnerable servers. It is very interesting to see that WebDAV search scans [URL 33] belong instead to the group Q4. This kind of activity is indeed extremely localized and almost always hits a *single* sensor. The partner hosting this sensor has already been targeted in the past by Distributed Denial of Service attacks based on WebDAV search requests targeting one of their webservers. Our hypothesis is that attackers are continuously scanning the network of this partner, probably taking advantage of a botnet, and are searching for WebDAV enabled web servers that could be used to repeat such an attack. Only a single alert group involving port 80 is instead associated with a globally spread activity (group Q3), and consists in an attempt to disclose ASP scripts' source code using the "Translate:" header [URL 32].

Most of the attacks observed by the SGNET deployment are concerning the typical Windows Netbios ports, namely TCP ports 445, 139 and 135. On these ports, we can identify multiple groups characterized by different combinations of significant alerts and of alerts related to the detection of NOOP instructions in the payloads. The latter alerts are generated by a set of signatures (*shellcode.rules*) that while included in the VRT Certified Rules are normally not enabled by default due to the high rate of false positives associated with them. The different groups can be associated with 4 different vulnerabilities, namely MS04-007 (ASN.1 library heap overflow), MS06-040 (Server Service NetpwPathCanonicalize Overflow), MS04-011 (LSASS Service DsRolerUpgradeDownlevelServer Overflow), MS03-026 (RPC DCOM Interface Overflow).

Despite the fact that the sample factory used to perform the learning *is* vulnerable to all the 4 vulnerabilities, a good number of alert groups do not contain successful code injection traversals, and those that do also contain unsuccessful traversals. We believe this phenomenon is associated with the exploit parameters. For instance, the LSASS exploit previously seen in Section 3.5.1 on page 53 can target different OS configurations: Windows XP Professional, Windows 2000 Professional, and Windows 2000 Advanced Server. The target is specified by a command line parameter when running the exploit and leads to different invo-

cations of the vulnerable service according to the characteristics of the targeted Operating System. When running the exploit against an SGNET sensor using the wrong target code, no code injection is detected and the exploit fails. SGNET observed a small number of exploitation attempts taking advantage of the LSASS exploit both on port 445 and on port 135. None of them led to a successful code injection though.

The two main classes of exploitation vectors observed by the SGNET deployment involve the MS04-007 ASN.1 exploit and the MS03-026 RPC DCOM Interface overflow. While the first one has high predominance in the Q3 group and is thus associated with global activities, the second one has a significant amount of representatives in all of the 4 areas of Figure 5.1. The RPC DCOM exploit is the main propagation vector for the Allaple infection [URL 12], a polymorphic worm that we have observed since the beginning of our experiments, and other older worms such as Blaster [URL 8] or Welchia. Also many different bots are known to be taking advantage of this exploit to propagate [The Honeynet Project 2005c]. It is thus possible that we are witnessing a superposition of different kinds of activities hitting our honeypots, both global scanning performed by hosts infected by the previously mentioned worms and other more coordinated or more localized scans performed by botnets of different size.

We witnessed exploitation attempts taking advantage of the MS06-040 vulnerability. Such exploitations are rare, and have been observed towards the end of the observation period (last week of June 2008). According to the SGNET observations, this vulnerability does not seem to be actively exploited yet by any large scale phenomenon, but only by the isolated activity of single attackers.

We already saw that a majority of the attack instances observed by SGNET is not associated with any meaningful alert. Among these instances, those associated with traversals tagged as successful code injection attacks attracted our attention.

Two code injection traversals belong to an alert group composed of a single x86 NOOP alert. A manual inspection revealed both traversals to be associated with a MS03-026 DCE RPC exploit on port 135, and accounted to a total of 15968 successful code injections. An alert group exists for this exploit on port 135, but in such alert group no successful code injection attack is witnessed. A more in-depth analysis revealed that out of a sample of 1000 samples of activity associated with these traversals, the corresponding alert is raised only 14 times and is not thus comprised in the representative alert set. The alert associated with the DCE RPC exploit (SID 8690 [URL 31]) takes advantage of the *flowbits* Snort directive to reduce false positives. Taking advantage of the *flowbits* directive, it is possible to add statefulness to Snort rules. In this case, the alert for the exploitation of the IActivation interface depends on the previous bind to this interface on the same DCE RPC connection. If Snort does not detect such a bind attempt in the previous interaction, the alert is not triggered. For an unknown reason that we have not been able to pinpoint, the bind attempt is not detected by Snort in the conversation samples and thus once the exploitation is witnessed the alert is not raised. We contacted the Snort development team and notified them about the issue.

41 FSM traversals leading to successful code injections have been grouped together as a consequence to two alerts generated by the detection of different NOOP instructions in the protocol stream (x86 NOOP, x86 inc ebx NOOP). Of the 7692 successful code injections instances belonging to this group, we found out that 5276 were associated with a single traversal. The FSM traversal involves an interaction on port 139 and involves a malformed field in the invocation of the SPNEGO (Simple and Protected GSSAPI Negotiation Mechanism) authentication in a Netbios Session Setup AndX request, a known way to exploit the ASN.1 MS04-007 vulnerability. The exploit has very similar structure to the one witnessed on port 445 and correctly recognized by Snort, with the addition of the initial NetBT (NetBios over TCP/IP) session establishment that is not required for the interaction on port 445. The manual analysis of the Snort signature with SID 12710, part of SourceFire's *specific-threats.rules* and associated with the ASN.1 exploit, showed that the alert would correctly match also the exploitation attempt on port 139. The Session Setup AndX request used in the two cases has in fact exactly the same structure. But the signature does not contemplate the analysis of packets on TCP ports other than 445 and the alert is thus not triggered. Interestingly enough, also the implementation of knowledge-based honeypots such as Nepenthes for this vulnerability does not contemplate an interaction on port 139 involving this vulnerability.

Summarizing, the Snort knowledge-based information on the network interaction associated with the different traversals offers an interesting perspective on the types of activities observed by SGNET. Only a few vulnerabilities are underneath the whole set of code injection attacks observed by SGNET. While the number of vulnerabilities is low, the scenario is far from being simple. The FSM-based classification used within SGNET allowed us to underline a proliferation of different traversals likely to be associated with different variants of the exploitation tools, and also different methodologies to trigger the same vulnerability. In such a complex and diverse scenario knowledge based approaches fail to exhaustively handle all the possible entry points for an attacker to exploit its victims: only a small percentage of the activities observed by SGNET is associated with a meaningful alert by Snort. These observations corroborate previous observations in the field on the limitations of knowledge-based Intrusion Detection Systems in the detection of Internet attacks [Zurutuza Ortega 2007] and clearly show the importance of collecting quantitative intelligence on the attack threats to improve their performance.

5.1.2 Malware behavior analysis

Until now, we have focused only on the network interaction and on the properties of the information generated by ScriptGen. The emulation capabilities of SGNET allow going beyond the exploit phase and, among other things, allow the collection of malware samples. SGNET does not give us any information on the nature of the collected samples. What do these samples do once executed on a

real host? What level of sophistication are we observing in modern malware? In order to find answers of these questions we took advantage of the information offered to us by Anubis. Anubis[1] is a malware analysis tool created by Bayer et al. [Bayer 2005, Bayer 2006]. It analyzes the behavior of Windows executables by running them in an emulated environment and monitoring their actions. Differently from many other sandbox solutions such as CWSandbox [URL 9], Anubis performs the analysis in an unobtrusive way by monitoring the OS from the standpoint of a CPU emulator, making its detection more difficult.

Every malware sample downloaded by SGNET is automatically submitted to Anubis on a daily basis, and the information generated on the sample behavior is stored in the SGNET dataset in an aggregated form. This information greatly enhances our knowledge on the ultimate purpose of the observed code injection attacks, and on the nature of the observed threats.

Table 5.2 shows an overview on the different characteristics of the malware samples collected SGNET. The different columns represent different high-level behaviors identified by the Anubis sandbox:

- **AUTOSTART:** the sample registers processes to be executed at system start.
- **INTERNETSETTINGS:** the sample modifies the settings of Internet Explorer.
- **WINDIRCOPY:** the sample creates files in the Windows system directory in order to hide its modifications.
- **DLF:** the execution of the sample leads to the download of additional binaries from the Internet.
- **IRCBOT:** the malware joins an IRC network.

To these high level behaviors, we add the CIDR prefix scanned by the malware during the sandbox execution. The results underlined in Table 5.2 are quite interesting in different aspects.

First of all, an important portion of the malware collected to SGNET can be associated with low-sophistication threats that do nothing else than scanning the Internet in an attempt to propagate. According to the Anubis information, out of 5325 malware downloads 927 (17%) of them lead to the download of low-sophistication samples. Only 191 downloads (4%) led to the collection of samples coordinated by an IRC channel.

Secondly, Anubis provides us with information on the connection attempts performed by the samples while executed. This information can be incomplete: the sample is executed by the sandbox for a default period of 2 minutes, which may not be sufficient to observe the scanning behavior of the sample. For the samples for which we observed outbound scanning, the scanning seems to be very focused on narrow areas of the IP space. Most samples focus their scanning on class B (/16) networks or smaller.

[1] http://anubis.iseclab.org/

AUTOSTART	INTERNETSETTINGS	WINDIRCOPY	DLF	IRCBOT	Scan prefix	Count
no	no	no	no	no	16	927
yes	no	no	no	no	16	803
yes	no	no	no	no	no	448
no	no	no	no	no	no	277
yes	**yes**	no	no	no	no	99
yes	**yes**	yes	no	**yes**	no	58
yes	**yes**	no	no	**yes**	23	55
yes	**yes**	no	no	**yes**	no	31
yes	**yes**	**yes**	no	**yes**	27	18
yes	**yes**	**yes**	no	**yes**	23	17
no	**yes**	no	no	no	no	10
yes	**yes**	**yes**	no	no	no	8
yes	no	no	no	no	17	6
no	no	no	no	**yes**	no	6
yes	no	no	no	no	18	4
yes	**yes**	**yes**	no	**yes**	22	3
no	**yes**	no	no	no	16	2
yes	**yes**	**yes**	**yes**	no	0	1
yes	**yes**	**yes**	no	**yes**	26	1
yes	**yes**	**yes**	no	**yes**	25	1
yes	**yes**	no	**yes**	no	5	1
yes	no	no	no	no	20	1
no	**yes**	**yes**	no	**yes**	no	1
17/6	15/8	9/14	2/21	10/13	13/10	

Table 5.2: Samples collected by each sensor

Thirdly, 1250 downloads (23% of total) led to the collection of a sample that could not be executed within the Anubis sandbox. Examining more closely these samples, we detected truncations of segments of the file and often invalid header information. The cause for this phenomenon seems to be found in network errors (e.g. premature connection termination) in the malware download phase. The Nepentes-based shellcode handler does not seem to detect these conditions, and stores an anomalous file as sample without providing any warning on the abrupt connection termination. We assume this problem to be a generic problem of the download protocols implemented in Nepenthes, and so to be applicable also to other malware repositories based on similar malware download techniques. The Nepenthes developers team has been made aware of the issue.

We have previously identified a considerable amount of malware samples (2521 samples) that were downloaded only once during the observation period of SGNET. We attributed this phenomenon to the increasing spread of polymorphic techniques in modern malware. A basic form of polymorphism consists in re-packing the malware sample at every propagation attempt using a random seed. Such a technique ensures each malware sample to completely mutate its binary content at every generation, making its detection much more complex to AV vendors. From our standpoint, the employment of such techniques leads to the proliferation of unique samples (downloaded only once) and makes the problem of attribution of two events to the activity of the same malware type much more complicated. How to define two completely different binaries to be *similar* and thus attribute two different code injections to the activity of the same malware?

The problem of malware classification is an extremely interesting problem widely addressed in current research [Bailey 2007, Rieck 2008]. While addressing this problem in-depth is out of the scope of this work, we have taken advantage of simple techniques to identify similarities between two different instances of a polymorphic malware. The behavioral information provided by Anubis proved to be instrumental to the achievement of this goal. For instance, one of the first polymorphic malwares that we are aware of is the Allaple worm [URL 12]. Allaple is characterized by the creation of a file, named "urdvxc.exe". 1174 of the 2521 unique samples generate a process with such name. In a first approximation, searching for the above string in the behavioral information provided by Anubis would seem to be sufficient to group together the whole Allaple family.

A more in depth analysis reveals that the problem is far more complex. The group of 1174 samples generating a process named "urdvxc.exe" does not have in fact uniform characteristics. For instance, 501 of these samples also generate an additional file, *C:\Documents and Settings\user\Local Settings\Temporary Internet Files\Content.IE5\012N45I3\ccxebztz.exe*, which is not generated by the other samples. Moreover, there seem to be important structural differences in the different malwares composing this set.

Despite the polymorphic nature of these samples, the distribution of their file size is not random. Figure 5.2 clearly shows that a significant amount of samples is clustered around very specific file lengths. 242 samples for instance have a size of

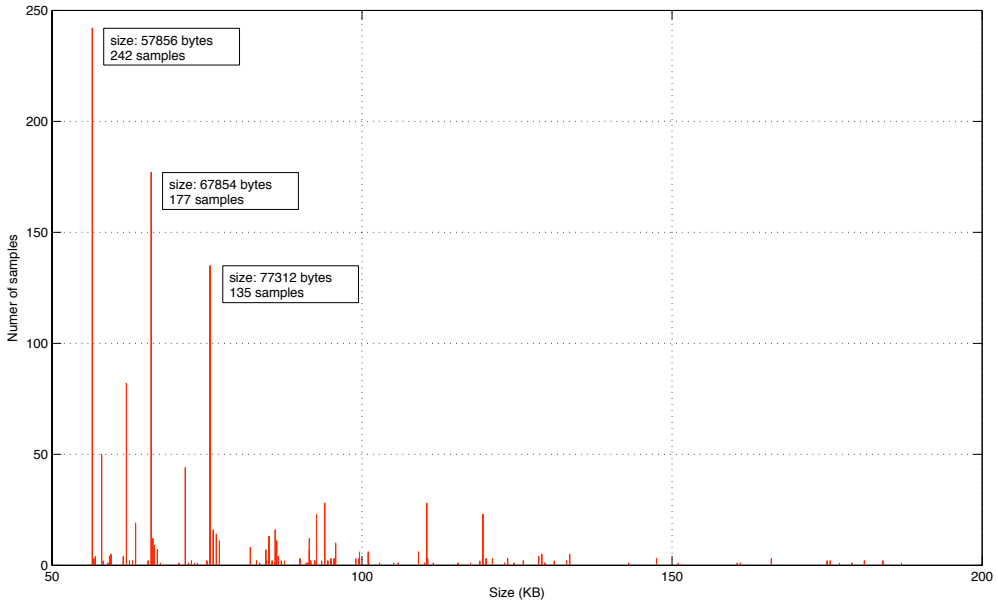

Figure 5.2: Distribution of the file size for samples creating the urdvxc.exe process

exactly 57856 bytes, while 177 of them are 67584 bytes long. This strongly suggests the existence of subgroups among the identified set, with possibly different origins or different microscopic characteristics that cannot be identified from the high-level behavioral analysis taken into consideration so far.

A possible explanation for the existence of different classes of samples sharing the same size could be found in different variants of the same source code, recompiled by the same author or by different authors sharing the same code base. In order to investigate this theory, we tried to look at some of the header information provided by the Packed Executable format taking advantage of the *pefile* library [URL 7]. We assume in fact that different compilers or different revisions of the code

Group #	Linker	Machine	OS	sections	Count	First seen
1	5.12	0x014c	0x40	3	886	?? (1-JUN-07)
2	5.12	0x014c	0x40	5	140	?? (1-JUN-07)
3	5.12	0x014c	0x40	4	103	?? (1-JUN-07)
4	8.12	0x014c	0x40	3	37	1-OCT-07
5	8.12	0x014c	0x40	7	8	4-AUG-07

(The earliest data contained in the SGNET dataset is dated back to the 1st of June 2007)

Table 5.3: Value of some PE header fields for samples creating the urdvxc.exe process

may lead to different characteristics at PE level. Table 5.3 partially backs up this hypothesis: the 1174 samples are grouped into 5 different groups, mainly differing for the number of PE sections and the linker version. Indeed, this result suggests the existence of different variants, sharing the same high level behavior but with structural differences. Also, the sets generated grouping the samples according to their sizes are all subsets of the sets generated by the PE information. Two samples with same size differ in content as a result of their polymorphic nature, but always have the same PE header information. Also, looking back in the data collected by the experimental setup deployed in June 2007 we can see that group 4 and 5 are relatively recent: they appeared respectively in October and August 2007. No information is instead available on the other three groups: it is likely that their birth precedes the date in which we started the data collection.

The problem of polymorphic malware is a very interesting problem. Spread of malware using such techniques leads to a proliferation of samples that make the problem of the assessment of the similarity of two different threats a hard problem. We focused here our attention on Allaple, one of the main contributors to polymorphic malware in our dataset. Even this relatively simple malware shows the difficulty of the underlying problem. Samples sharing very similar or identical high level behavior reveal differences and subgroups when analyzed from more low-level perspectives. Such a scenario is probably an effect of the extensive sharing of code among attackers and the consequent proliferation of different variants of the same code base. This scenario is continuously evolving, and leads to a phylogenesis of different mutations of malware code.

5.1.3 Malware labelling using AV information

When collecting malware samples with SGNET, little or no information is available a priori on the nature of the threats. We have seen that behavioral information provided by Anubis helps in understanding what a given malware is doing and to detect similarities among instances of polymorphic malware. This behavioral information is although not helpful in weighting the *relevance* of a sample. We want a technique allowing us to easily distinguish among the propagation of well-known malware such as Blaster [URL 8] and the propagation of new malware variants on which little or nothing is known. In order to make this separation possible, we took advantage of the analysis output of AntiVirus (AV) engines.

VirusTotal [URL 45] is a free web service provided by Hispasec Systemas, an IT Security Laboratory in Malaga. VirusTotal receives malware samples and provides in exchange a comprehensive report on the signatures assigned to the sample by various commercial AV solutions supported by the service. At the moment of writing, VirusTotal supports 36 different antivirus softwares, whose signatures are updated daily. Also, the samples received by the VirusTotal service are automatically shared with the different vendors: submitting malware to VirusTotal indirectly helps the AV community to refine the signatures for the various products.

All the malware collected by the SGNET deployment is automatically submitted

Algorithm 5 submission_policy(*sample*)

```
n ⇐ get_number_submissions(sample)
last_reports ⇐ get_reports(sample)[n − 7 : n]
if n ≥ 30 and all_equal(last_reports) then
    submit ⇐ False
else
    submit ⇐ True
```

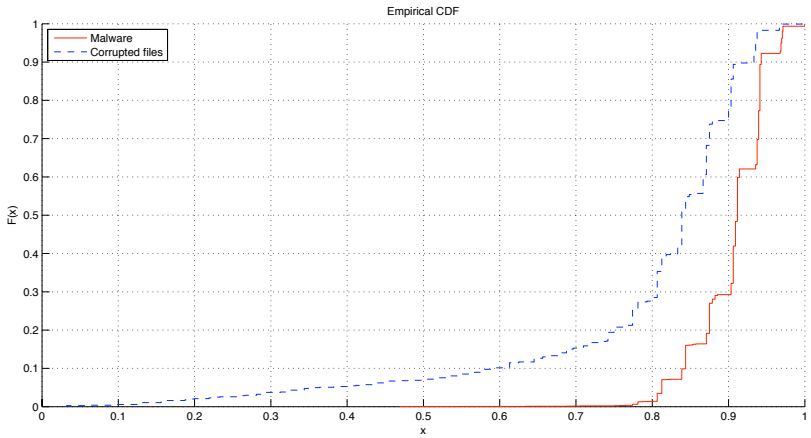

Figure 5.3: Cumulative distribution function for the ratio of vendors recognizing each sample

to VirusTotal, and the generated report is stored in the SGNET dataset. While in Anubis each sample is submitted and analyzed only once, the submission policy for VirusTotal is more sophisticated and is schematized in Algorithm 5. In order to observe the evolution of the labels provided by each vendor for the sample we collected, we repeat the analysis for at least 30 days.

When trying to evaluate the ability of the different vendors to recognize the collected samples, we encountered a rather peculiar problem. Some samples, recognized by some vendors as well-known and rather old types, were never recognized by other vendors. When cross-correlating this information with the Anubis information we identified these low-recognition samples with those samples that Anubis flagged as non-executable due to missing portions of the binary. In other words, the 1250 corrupted samples identified by Anubis are often not recognized by many AV vendors.

In order to evaluate the impact of sample corruption on AV performance, we compared the performance of all the AV vendors with the two classes of samples. As a measure of performance, we took into consideration the recognition rate, which is the ratio between the number of AV vendors that correctly recognized a sample and the total amount of AV vendors provided by VirusTotal. Among all the reports generated by the resubmission policy for each sample, we selected

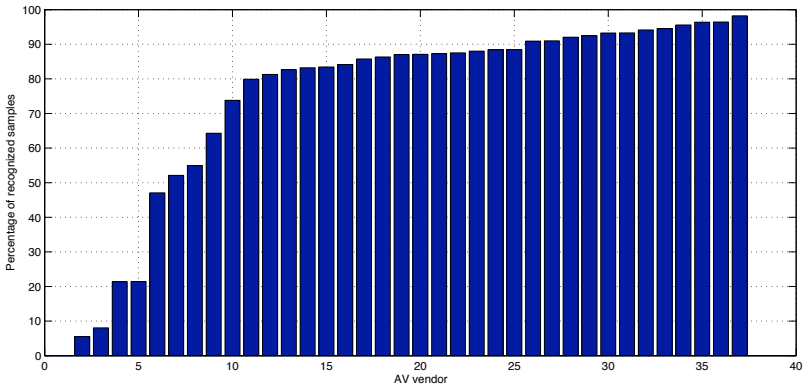

Figure 5.4: Performance of different vendors when analyzing corrupted samples

MD5	Packer	First seen
2b68b2ca1ab620f33d220411f93941bf	kkrunchy 0.23 alpha	09-APR-08
39b81ab57624d9b174d9f13e0b73691a	EXECryptor 2.2.4	16-FEB-08
5865e732663d75b501ffd7d98bc49005	MoleBox V2.3X	31-JAN-08
a3781d5747a5bd632d0966e061416088	Upack v0.33 - v0.34 Beta	31-JAN-08
a5e409732960219264fbb643ad982c2c	ASProtect v1.23 RC1	09-APR-08
e0d35579ef892259370a08dd938a15e3	UPX 2.90 [LZMA]	16-MAR-08

Table 5.4: Samples not recognized by any vendor

the most recent one, which corresponds to the best recognition rate. Figure 5.3 shows the CDF for the recognition rates achieved by the AV vendors for the two classes. Whereas 80% of the vendors always recognize the regular malware samples, the CDF for the corrupted ones is much less steep and underlines a significant difference in performance of the various AV solutions.

The output of an AV engine when analyzing such corrupted samples is not easy to define. Should a binary that cannot be executed be considered as malicious? On the one hand, the specific implementation of an engine or of the corresponding signature may or may not be affected by missing parts of the original binary. On the other hand, different vendors may have different policies with respect to these corrupted files. Figure 5.4 validates this intuition. Each bar on the X axis corresponds to one of the 37 vendors supported by VirusTotal, and represents the percentage of corrupted malwares recognized by that vendor. A minority of the vendors raises almost no alerts for the corrupted files, considering them harmless. But the majority of AV products ignores the inconsistent structure of the executable file (easily detectable looking at its headers) and considers the sample as malicious.

The identification of the corrupted files performed by Anubis allows us to filter out this ambiguity and focus on the 2088 samples that can be correctly executed within a real OS. Of these 2088 samples, only 20 were correctly flagged as malicious

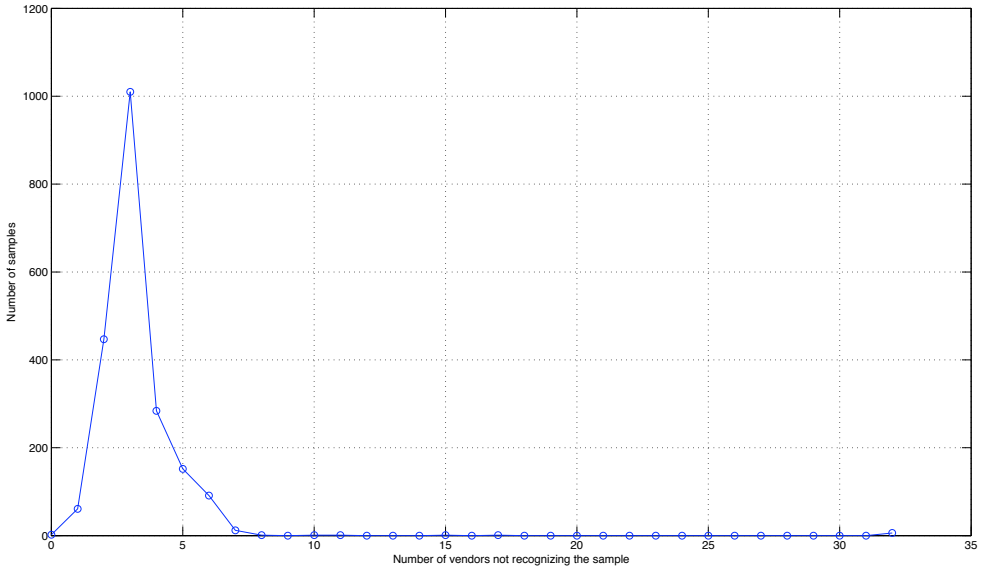

Figure 5.5: Number of vendors not recognizing a given sample

by *all* the AV vendors provided by VirusTotal. As shown in Figure 5.5, most of the samples are recognized by all but three vendors, and only 11 of them are undetected by more than 7 vendors.

By looking at Figure 5.5 it is clear that the vast majority of the malware samples observed by SGNET honeypots was already known to most vendors. This fact is not surprising for several reasons. Firstly, the scope of the experimental deployment on the IP space is rather limited compared with the extensive deployments used by Antivirus vendors to collect new samples and update their signatures. Being hit by a new malware variant before it is witnessed by the vendors is thus a rather unlikely event. Secondly, as previously explained many of these samples employ polymorphism: a large portion of unique samples is likely to be associated to a small number of polymorphic worms, and this biases the results.

It is quite interesting to identify in this set 6 samples that in the submission period did not trigger any alert from any of the 32 vendors active at that moment in VirusTotal. These samples are listed in Table 5.4, with the information on the packer provided by the PeID database. In fact, all these samples take advantage of a commercial or well-known packer. A minority of the samples that we observe take advantage of such packers: only 31 samples out of 2088 take advantage of a packer recognized by the PeID database, while in most cases the packer, if existent, is undetected. According to the claims of some vendors [URL 37], some of these recognized packers offer highly sophisticated obfuscation techniques. We can thus hypothesize that the presence of such packers has made the detection of the samples

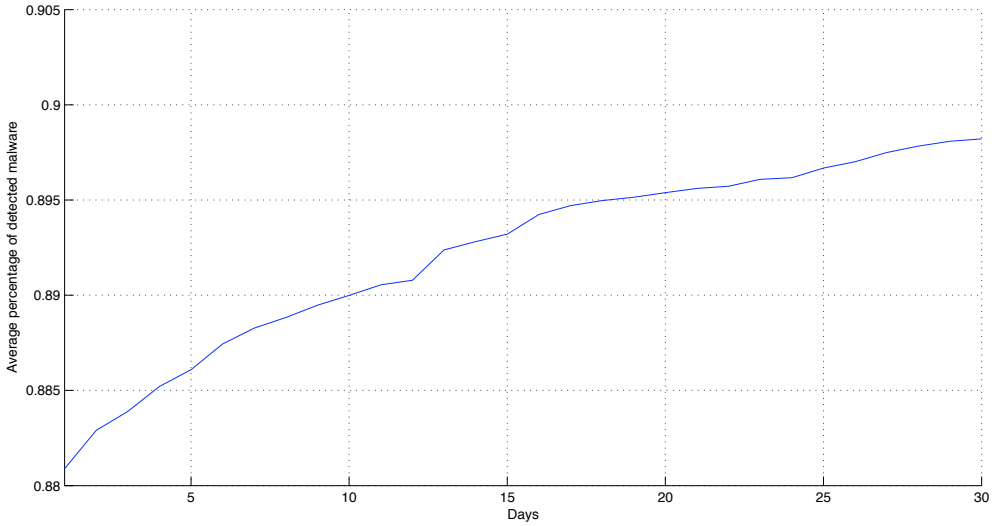

Figure 5.6: Evolution of the detection performance along the days of re-submission

more complex to the different AV vendors, and this led to a delay in the ability of vendors to take advantage of the submissions to update their signatures.

In normal conditions, many vendors seem to actively react to VirusTotal submissions, refining their signatures to recognize the new samples. Figure 5.6 shows the ratio of samples that, on average, the different vendors have been able to recognize in the different days in which each sample was resubmitted. The improvement between the first day of submission and the 30th is of approximately 1%, thus very small. But looking at the single vendors, the active reuse of samples received via VirusTotal is more evident. For instance, in 14 cases a vendor did not recognize the sample at the first submission, but immediately refined its signatures so that the same sample was recognized the day after. Other vendors exposed similar behaviors but with longer delays.

The reaction of AV vendors to our submissions to VirusTotal is not necessarily to be considered a good thing. The 14 samples that were correctly recognized by a vendor at the second submission were all variants of a polymorphic malware. The fact that the vendor was able to recognize a sample only after having seen one indirectly meant that the vendor was not able to cope with the polymorphic packing technique used by such sample, and that was building signatures for each separate instance of the packed executable. This underlines a generic problem of the AV community when dealing with the proliferation of packers for malware: the lack of unpacking routines allowing to build signatures on the unpacked binary. This problem is currently being addressed by additional components included in modern AV systems, components that in most cases are disabled in the command-line interface used by VirusTotal.

Allaple.gen6	3	Net-Worm.Win32.Allaple.e
	106	Net-Worm.Win32.Allaple.b
W32/Virut.P	1	Net-Worm.Win32.Allaple.d
	16	Virus.Win32.Virut.q
	1	Backdoor.Win32.Rbot.bni
	7	Net-Worm.Win32.Allaple.e
	1	Net-Worm.Win32.Allaple.b
W32/Virut.BF	2	Backdoor.Win32.VanBot.ps
	159	Virus.Win32.Virut.n
Allaple.gen10	10	Net-Worm.Win32.Allaple.e
	1	Backdoor.Win32.Rbot.bni
Allaple.gen1	37	Net-Worm.Win32.Allaple.d
	85	Net-Worm.Win32.Allaple.e
	17	Net-Worm.Win32.Allaple.b
W32/Virut.T	2	Net-Worm.Win32.Allaple.b
	2	Virus.Win32.Virut.q
	1	Backdoor.Win32.Rbot.bni

Table 5.5: Labelling inconsistencies among vendors

AV information was used until now only to discriminate between detected and undetected samples, but no attention was given to the label assigned to the samples by the vendors. We initially considered the possibility of using labels provided by AV vendors to group together samples likely to be associated with the same phenomenon, thus addressing the problem of polymorphism. A more in depth analysis of the AV labels revealed a set of inconsistencies and imprecisions that led us to abandon the idea. Two different kinds of inconsistencies can be identified: inconsistencies in the name given to each sample by the same vendor, and inconsistencies in the grouping resulting from the usage of the labels.

Looking at successive reports for the same sample, one would expect the label given by a vendor to remain constant. We detected instead a considerable amount of variations in the label assigned by a vendor to the same malware sample. In total, we have been able to observe 10314 modifications to the name given to a sample by a given vendor. We identified 1081 different modifications often applied to groups of samples. Many of these modifications consist in a better specification of the name. For instance, 625 MD5s initially classified by a vendor as "Suspicious file" have been later classified as "Win32.Allaple.b". Other modifications instead involve names associated with completely different behavior, and thus underline a labelling error made by the vendor when generating the signature. For instance, 25 MD5s have been classified by the same vendor as "suspicious", as "Allaple.gen3", as "Virus.Win32.Virut.n" and as "W32/Virut.BF" in different days. An exhaustive analysis of all these cases is left for future work, but this brief overview clearly shows the labelling problems inherent in AV signatures.

Labels are assigned by each vendor without any shared agreement on the

naming conventions. It is thus not easy to compare labels assigned by different vendors. But even if the label string differs, one would expect the grouping of samples performed by two vendors to be consistent. Grouping the samples in sets according to their label, we have compared the groups generated by two different vendors on a set of 848 samples analyzed on the same day by VirusTotal. We have detected 6 inconsistencies in the resulting grouping as represented in Table 5.5, confirming previous results obtained by Bailey et al. in [Bailey 2007]. The table shows how the different elements of a set defined by the first vendor are mapped on the sets defined by the second one.

Summarizing, our experience with the output of different AV engines on the SGNET malware collection underlines some interesting challenges. Firstly, while the lack of a common naming convention among different vendors is a commonly known fact, the labelling problem goes much beyond the simple assignment of a common name. Our results seem to suggest that the main focus of the AV community consists in the detection of malware and not on the rigorous classification of the different variants, as also pointed out in [Bailey 2007]. Also, the proliferation of packers and of different forms of polymorphism is indeed a challenge for AV vendors. Signature-based approaches seem to be insufficient to handle the increase of diversity generated by these techniques, and justify the adoption of behavior-based approaches. While these approaches are present in most of current AV implementations, their results do not appear in VirusTotal reports.

5.2 Semantic lattices

Until now, we have separately examined different perspectives of the SGNET dataset. We examined the distribution of the traversals and their nature taking advantage of signature-based approaches, and we examined the content of the SGNET malware collection from the behavioral point of view and taking advantage of the output of different AV engines. We have purposely delayed the bridge between these different aspects. Here we want to propose a methodology to combine all these different perspectives into a generic overview on the relationships between the different dimensions of the epsilon-gamma-pi-mu model.

Section 5.1 showed that the definition of the coordinates in the epsilon-gamma-pi-mu is a rather challenging task.

ScriptGen's FSM traversals are an identifier of a certain activity type. That is, two activities seen by two different honeypots and sharing the same FSM traversal are very likely to be the same. The opposite inference instead does not hold. For the way FSMs are generated in SGNET, two different traversals do not necessarily correspond to two different activities. In Section 4.4 we called this phenomenon *high specificity*. In order to identify extremely localized activities, we allowed the generation of traversals generated by a single attacker or observed by a single sensor. This phenomenon is widespread in the SGNET dataset: 20% of the generated FSM traversals are associated with a single attacker and a single victim. It is highly

likely that multiple traversals with similar characteristics are associated with the same kind of activity.

The problem of the identification of malware samples was shown to be an extremely hard problem in Section 5.1.2. The usage of polymorphic techniques prevents us from identifying a malware from its content (i.e. its MD5 hash). The high level behavioral information on the other hand proved to be too simplistic: the generated groups clearly had structural differences likely to be generated by different revisions or variants of the code. The information provided by the AV vendors proved instead to be unusable in practice for classification purposes.

Little was said until now about the gamma and pi dimension and on the corresponding available information. The gamma dimension corresponds to the bogus control flow information used to redirect the victim instruction pointer towards the shellcode. We can say that SGNET collects information on a code injection attack using a "black box" approach. While host-based information is used to understand and correctly emulate the attack trace, the collected information focuses on the effects of the code injection on the network interaction. Differently from the other dimensions of the epsilon-gamma-pi-mu model, the gamma dimension is *invisible* from this perspective. If the bogus control flow data is correct, the attack is recognized as successful by the Argos-based sample factories. If instead the bogus control flow data is incorrect (e.g. it is targeting a different OS version), the instruction pointer will not be redirected towards a tainted memory area and no attack will be detected. In a simplistic approximation that will be probably refined in future work, we considered as sole representative of the gamma dimension the notion of success for a code injection attack. Considering in this context the sole categorization of the successful control flow hijacks, the gamma dimension loses significance with respect to the other three.

The previously explained challenges can all be reduced to differences in the variability of some characteristics of the attack trace. For instance, a malware family may be characterized by its MD5; another malware family may always have different MD5 but may always share the same file length; another malware family may have always random size and random MD5, but may instead be identified by the name of the process generated when the sample is executed, or by the number of generated mutexes. A similar scenario is identifiable also for the other dimensions: a set of exploits may be associated exactly to the same traversal, while another set may be associated with many highly specific traversals that share the same set of Snort alerts.

In order to handle this situation we took advantage of a simple algorithm derived from a simplification of that proposed by Klaus Julisch in [Julisch 2003]. Julisch proposed to take advantage of attribute-oriented *generalization hierarchies* to define generalizations of different fields in intrusion detection alerts. Julisch modelled each alert as a tuple composed of different attributes and associated each attribute to a generalization hierarchy. For instance, the generalization of a given IP address can be information on its nature (i.e. web server). Given a set of alerts, Julisch proposed a clustering algorithm aiming at the generation of clusters of a

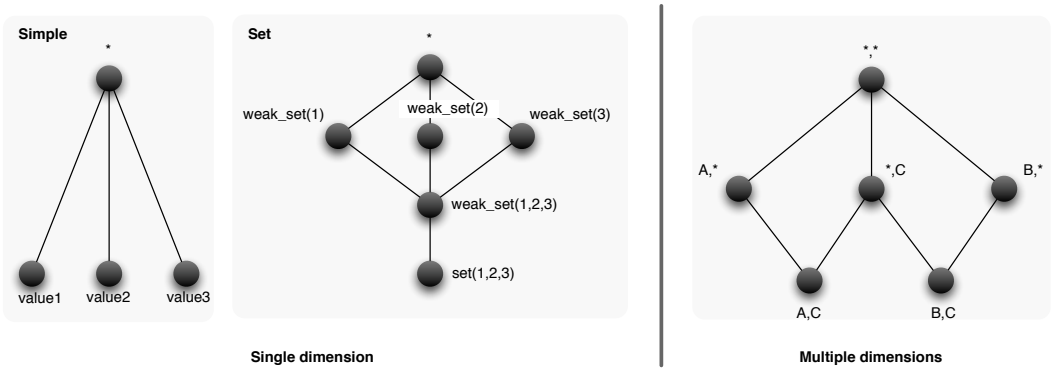

Figure 5.7: Generalizations and lattices

minimum size specified as an input parameter to the algorithm. Each cluster is built in such a way as to minimize the heterogeneity of the attributes, which is to maximize the specificity in terms of the corresponding generalization hierarchy.

The usage of the generalizations with respect to classical definition of distances also fits very well to the epsilon-gamma-pi-mu (EGPM) clustering problem. Let's consider for instance the attribute *binary length* for a group of malware samples. A single byte of difference between two samples has the same semantic value of a difference of 100 bytes: a difference in size implies a difference in structure regardless of the volume of such difference. Also, a cluster can be characterized by identical size for all the samples, but can also be characterized by the complete randomness of the size of its samples. It would be difficult to take advantage of classical hierarchical clustering approaches based on the concept of distance to model such difference.

We propose in this work a simplification of the work done by Julisch that proved to be sufficient to achieve our classification objectives. In the original work proposed in [Julisch 2003], each attribute is associated with a generalization hierarchy based on the semantic of the field. For instance, the generalization hierarchy for a port number can group port numbers according to the type of service normally associated with these ports. We simplified this concept by assigning generalization hierarchies to attributes according to their datatypes rather than their semantics. All the attributes taken into consideration have been assigned to one of these two hierarchies, graphically represented in Figure 5.7 (left):

- **Simple type.** In the case of simple scalar values (malware hash, malware file length, exploit port,...) a very simple generalization is defined. The generalization of the attribute is the node $n(*)$, which matches all the possible values for the attribute.

- **Set type.** In case of attributes associated with sets of values (for instance the set of Snort alerts generated by a traversal) we define a generalization hierar-

chy on 3 levels. The first level is what we called *weak set*. A weak set matches any proper superset of itself. For instance, the weak set $weak\_set(1,2,3)$ matches both $set(1,2,3,4)$ and $set(1,2,3,5)$. The second generalization level is generated by the expansion of a weak set into smaller weak sets composed of each of its elements. Finally, the generalization of a weak set composed of a single element is the root $n(*)$.

The use of only two different hierarchies for the totality of the attributes is a significant oversimplification of the work done in [Julisch 2003]. While a future expansion of the method to handle semantic-specific generalization hierarchies is an interesting research venue for the future, we will show here how this very simple approach was sufficient to achieve our goals.

The hierarchical structure of the generalizations makes clustering a straightforward operation when a single attribute is involved. Given a hierarchical structure L composed of k nodes $\{n_1, n_2, ..., n_k\}$ we can label each node with a frequency $\{f(n)\}$. Whenever a sample is taken into consideration, all the nodes of the structure *matching its value* will have their frequency increased by one. For instance, in the example of Figure 5.7 for the *set* datatype, the value $set(1,2,3)$ will increase the frequency of the following nodes: $n(set(1,2,3)), n(weak\_set(1,2,3)), n(weak\_set(1)), n(weak\_set(2)), n(weak\_set(3)), n(*)$.

The clustering based on generalizations is governed by two thresholds: $cluster_{thresh}$ and $relevance_{thresh}$.

The $cluster_{thresh}$ parameter defines the minimum size of a cluster. Each cluster is identified by the most specific node $n \in L$ in the set $M = \{n, f(n) \geq cluster_{thresh}\}$ and the components of the cluster will be the leaves of the corresponding subtree.

The $relevance_{thresh}$ is required to filter out the artifacts deriving from the usage of this clustering technique in an incremental way. Let's consider, for instance, an attribute representing the file size of each malware sample, modeled with a generalization hierarchy of type *simple* and using $cluster_{thresh} = 3$. An initial sample set composed by distinct values, such as $15424, 17423, 19854, 18541$ would lead to the generation of a generic cluster having as root $n(*)$. If the initial sample set is incremented by the following values: $15424, 17423, 15424, 17423$ the clustering decision will change. The nodes $n(15424)$ and $n(17423)$ will reach the threshold $cluster_{thresh}$ and will thus lead to the refinement of the initial clustering decision. The attribute, initially considered as a random attribute, is now used to identify two malware classes: one with size 15424 and one with size 17423. In general, an event will be properly classified only after having been observed at least $cluster_{thresh}$ times. That is, the algorithm has an *inertia*. The highest is the frequency of a cluster, the highest is the confidence in its classification. The $relevance_{thresh}$ parameter specifies the minimum required frequency to consider a clustering decision sufficiently reliable.

Until now, the clustering technique was presented taking into consideration a single attribute per time. Figure 5.7 (right) graphically shows the extension of the algorithm to the multidimensional case. The product of multiple generalization hierarchies generates a *lattice structure*. In such a structure every node has as

immediate relatives all the nodes generated by the generalization of each of its attributes. The size of the structure is thus exponential with the number of attributes and with the complexity of the generalization hierarchies associated with each attribute. While such complexity would be infeasible in more complex scenarios and would require heuristics such as those proposed in [Julisch 2003], it fits to the small scale of our experiment.

We considered the following attributes to be significant to generate clusters along the three dimensions epsilon, pi and mu.

Epsilon.

- **Path ID (simple).** The path identifier uniquely identifies a FSM traversal.
- **TCP port number (simple).** The port number on which the exploitation takes place.
- **Snort alert identifiers (set).** The set of Snort IDs associated with a FSM traversal.

Pi.

- **Download strategy (simple).** A binary flag indicating whether the malware sample is pushed to the victim through a TCP connection initiated by the attacker (PUSH protocol) or whether the victim is forced to actively start a download (PULL protocol).
- **Download protocol (simple).** The specific protocol used to perform the download among those supported by Nepenthes.
- **Involved TCP/UDP port (simple).** The TCP/UDP port characterizing the file transfer. In the case of PUSH protocols, this port corresponds to the port that the victim is forced to open in order to receive the malware. In the case of PULL protocols, it corresponds to the server port to which the victim connects in order to retrieve the malware.
- **MD5 hash of the shellcode binary (simple).** Included to identify the variability of the payload within the samples of a cluster.

Mu.

- **MD5 hash of the binary (simple).** Included to differentiate polymorphic samples from non-polymorphic ones.
- **Size (simple).** Size of the sample (in bytes).
- **Number of created mutexes (set).** Number of mutexes generated by the sample and by all its generated subprocesses in the 2 minutes execution time monitored by Anubis.

- **Created processes (set).** Name of the processes created by the sample during Anubis execution time.
- **Number of PE Sections (simple).** Number of separate sections (code/data) defined in the PE header.
- **Linker version (simple).** Version of the linker that generated the binary according to the Portable Executable header.
- **Packer (simple).** Name of the packer identified by the PEiD signatures (when detected).
- **Self modifying sections (set).** Set of PE section numbers marked both executable and writable. This is known to be an indicator of the presence of a packing routine, which modifies an executable section of the sample at runtime.

We applied the classification algorithm on the ordered list of code injections observed in the 8 months of data collection period using the parameters $cluster_{thresh} = 3$ and $relevance_{thresh} = 10$. Three different lattices were generated for the three dimensions, which were thus trained independently. We included in the analysis of the mu dimension only those samples that were detected as executable by Anubis, thus filtering out all the corrupted samples. In the rest of this work, we will refer to the name e-clusters, p-clusters and m-clusters to refer to the clusters of activities along the epsilon, pi and mu dimensions respectively.

5.3 Results

In the observation period, we have been able to group all the 29283 observed code injections into 19 different e-clusters, 57 different p-clusters and 73 different m-clusters. This led to the generation of 413 distinct tuples corresponding to all the observed combination of e-clusters, p-clusters and m-clusters.

Figure 5.8 provides an overview of all the observed code injections associated with the exploitation of the ASN.1 vulnerability (MS04-007) on TCP port 139. Two separate epsilon clusters (*e2* and *e3*) are associated with such an exploitation attempt. One of them is characterized by a specific FSM traversal, while the other one is composed of multiple highly specific traversals.

Interestingly, the totality of the *e2* exploits always pushes to the victim a single type of payload (*p57*). This payload is based on a download method called by Nepenthes *creceive*. In a creceive download, the attacker forces the victim to run a very small downloader that binds itself to a given TCP port. The downloader saves to the victim's hard drive and executes the binary content pushed by the attacker upon connection. The payload *p57* is characterized by the same binary content and by the same listening port for receiving the sample. This is rather surprising, and in our opinion not an optimal choice from the point of view of the propagation

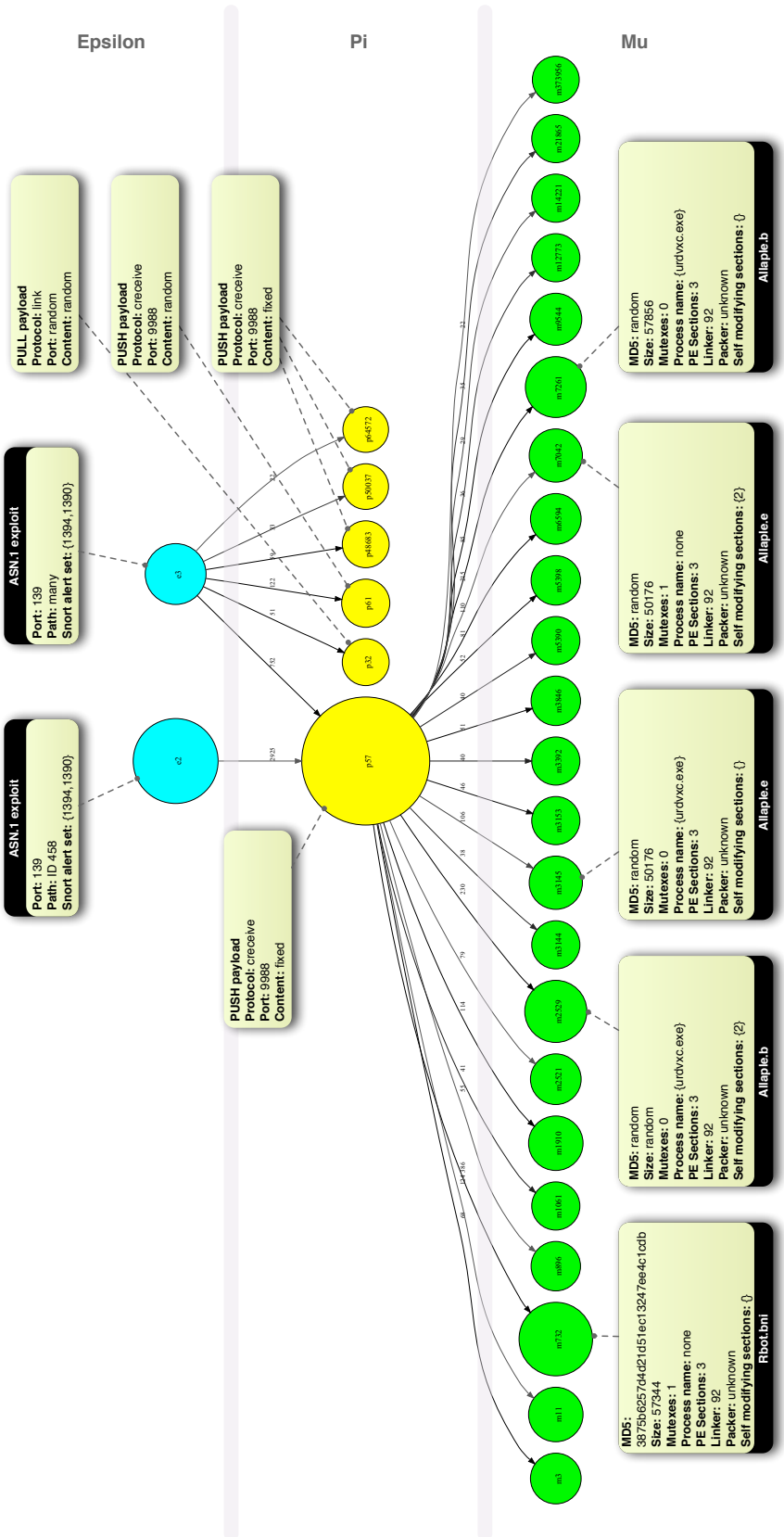

Figure 5.8: The ASN.1 exploit (port 139)

success. In order to successfully block any propagation attempt taking advantage of this exploitation technique it is sufficient to block this port at the border of the network. Being the port number a high port, it is likely to be blocked by default.

Despite its simplicity and its lack of variability, the payload *p57* is responsible for the download of a high number of m-clusters. For each of these clusters, we reported in Figure 5.8 the label associated with the malware samples by Kaspersky antivirus. The reason for the choice of this particular vendor lies in the characteristics of the provided labels. While other vendors often assign very generic labels, we found the labels assigned by this vendor enough to distinguish among different variants of the same malware family.

Four of the five predominant mu clusters are associated with the Allaple [URL 12] worm. We have seen in Section 5.1.2 the challenges associated with the identification of the different variants of this polymorphic worm. Our clustering algorithm attempted to infer automatically these groupings according to their behavioral and structural characteristics.

When comparing the information collected by SGNET with the information provided by various AV vendors on Allaple, we identified an interesting amount of contradictory information on the vulnerabilities exploited by this worm. Many reports described Allaple as propagating through the RPC DCOM vulnerability or through the MS06-040 vulnerability, and not many of them acknowledged the ability of Allaple to exploit the ASN.1 vulnerability. All the four groups shown in Figure 5.8 and labelled as Allaple variants (*m2529*,*m3145*,*m7042*,*m7261*) were seen by SGNET as propagating solely on port 139 and exploiting the ASN.1 vulnerability. This may not be the sole propagation method for this malware: different high interaction profiles may reveal in the future also different infection methods that we are not able to witness at the moment. Still, we found the information provided by the different vendors very few acknowledgements on its ability to exploit such a vulnerability.

The propagation strategy used by the malware group *m732* shown in Figure 5.8 and labelled by Kaspersky as Rbot.bni catched our attention. Figure 5.9 shows an overview of all the exploitation vectors and all the different epsilon-pi combinations that led to the download of such m-cluster. While most of the infections were witnessed through the previously analyzed ASN.1 exploit on port 139, SGNET honeypots observed a more diversified propagation strategy. This malware family was in fact also witnessed exploiting the ASN.1 exploit on port 445 and the DCOM RPC exploit on port 135. While all the ASN.1 exploits took advantage of the payload *p57* previously described, the RPC DCOM exploit took advantage of a completely different download strategy. The exploits on the *e22* cluster forced the victim to actively open a connection and download the malware from the attacker on a random port.

The payload cluster *p15* is also characterized by randomly mutating payloads. The manual analysis of the payloads showed that, after a sledge of NOOPs and 12 bytes having always same content in all the payloads, the rest of the payloads was differing every time. While we leave an in depth analysis of the payloads and on

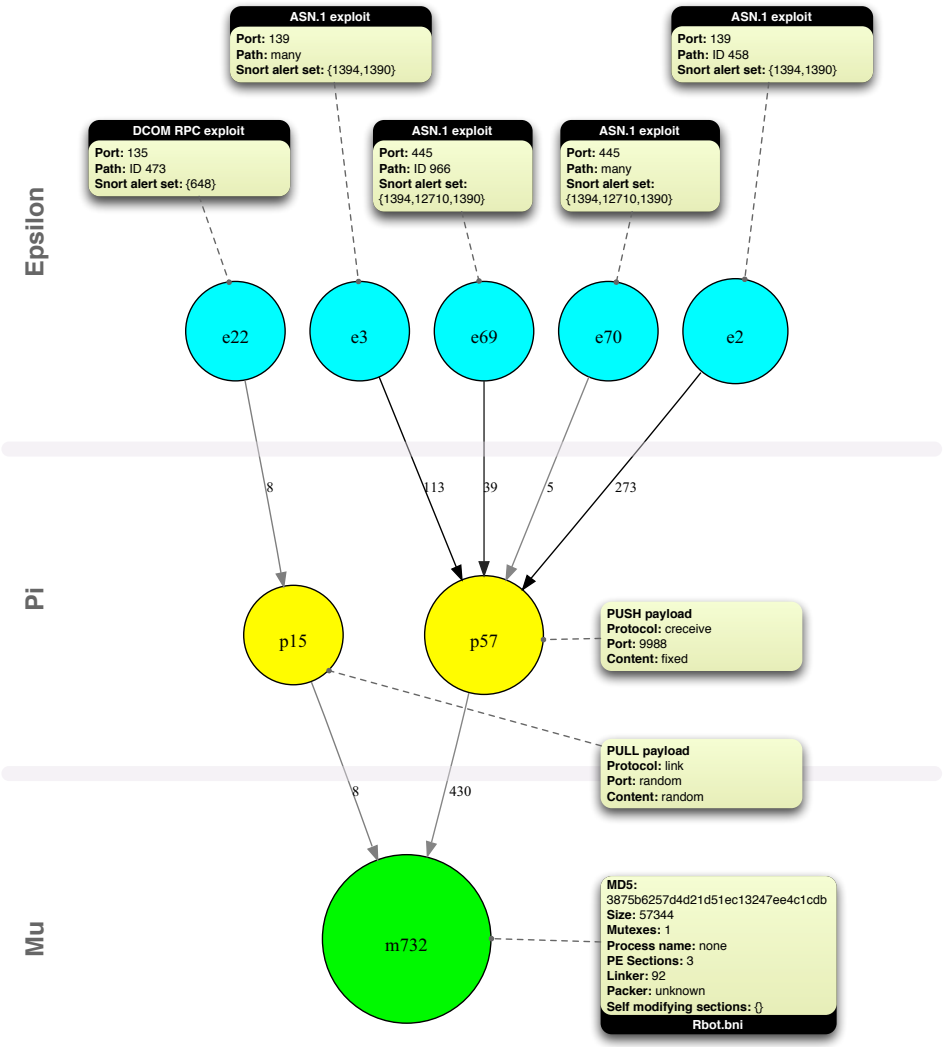

Figure 5.9: Propagation ability for mu group m732

the usage of polymorphic techniques in shellcodes as future work, this is a clear example of variability between exploitation technique, injected payload and effect of the payload on the system.

Figure 5.10 shows an overview of all the exploitation attempts associated with the epsilon cluster *e22*, previously encountered in the analysis of Rbot.bni.

Figure 5.10 reveals the presence of a high amount of p-clusters not associated with any m-cluster. While this phenomenon was less apparent in Figure 5.8 due to the low amount of diversity in the payloads, the wide amount of different payloads associated with the exploit group *e22* makes this phenomenon more visible. Many of the payloads not leading to successful downloads employ PULL strategies. Since it is the victim of the exploit the one in charge of connecting back to the attacker, we could hypothesize that such connection attempts are being blocked by some firewall on the path between the attacker and the victim. The scale of the phenomenon and the correlation between the failures of the download attempts with the protocol being used led us to the identification of a coding problem associated with our implementation of the shellcode handler. The epsilon-gamma-pi-mu classification and the results of their analysis allowed us to detect the problem, which will be fixed in future releases of the deployment. Nonetheless, its late identification affected the vision of the SGNET on the mu dimension in the considered dataset. The problem does not affect the validity of the analysis performed in this work, but shows a limitation in perspective. This limitation is assimilable to that caused by the choice of a specific high interaction host profile for the generation of the dataset taken into consideration in this work.

Differently from what we saw for the ASN.1 exploit, the same traversal is used here to push different payload types. Three different classes of payloads can be identified: a PULL payload (*p24073*) forces the download of malware from the attacker on port 2755; a PUSH payload (*p258*) forces the victim to accept the malware on a random port using the protocol *blink*; finally, there is a proliferation of clusters related to PULL payloads taking advantage of the protocol *link*. The distribution of the latter class is rather interesting. A single payload cluster (*p15*) is associated with downloads on random ports. In parallel to this, a considerable number of p-clusters is instead associated the same download strategy but on well defined ports.

Manually inspecting these p-clusters, we found out that they were all generated by a very small number of attacking IPs, repeating the attack many times against the same or another honeypot. Although each IP appears many times in each p-clusters, we often found attacks generated by a single IP in more than one of them. A possible explanation for such behavior is the following: at each execution, the malware runs the service in charge of distributing a copy of itself on a randomly chosen port and instructs its victims to download from that same port. When restarted, the port chosen by the malware changes and thus the same infected machine appears in a different pi cluster.

The variability in terms of payloads is also reflected by a variability in terms of malware variants pushed through these combinations of epsilon and pi. While

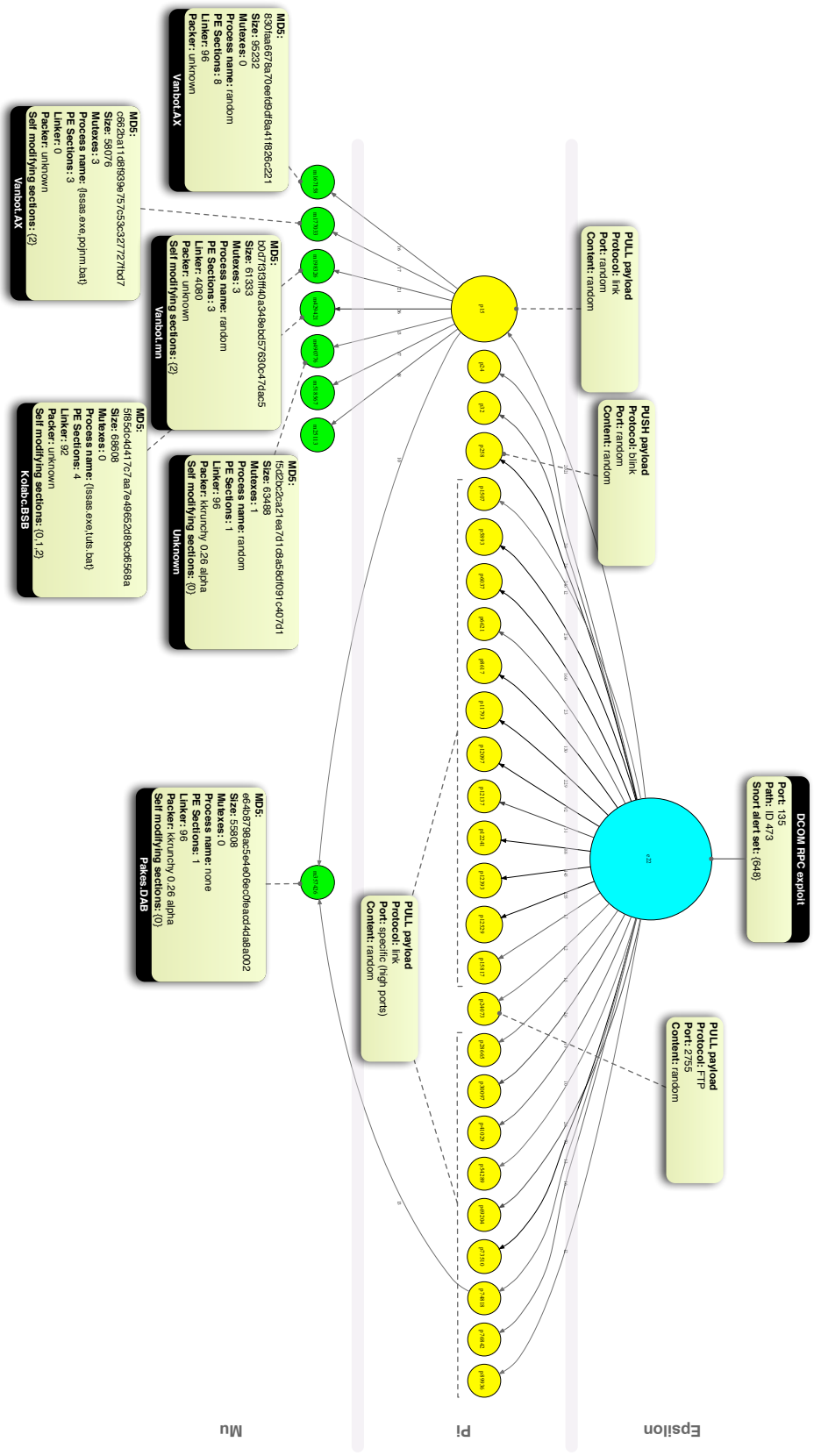

Figure 5.10: The DCOM RPC exploit

none of the mu clusters in Figure 5.10 correspond to polymorphic malware, all of them are associated with IRC-based C&C channels.

5.4 Conclusion

In this Chapter we presented a two-step process for the inference of relationships among three of the dimensions of the epsilon-gamma-pi-mu model. Firstly, we showed how we have been able to enrich the SGNET dataset with the result of different tools and web-based analysis services. Secondly, we showed a simple way to exploit this augmented dataset to cluster observed injection attacks over different dimensions of the epsilon-gamma-pi-mu space.

We add in this Chapter several lessons to those derived from the observation of ScriptGen's interaction in Chapter 4.

In Section 5.1.1 we underlined significant deviations between the observations of SGNET and the signature-based knowledge of Intrusion Detection Systems such as Snort. We have been able to identify successful code injection attacks for which no signature was present in the latest version of the VRT Sourcefire rules, showing on the one hand the limitations of knowledge-based approaches and on the other hand the need to rely on datasets as sources of intelligence on attack trends.

In Section 5.1.2 and 5.1.3 we presented two different perspectives over the collected malware samples: behavioral information and the detection performance of different commercial AV engines. All these perspectives underlined different challenges inherent to malware classification and gathering of information on their characteristics.

Finally, we introduced a simple clustering technique allowing us to combine all the different perspectives in a global view on the structure of the observed injection attacks in terms of exploits, shellcodes, and resulting malware samples. We showed how the level of correlation between these three dimensions is today extremely low. Different threats take advantage of the exploitation techniques in different ways, reusing the same exploit with very different payload types. The assumption with which we started Chapter 4 and that motivated the design choices underneath SGNET proved to hold: current Internet threats are characterized by high variability in terms of combination of exploits and payloads used by malware to propagate itself.

The results of this work open many questions and many interesting avenues for further research, and show the potential usefulness of a similar dataset in intelligence gathering on code injection attacks.

Chapter 6

Conclusion and future challenges

6.1 Challenges and future avenues

This work is centered on a data collection framework for the distributed observation of server-based code injection attacks. We have built a prototype that allowed us to collect valuable data and organize it in a centralized dataset. Through the analysis of this dataset, we have been able to give at least partial answers to many of the questions that opened this work. The analysis of the behavior and of the observations on this infrastructure has also opened a number of questions and challenges likely to open new avenues for future research.

6.1.1 Evolution of the Finite State Machines (FSM)

We have seen during the SGNET operation the continuous creation of Finite State Machine traversals, and the continuous death of them. ScriptGen's Finite State Machines are dynamic objects, which reflect the dynamic nature of the scenario witnessed by SGNET. Our analysis underlined in several occasions the necessity to monitor the FSM knowledge and detect anomalies.

We saw in Section 4.4.1, for instance, how the generation of an *incorrect* traversal can deprecate other traversals and bias the observations. We also witnessed the generation of highly specific traversals hit only for very limited periods of times and never traversed again. The evolution of the FSM knowledge needs to be continuously monitored and correlated with the trends of the observations performed by the deployment in order to cope with the dynamic nature of the threat scenario.

6.1.2 Abusing ScriptGen

We have studied in this work the ability of ScriptGen to cope with generic Internet attacks. The reader might wonder whether it would be possible for an attacker to generate specific network interactions to manipulate or disrupt the SGNET observation capabilities.

In Section 3.5.4 we have seen how the randomization of the protocol structure can push ScriptGen's learning ability to its limits and lead to excessive consumption of resources and learning time. Such "computational attacks" could be used to impact the ability of the SGNET infrastructure to correctly perform the learning by exhausting its resources.

Moreover, in Section 4.4.1 we saw that the effect of "deprecation" of traversals is generated by accidental pollutions of the sample set. An attacker could exploit

the deprecation effect to make the deployment blind to certain types of activities. It would be possible for an attacker to generate ad-hoc activities to control the learning process and generate on purpose bogus transitions preventing SGNET from correctly handling a certain class of code injection attacks and from correctly downloading the associated malware.

While these attacks are theoretically possible, we their impact on the system to be low: an automated analysis of the evolution of the FSM dynamics can easily detect and erase such abuses.

6.1.3 ScriptGen FSM traversals and vulnerabilities

The proliferation of highly specific traversals poses an interesting problem. Ideally, the generation of a new traversal is related to the observation of a *new* activity, and is thus of interest. In practice, we saw many highly localized traversals and we showed their association to a very small set of underlying vulnerabilities. How to discriminate the exploitation of a new vulnerability from a different exploit implementation addressing a known vulnerability?

Knowledge-based approaches such as Snort, used in this work to associate the traversals to set of alerts, proved to be insufficient to correctly perform the task, and left a considerable amount of observed activities unclassified.

The problem might find its solution through the usage of tainting and host-based information to infer the system service affected by the vulnerability independently from the exploit interaction.

6.1.4 Shellcode emulation

The knowledge-based approach based on Nepenthes consists in a first step in the emulation of the shellcodes collected by SGNET. While we have been able to take advantage of this solution to collect a considerable amount of malware, we have been able to identify a number of limitations to the employment of this solution, mainly associated with the usage of knowledge based signatures, that can be easily defeated by simple modifications to existing packing routines.

Recent work such as [Polychronakis 2006, Polychronakis 2007] investigated the usage of CPU emulators to unpack polymorphic shellcode. Usage of similar techniques would allow to significantly reduce the number of knowledge-based heuristics required to correctly understand the shellcode behavior.

6.1.5 Epsilon-gamma-pi-mu clustering

We proposed a very simple classification technique to identify clusters in the various dimensions of the epsilon-gamma-pi-mu model. This classification showed very interesting facts and helped us to underline the very low degree of interdependency among exploits, shellcodes and malware. We consider of extreme interest for future research the refinement and the exploitation of these techniques to gather intelligence on malware and on its propagation strategies.

6.1.6 ScriptGen and generic interaction models

This work has evaluated the usage of protocol learning techniques on a relatively simple problem, the generation of interaction models to handle deterministic exploit tools. Future work might expand the scope of this evaluation to benign protocol interaction associated to a common network.

The comparison of the generated automata with the protocol specification in case of open protocols such as HTTP or SMTP would provide a valuable benchmark for the understanding of the real limitations of this method, and might provide interaction models useful to domains such as automated protocol identification.

6.2 Conclusion

This work started by stating the necessity to generate quantitative and rich datasets on Internet attack threats. We addressed a challenge only partially solved in modern data collection techniques. The challenge derives from the combined need to deploy a large amount of distributed sensors over the Internet with that of achieving a sufficient level of sophistication to perform meaningful inferences. We proposed a protocol learning technique, called ScriptGen, explicitly addressing this challenge taking advantage of bioinformatics algorithms.

Echoing the thesis statement that opened this work, we showed that protocol learning techniques allow the automated inference of semantics out of a set of interaction samples and the generation of responders able to correctly handle future instances of the same activity. We showed how these techniques can be used to dynamically react to the appearance of previously unknown activities, incrementally refining the protocol knowledge. We have implemented our techniques in a distributed infrastructure able to take advantage of a set of sensors deployed worldwide to contribute to the generation of a common knowledge on the protocols behavior. Taking advantage of existing memory tainting techniques and of a shellcode emulator, we have enabled our infrastructure to correctly emulate code injection attacks and collect rich information on their structure.

The SGNET framework resulting from this work is associated with some characteristics that render the generated dataset of extreme interest. Firstly, the observations are based on protocol agnostic assumptions. No *a priori* knowledge on the nature of the expected observations is used, allowing the infrastructure to potentially observe also unknown exploitation attempts. Secondly, the system achieves very high scalability and is suitable for the distributed deployment of a high number of sensors. The dataset analyzed in this work is based on the observations of 23 sensors, but this number can be increased in the future in order to provide a unique perspective on the Internet network attacks. Finally, the operation of the data collection framework and its integration with external data sources offer a wide perspective over the characteristics of the various phases of the code injection attacks. We offer in this work a comprehensive analysis of the observed exploitations based on simple clustering techniques that is likely to open new avenues for

future research.

Chapitre 7

Résumé en Français

Contents

7.1 Définition du problème

La sécurité est un concept très générique qui comprend de nombreuses et différentes perspectives. L'une des principales menaces actuelles qui pèse sur la sécurité Internet est la prolifération de ce qu'on appelle "malware" ou logiciels malveillants. Le concept de ver, un logiciel qui peut se propager de façon autonome,

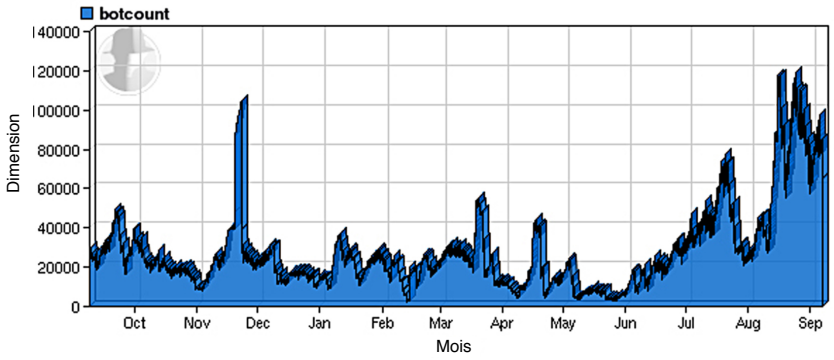

Fig. 7.1 – Evolution de la dimension des botnets

a été introduit avec le ver Morris en 1988. Les propagations de vers comme Blaster et Slammer coexistent avec la croissance de botnets dont le comportement est coordonné par un canal dit de "Commande et Contrôle" (C&C).

La communauté des chercheurs en sécurité informatique a développé un nombre considérable d'initiatives pour le suivi et l'observation de ces menaces afin de construire des défenses adéquates. Récemment, des conjectures ont été faites sur l'accroissement de l'efficacité des attaques en ce qui concerne leur capacité à compromettre des systèmes vulnérables et à accroître la taille de leurs botnets [**?**]. La Figure 1.1 montre l'évolution du nombre de bots suivis par la fondation Shadow-Server au cours de la période comprise entre Septembre 2007 et Septembre 2008. La fondation ShadowServer recherche et surveille les canaux de C&C basés sur les sites de discussion en ligne (IRC), et collecte des informations sur le nombre de participants. Ces chiffres montrent une croissance régulière de la population de botnets suivis, croissance qui a commencé en Janvier 2008 et qui suggère une amélioration des méthodes de propagation. Cette tendance va en parallèle avec l'augmentation de la sophistication des techniques utilisées par les logiciels malveillants afin d'interdire leur analyse et leur identification [Nazario 2007]. On observe également une croissance exponentielle des différentes versions des logiciels malveillants [Turner 2008].

Malheureusement, les informations obtenues après analyse de ces menaces sont souvent fondées sur des bases de données privées ou, pire, ne sont que des conjectures. Cette situation mène à un intérêt croissant pour la collecte de données pour la compréhension et l'étude des techniques de propagation utilisées par les logiciels malveillants. Comment tel ou tel malware se propage-t-il ? L'augmentation de l'efficacité ou de la sophistication de sa propagation s'accompagne-t-elle également d'un recours à des techniques d'exploitation plus complexes ? Quels types de vulnérabilités sont effectivement utilisées pour compromettre les victimes ?

Répondre à ces questions est une tâche très complexe qui requiert de s'attaquer à trois problèmes distincts : prendre en compte la diversité des applications, la diversité spatiale des attaques et parvenir à glaner des informations suffisamment riches sur les phénomènes à étudier.

7.1.1 La diversité des applications

L'Internet est aujourd'hui un système extrêmement complexe en raison de la sophistication des applications et des protocoles associés. La popularité croissante des applications Web complexes et le nombre croissant de clients pour les applications réseau, telles celles permettant de faire transiter la voix sur les réseaux IP (VoIP), ouvre un éventail de vulnérabilités aux attaquants et des moyens d'en tirer avantage. Selon [Turner 2008], 2134 vulnérabilités ont été découvertes dans la deuxième moitié de 2007, 73 % en sont considérées comme facilement utilisables à des fins malveillantes. Les applications concernées par ces problèmes varient entre applications Web, navigateurs Web, autres types de clients, des serveurs et des applications locales. Chaque classe d'applications a des caractéristiques différentes

qui peuvent être exploitées de différentes façons par les attaquants. Le processus nécessaire au suivi de ces activités est donc spécifique à chaque classe.

Ce travail aborde ce problème en s'intéressant à une catégorie spécifique d'activités : les attaques d'injection de code côté serveur. Pour cette classe, nous développons une technique pour apprendre les protocoles et gérer la diversité des applications impliquées dans ces activités. A la différence d'autres travaux dans le domaine, nous choisissons d'éviter toutes les hypothèses sur la nature des observations et nous essayons d'être agnostique en ce qui concerne la structure des protocoles.

7.1.2 Diversité spatiale

Les travaux précédents qui se sont intéressés à la collecte d'informations sur l'évolution des attaques dans l'Internet [Dacier 2004a, Cooke 2004] ont montré que les différents réseaux sont caractérisées par des profils d'attaque très différents. La production de données d'observations issus d'un seul point d'observation en un seul réseau conduit à la génération d'informations pertinentes à ce réseau spécifique, mais pas représentatives de tout l'espace des addresses IP.

Cette diversité spatiale nécessite l'extension des points d'observation à un plus grand nombre de sous-réseaux afin de couvrir autant que possible l'espace des addresses IP. Autant que possible, il est nécessaire de réduire au minimum le coût d'installation et de maintenance de chaque point de collecte de données.

7.1.3 Richesse des informations

Les informations recueillies doivent être suffisamment riches pour permettre des inférences raisonnables sur la nature et les causes des phénomènes observés. Par exemple, des informations de haut niveau sur les événements observés dans un réseau ne sont pas suffisantes pour pouvoir différencier les divers types d'activité. [Yegneswaran 2004] explique que la discrimination entre les différents types d'activités n'est possible que par le biais d'une conversation assez longue avec les attaquants.

7.2 Objectifs

Ce travail aborde les questions précédemment exposées pour une classe spécifique d'attaques : les tentatives d'injection de code côté serveur. Une technique qui s'est avérée extrêmement efficace dans la surveillance et la collecte de données pour cette classe d'attaques est l'utilisation de pots de miel. Les pots de miel sont des ressources réseau dont le seul objectif est d'être contactées par les attaquants, qui ne connaissent pas le type de réseau et donc ignorent la nature particulière des systèmes contactés. Les pots de miel, par conséquent, ont la capacité de collecter seulement le trafic suspect. Ces techniques sont largement utilisées pour la collecte des données sur les activités suspectes. Cependant, leur emploi nécessite de faire

un choix entre représentativité des données collectées d'une part et leur richesse d'autre part

> **Postulat :** Les techniques de pots de miel existantes sont incapables de collecter à la fois des données représentatives et suffisamment riches.

Les deux points principaux de ce postulat peuvent être expliqués de la façon suivante :

- Pour générer un ensemble de données représentatif des attaques présentes sur l'Internet, les points d'observation choisis pour les pots de miel doivent être les plus nombreux possibles. Cette nécessité a conduit à la mise sur pied d'installations distribuées de pots de miel comme celle du projet Leurré.com [Pouget 2006] dans lequel les plateformes de pots de miel sont installées par des volontaires intéressés à exploiter les informations recueillies.
- La nature distribuée de ces installations de pots de miel a un impact direct sur la richesse des informations recueillies. Les technologies de pots de miel nécessaires pour permettre un niveau suffisant d'interaction avec les clients sont soit trop chers par rapport à la nature de l'architecture, soit sont fondées sur des hypothèses sur la nature des attaques qui compromettent la généralité des observations.

Le travail présenté dans cette thèse offre une solution nouvelle qui permet de répondre à ces deux exigences contradictoires à l'aide d'une nouvelle technologie.

> **Hypothèse.** Les malwares qui se propagent automatiquement sont les principales causes des attaques d'injection de code observées par les pots de miel. Il est possible d'interagir de façon satisfaisante avec des attaques de ce type à l'aide de programmes déterministes qui définissent, de façon prédeterminée, les réponses à fournir aux requêtes des clients attaquants.

Ce travail se base sur cette hypothèse de départ pour générer une méthode automatique d'analyse des protocoles, appelée ScriptGen, capable d'apprendre la structure des protocoles grâce à un ensemble d'échantillons d'interaction observés sur le réseau. Elle va déduire partiellement la sémantique du protocole étudié. L'approche est agnostique par rapport à la structure du protocole : aucune hypothèse n'est faite quant à la structure des messages échangés et, a fortiori, quant à leur sémantique. Ceci permet d'éviter tout biais dans les observations produites. Les émulateurs utilisent une machine à états finis représentant les interactions connues. Cette approche permet, pour un coût faible, d'interagir avec les clients suffisament longtemps pour voir leur attaque complète. Cela permet également, grâce à ce coût faible, d'installer un grand nombre de pots de miels capables d'extraire de riches informations sur la nature des activités observées.

> **Objectif.** Dans cette thèse, nous voulons montrer que :

- L'apprentissage automatique des protocoles utilisés dans les attaques est possible, et permet la génération d'émulateurs capables de gérer correctement les futures instances des mêmes activités.
- L'apprentissage automatique peut être utilisé pour accroître progressivement les connaissances sur les interactions d'un protocole et réagir dynamiquement à de nouvelles activités.
- L'apprentissage automatique permet aux pots de miel de gérer à faible coût les activités connues et requiert l'utilisation de techniques plus coûteuses seulement pour la gestion d'activités encore inconnues. Le processus d'apprentissage donne un haut niveau d'évolutivité à un coût raisonnable en termes de complexité.
- L'augmentation du niveau d'interactivité des pots de miel peut être combinée avec des techniques de "memory tainting" et d'émulation de code pour traiter les attaques d'injection de code et télécharger des échantillons de logiciels malveillants.
- Ces techniques peuvent être utilisées pour générer une base de données riches d'informations précieuses sur les stratégies de diffusion de malware et de la durée de vie des menaces.

La résolution des points précédents nous a conduit à créer SGNET, un système distribué de pots de miel basé sur des techniques d'apprentissage automatique des protocoles. A ce jour, SGNET se compose de 23 capteurs installés dans différents réseaux en Amérique, en Europe, en Asie et en Australie. SGNET est ouvert à toutes les institutions interessées à bénéficier des informations qu'il collecte.

7.3 ScriptGen

Le concept de pot de miel a été introduit en 1995. Dans [Halme 1995] les auteurs introduisent l'idée dans le cadre de la "redirection d'intrusion" pour attirer les attaquants d'un réseau vers un autre spécialement préparé pour les étudier. Ce concept a été formalisé par L. Spitzner dans [Spitzner 2002] de la façon suivante :

' Un pot de miel est une ressource d'un système d'information dont la seule valeur réside dans le fait qu'elle soit utilisée de façon non autorisée."

Cette définition rassemble tout type de ressource réseau dont la valeur réside dans l'interaction avec les attaquants, éventuellement jusqu'au point d'être compromise par des attaques. La mise en oeuvre de ce concept dépend d'un certain nombre de variables, dont la plus importante est la nature de l'activité recherchée. Un pot de miel peut servir à observer le comportement des serveurs de réseau lorsqu'ils sont contactés par des assaillants (pot de miel côté serveur). De façon duale, un pot de miel peut essayer d'observer le comportement des applications clientes qui interagissent avec des serveurs malveillants (pot de miel côté client). Ce travail se concentre sur la première de ces deux classes.

La plupart des implémentations de pot de miel côté serveur émulent la présence des ressources du réseau sur un ensemble d'adresses IP inutilisées. Un tel

bloc d'adresses IP non utilisées est communément appelé "dark space" dans la littérature. Ces adresses peuvent appartenir soit à un réseau d'entreprise comme moyen de détection des intrusions ou des anomalies internes, soit à un réseau routable pour l'étude des attaques venant de l'Internet. L'usage d'adresses IP qui ne sont pas associées à d'autres services permet de filtrer le trafic bénin et de se concentrer sur ce qui est malveillant.En effet, mis à part les erreurs de configuration, il n'existe aucune raison de voir ces machines contactées pour de "bonnes raisons". Il faut noter que cette approche conduit à un biais dans la nature des activité observables. En effet, les attaques visant une machine spécifique, le serveur web, mail ou DNS de l'entreprise par exemple, ne seront pas vues par les pots de miel. En d'autres termes, ces approches ne permettent de voir que les attaques qui n'ont pas de connaissance préalable de la configuration du réseau qui'ls veulent attaquer.

L'état de l'art comprend un grand nombre d'implémentations différentes de ce concept. Ces implémentations diffèrent principalement par la façon dont elles émulent la présence d'un hôte sur le réseau étudié. Spitzner [Spitzner 2002] classe les différentes solutions en fonction de leur niveau d'interaction avec l'attaquant, en parlant de pot de miel à basse ou à haute interaction.

Les pots de miel de basse interaction sont généralement implémentés sous forme d'application de faible ou moyenne complexité qui imitent la présence de services réseau sans jamais offrir aux attaquants de réelle vulnérabilité dont ils puissent tirer parti. Souvent, ces pot de miel utilisent des scripts pour simuler la présence de services réseau IP tels httpd, ftpd ou encore smtpd. L'émulation d'un pot de miel à basse interaction peut être aussi simple que d'associer un socket à plusieurs portes et terminer les connections dès qu'elles sont établies.

Le niveau avec lequel un pot de miel émule des ressources peut varier considérablement selon les différentes implémentations, offrant des perspectives différentes sur les attaques observés. Chaque approche tente d'établir un compromis entre la complexité de la solution et la richesse des informations collectées.

Les pots de miel à haute interaction utilisent un système d'exploitation complet pour engager la conversation avec les attaquants. Les systèmes de virtualisation comme VMware [URL 46] ou qemu [Bellard 2005] sont des solutions simples pour l'émulation de plusieurs instances d'un système d'exploitation sur une seule ressource physique.

S'appuyant sur des systèmes d'exploitation réels, les pots de miel à haute interaction offrent le plus haut niveau possible d'interactivité avec les assaillants. Dans les pots de miel à basse interaction, la création de scripts d'emulation est souvent coûteux ou presque impossible. Dans le cas des pots de miel a haute interaction, cependant, l'émulation de protocoles n'est pas un problème. Un nouveau problème se pose cependant dans la mise en oeuvre de ces systèmes, à savoir le confinement des activités malveillantes qui peuvent y avoir lieu. En effet, par définition, ces systèmes sont vulnérables, vont être compromis et vont être utilisés par des personnes ou logiciels malveillants. Il est important de contrôler les pots de miel pour ne pas qu'ils servent, par exemple, à mener à bien des actes répréhensibles.

1	MAIL FROM : <alice@eurecom.fr
2	MAIL FROM : <bob@eurecom.fr>
3	MAIL FROM : <carl@free.fr>

TAB. 7.1 – Exemple d'entrée pour l'algorithme

La responsabilité légale du propriétaire du pot de miel se trouverait engagée dans ces méfaits, ce que personne ne souhaite. Différentes solutions existent mais ces techniques ne sont pas complètement fiables ou nécessitent un coût prohibitif au cours de la vie opérationnelle du système, notamment en termes de surveillance et de reconfiguration.

Nous proposons dans ce travail une nouvelle technique permettant de combiner les avantages de l'interaction des pots de miel à haute interaction (la richesse des informations recueillies) avec celles d'un pot de miel à basse interaction (facilité d'installation et de maintenance). ScriptGen peut observer une série d'interactions entre un client et un serveur et construire à partir de cet ensemble d'échantillons un modèle, sous forme de machine à états finis. Ce modèle peut être utilisé pour interagir ensuite avec les attaquants du même type. Nous allons voir comment ce modèle peut être utilisé avec un technique de proxy pour un apprentissage incrémental des nouvelles activités en profitant de l'interaction avec un pot de miel à haute interaction pour générer des échantillons d'activités encore inconnues.

7.3.1 Analyse par régions

Le point central de ScriptGen est l'algorithme d'analyse par régions. L'analyse par régions est responsable de la reconstruction partielle d'une structure sémantique du protocole à partir d'un ensemble d'échantillons. L'entrée de l'algorithme d'analyse par régions est constituée d'un ensemble de messages considérés comme étant sémantiquement similaires. Un exemple d'entrée est indiquée dans le tableau 7.1. Il s'agit de la commande "MAIL FROM" dans le protocole SMTP. Il est très important de comprendre que cette entrée est libre de toute sémantique et est considérée par l'algorithme comme un ensemble de séquences d'octets non structurés.

Pour reconstruire la sémantique du protocole depuis cette entrée, l'algorithme d'analyse par régions utilise des algorithmes bio-informatiques pour produire l'alignement des échantillons. Cette idée s'inspire du travail fait dans le " Protocol Informatics Projet" de Marhsall Beddoe [Beddoe 2005]. Beddoe a proposé d'utiliser ces techniques d'alignement pour simplifier les opérations d'ingénierie inverse des protocoles.

Les techniques d'alignement sont utilisées en bio-informatique pour trouver des chevauchements dans deux ou plusieurs séquences d'acides aminés afin d'identifier des gènes spécifiques. Il existe deux techniques différentes : alignement local et global. Les algorithmes d'alignement local tentent d'identifier la plus proche sous-séquence entre deux séquences, puis tentent d'identifier les similitudes entre les deux voies d'évolution. Les algorithmes d'alignement global sont utilisés pour

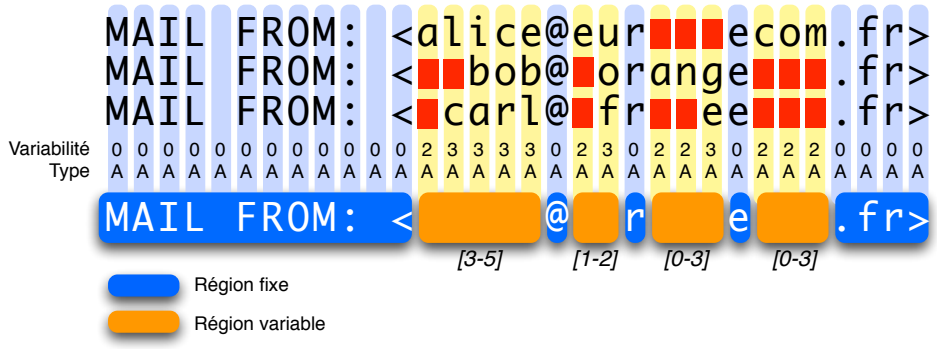

FIG. 7.2 – Synthese des regions

identifier les alignements du début à la fin d'une séquence, et sont utilisés lorsque deux séquences sont très similaires.

En bref, les algorithmes d'alignement permettent d'identifier les chevauchements entre les séquences grâce à l'introduction d'espaces. D'un point de vue du protocole, l'insertion d'espaces permet l'identification des parties invariantes du protocole même en présence de champs de longueurs variables. Par exemple, en appliquant un algorithme d'alignement à l'ensemble des séquences dans le Tableau 7.1 on obtient les résultats suivants :

```
MAIL FROM : <alice@eur___ecom.fr>
MAIL FROM : <__bob@_orange___.fr>
MAIL FROM : <_carl@_fr__ee___.fr>
```

L'algorithme d'analyse par régions utilise la production des algorithmes d'alignement pour reconstruire une structure sémantique du protocole. L'algorithme définit un ensemble de caractéristiques pour chaque octet des séquences alignées, comme c'est montré dans la Figure 7.2. Pour chaque octet, on détermine :

- **Type d'octet.** A chaque octet, nous assignons un type qui correspond au type le plus fréquent pour cet octet dans tous les échantillons. Le type, ASCII ou binaire, est déterminé sur base des valeurs les plus fréquentes pour ces deux types.
- **La variabilité de l'octet.** Pour chaque octet, nous estimons la gamme des valeurs données par les différents échantillons et nous distinguons trois catégories : octets avec valeur constante, octets dont le contenu est complètement aléatoire, et octets dont le contenu prend un nombre limité de valeurs.

Nous définissons une région comme une séquence d'octets contigues ayant les mêmes caractéristiques en termes de type et de variabilité statistique. Nous supposons, et validons par après, qu'une région est équivalente avec une bonne probabilité à une zone du protocole. Nous distinguons des régions fixes ou variables en fonction des caractéristiques de variabilité d'octets qui la composent.

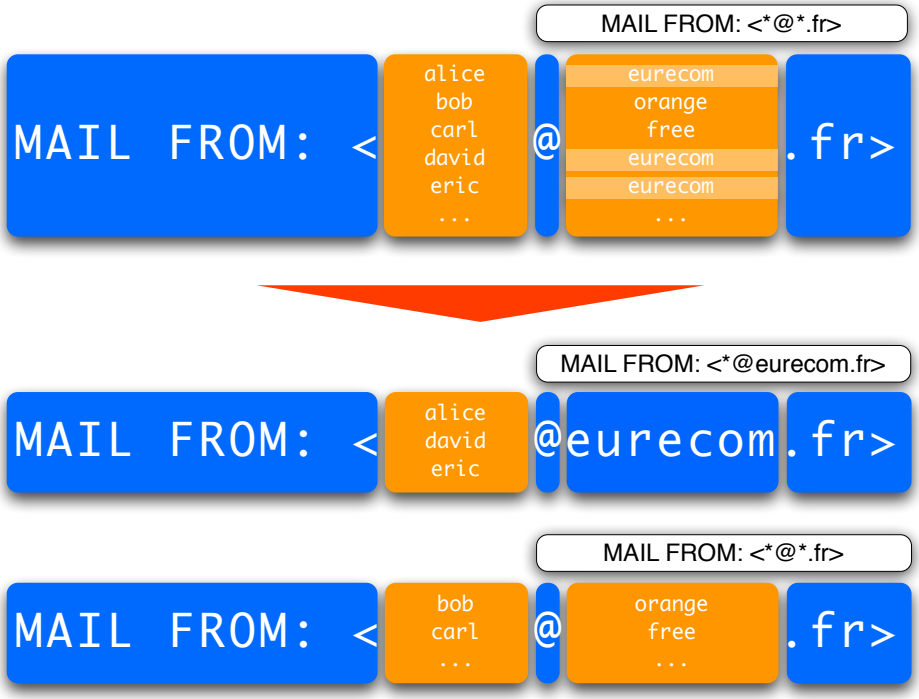

FIG. 7.3 – Micro-clustering

Le contenu des régions fixe est toujours le même dans tous les échantillons de la conversation. Ces régions sont associées à des commandes sémantiquement importantes ou à des séparateurs.

Les régions dites variables contiennent des valeurs qui, soit, sont complètement aléatoires, soit varient dans une gamme limitée de possibilités. Les premières peuvent être associées avec des estampilles ou des cookies, avec un valeur sémantique très faible. Les deuxièmes ont un valeur sémantique plus importante, qui est conservée par le biais du processus de " micro-cluster" représenté dans la Figure 7.3.

Le processus d'analyse par région exécute deux types différents de regroupement sur les échantillons d'entrée. Nous les appelons 'micro-clustering" et "macro-clustering". Le processus de "macro-clustering", exécuté en parallèle à l'alignement de séquences, regroupe les échantillons en fonction de leur contenu. Si l'entrée des séquences montre de profondes différences dans le contenu de leurs octets, ces séquences seront associées à des groupes distincts. La division en groupes est ensuite affinée grâce à l'étape de "micro-clustering". Pour toutes les régions variables associées à un ensemble limité de valeurs, le processus de "micro-clustering" regroupe ces séquences en fonction des valeurs, diverses mais en nombre limité, associées a cette région.

La synthèse des régions peut générer pour chacun de ces groupes une structure

sémantique partielle du protocole, avec une sémantique semblable à celui des expressions régulières : les invariants du protocole sont préservés, tandis que les parties aléatoires sont associées avec des valeurs de "don't care". Par exemple, nous pouvons déduire de la Figure 7.2 l'expression régulière suivante :

```
(MAIL FROM : <)([[ :alnum :]]{3,5})(@)
([[ :alnum :]]{1,2})(r)([[ :alnum :]]{0,3})(e)([[ :alnum :]]{0,3})(.fr>)
```

Nous invitons le lecteur intéressé par les détails de l'algorithme à se référer à la Section 3.1.

7.3.2 Génération de machines à états finis

L'algorithme de synthèse par régions permet la génération des abstractions sémantiques depuis une série de demandes des clients sémantiquement similaires. En pratique, cet algorithme est capable de générer automatiquement des expressions régulières en mesure de reconnaitre les instances du même type de demande.

ScriptGen représente l'interaction du protocole comme une machine à états finis représentant le langage du protocole du point de vue du serveur. Formellement, une machine à états finis ScriptGen est un tuple $(S, \Sigma_C, \Sigma_S, T, L, s, A)$ composé de

- un ensemble fini d'états (S)
- un ensemble fini appelé dictionnaire du client (Σ_C)
- un ensemble fini appelé dictionnaire du serveur (Σ_S)
- une fonction de transition ($T : S \times \Sigma_C \rightarrow S$)
- une fonction d'étiquetage ($L : S \rightarrow \Sigma_S$)
- un état initial ($s \in S$)
- un ensemble d'états finaux ($A \subset S$)

La portée de l'interaction modélisée par cette machine à états finis correspond à une connexion TCP complète : le début de la connexion correspond à l'état initial de la machine à états, la déconnexion est associée à un état final. Les transitions sont associés avec des expressions régulières générées par l'algorithme d'analyse par régions, et permettent de suivre l'évolution de l'état de la connection en fonction de la demande envoyée par le client. Chaque état est étiqueté avec la réponse à renvoyer au client. Il faut remarquer que la définition de la machine à états finis pourrait être facilement inversée pour la modélisation des interactions client, mais ceci est en dehors du contexte de ce travail.

Un certain nombre d'implémentations de pots de miel, comme Honeytank [Vanderavero 2004] et iSink [Yegneswaran 2004] choisissent de maximiser leur capacité à répondre à un grand nombre de requêtes, c'est à dire à simuler l'existence d'un grand nombre de pots de miel, en utilisant des approches sans mémoire. Ceci n'est pas notre cas puisque nous devons mémoriser, pour chaque connection, sa position dans notre machine à états. Ce choix est justifié par la nécessité de lancer un groupement sémantique des échantillons en entrée. Nous avons vu qu'une condition préalable à l'algorithme d'analyse par régions est l'utilisation de requêtes sémantiquement similaires en entrée.

ScriptGen réalise la plus grande partie du groupement sémantique grâce à la notion de contexte liée à la connexion TCP. Par exemple, dans le protocole SMTP, différentes étapes d'interaction correspondent à différentes positions au sein de la connexion. Ainsi, dans une communication normale, la commande "HELO" précède toujours la commande "BYE" et est généralement localisée au début de la connexion. Une approche sans mémoire ne nous permettrait pas de tirer parti de cette information.

Chaque nouvel échantillon en entrée est classé par l'algorithme en profitant de la machine à états finis en tant que "ossature sémantique". Chaque message de la conversation est comparé avec la machine à états finis actuel et associé a l'état correspondant. Dans le cas d'une nouvelle activité qui n'est pas encore représentée dans la machine à états finis, il y aura une demande spécifique du client qui ne correspondra pas à une transition depuis l'état courant. La demande sera alors ajoutée à la liste " bucket" de cet état. Elle devra être traitée ultérieurement par l'algorithme d'analyse par régions afin de générer une nouvelle transition. Deux types de contexte sont pris en compte pour le classement sémantique :

- **Contexte passé.** Résultat de l'interaction précédant la demande prise en compte. Il sera géré par comparaison avec les transitions de la machine à états finis existante selon le processus décrit ci-dessus.
- **Contexte futur.** La sémantique d'une requête client peut être en outre classée en fonction de ses effets sur les réponses futures du serveur. Toutes les demandes associées au " bucket" d'un état donné sont ensuite regroupées en fonction de la taille des réponses suivantes générées par le serveur dans l'échantillon d'interaction.

La répétition récursive du processus d'ajout de l'échantillon et de génération de la transition par le biais de l'algorithme d'analyse par régions permet d'affiner la machine à états finis et d'ajouter des nouvelles connaissances.

Il est important de comprendre que la qualité des inférences de l'algorithme d'analyse pour les régions dépend de la taille et de la "qualité" de l'échantillon. Dans un souci de brièveté, nous n'abordons pas ce problème ici. Il est traité en détail dans le chapitre 3.

7.4 SGNET : mise en oeuvre de ScriptGen

SGNET est la mise en oeuvre pratique de la notion de ScriptGen dans un système distribué de pots de miel. SGNET incorpore les caractéristiques de ScriptGen et les exploite pour obtenir des pots de miel qui sont :

- Agnostiques par rapport aux protocoles.
- Capables de réagir automatiquement aux nouvelles attaques (0-day attack).
- Capables d'émuler et de recueillir des informations détaillées sur le comportement des attaques, l'insertion du code shell et le chargement des échantillons de code malveillant.

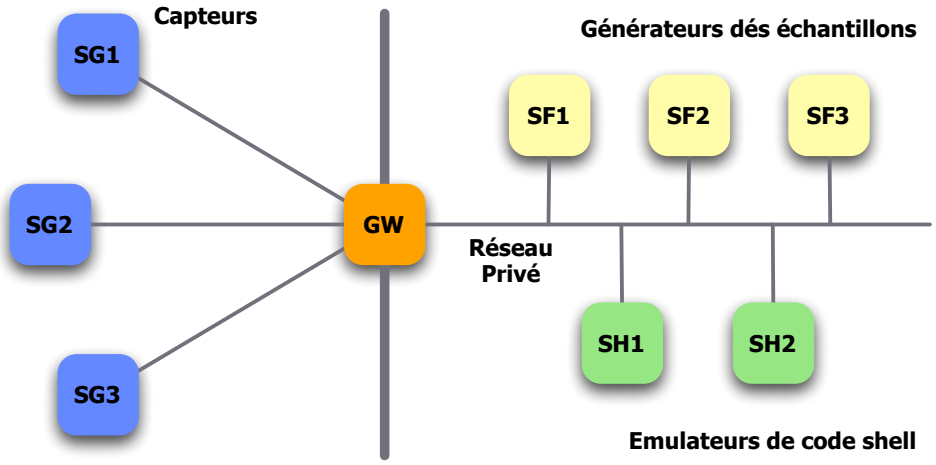

FIG. 7.4 – Architecture de SGNET

Un prototype a été mis en oeuvre et évalué au cours de ce travail. L'installation a recueilli des données pendant 8 mois en profitant d'un total de 23 installations de capteurs dans différents continents (Europe, Etats-Unis, Asie, Australie). Les informations ont été collectées dans une base de données et enrichies grâce à diverses sources de données supplémentaires.

Dans ce travail, nous considérons une attaque d'injection de code comme composé de 4 étapes, dérivées à partir d'un modèle initialement défini par Crandall et al. in [Crandall 2005].

- **Epsilon.** L'interaction de réseau nécessaire pour contrôler le flux de contrôle des applications vulnérables et le conduire vers le point où il est possible de prendre le contrôle.
- **Gamma.** Les octets utilisés par l'interaction réseau pour détourner le flux de contrôle vers un point sous le contrôle de l'attaquant.
- **Pi.** L'ensemble des octets de réseau exécutés par la victime comme conséquence du détournement du flux de contrôle.
- **Mu.** L'échantillon de code malveillant téléchargé comme conséquence de l'exécution de π.

SGNET émule l'interaction avec les clients pendant toutes ces phases par le biais de divers outils : ScriptGen pour l'émulation de l'interaction associée à l'exploit (epsilon) ; Argos [Portokalidis 2006] pour l'identification des attaques d'injection de code ; Nepenthes [Baecher 2006] pour l'identification et l'émulation du payload π et le téléchargement des logiciels malveillants μ.

7.4.1 Architecture

L'architecture de SGNET, représentée en Figure 7.4, est composée de quatre éléments.

Capteurs : les pots de miel qui, installés dans divers réseaux, interagissent avec

les clients malveillants. Les capteurs sont responsables de l'interaction réseau avec les attaquants au cours de chaque étape de l'attaque. Le trafic peut être produit localement par les capteurs ou peut être généré par les entités à distance et passé aux capteurs à travers un tunnel de paquets. Les capteurs utilisent la machine à états finis générée par ScriptGen pour interagir avec les clients pour toutes les activités "connues" par le modèle d'interaction actuel.

Générateurs d'échantillons : si un capteur reçoit une requête inconnue, il ne sera pas en mesure d'interagir correctement avec l'attaquant : cette requête n'aura pas de transition associée a l'état courant dans la machine à états finis. Dans ce cas, le capteur ne peut décider de la bonne réaction de façon indépendante. Nous proposons dans ce travail un algorithme de proxy qui permet au capteur de tirer parti d'un pot de miel de haute interaction pour continuer la conversation avec l'agresseur. Le pot de miel agit comme un proxy entre l'attaquant et un générateur d'échantillons, dérivé d'Argos [Portokalidis 2006]. Cette session fournira un nouvel échantillon de l'interaction qui peut être utilisé pour améliorer les connaissances de la machine à états finis. Les caractéristiques d'Argos nous permettent également d'obtenir des informations sur le détournement des flux de contrôle de la victime et, à partir de là, de "comprendre" le stade γ.

Emulateurs de code shell : après avoir identifié le payload π, nous utilisons des modules d'un autre logiciel, Nepenthes [Baecher 2006]. Ces modules sont capables, à l'aide d'heuristiques, de déterminer ce que fait le code shell injecté, sans pour autant l'exécuter. Grâce à cette information, Nepenthes peut imiter l'interaction associée à π. Concrètement, cela revient à télécharger le malware complet sans prendre le risque de lancer de commandes inopportunes. Encore une fois, l'interaction est passé aux capteurs via un tunnel.

Passerelle : la passerelle est l'élément central de l'architecture.Elle agit en qualité d'équilibreur de charge pour les requêtes de service émis par les capteurs vers les composants du réseau interne. En outre, le portail rassemble tous les échantillons générés par l'interaction entre les capteurs et les générateurs d'échantillons et génère des nouvelles versions de la machine à états finis en utilisant les algorithmes précédemment décrits. Quand une nouvelle version de la machine à états finis est produite, le portail met à jour la connaissance de tous les capteurs actifs.

L'interaction entre ces composants est assurée par un protocole ad-hoc inspiré du protocole HTTP, appelé Peiros. Ce protocole permet aux capteurs de générer des demandes de service aux générateurs des échantillons et aux émulateurs de code shell, et peut transférer des paquets IP bruts pour la mise en oeuvre d'algorithmes de proxy. En outre, le protocole permet à la passerelle de mettre à jour la connaissance des capteurs quand une nouvelle version de la machine à états fini a été générée. Plus de détails sur ce protocole peuvent être trouvés dans l'annexe A.

Les informations recueillies par les différentes composantes de SGNET nous aident à construire un scénario complet sur les attaques d'injection de code observées par le prototype expérimental. L'interaction avec la machine à états finis de ScriptGen permet de classer les différentes activités en fonction des chemins suivis lors de l'émulation. Le processus d'apprentissage résultant de l'interaction avec les

générateurs d'échantillons fournit des informations sur les attaques d'injection de code terminés avec succès. Pour chaque attaque d'injection de code, les émulateurs de code shell nous permettent de comprendre l'effet du payload π et de télécharger des logiciels malveillants. Toutes ces informations sont intégrées et corrélées dans une base de données. Cette base de données est enrichie par les informations générées par des outils externes. Parmi les principaux nous relevons les suivants :

- Informations sur les alertes générés par Snort ([Roesch 1999])
- Informations sur le comportement du code malveillant (Anubis [Bayer 2005])
- Informations sur la capacité des solutions antivirus commerciales à identifier le code malveillant téléchargé (VirusTotal [URL 45])
- Informations sur les entêtes PE des code malveillantes (PEfile [URL 7])

Toutes ces informations sont intégrées dans la base de données, et permettent de maximiser les informations disponibles sur chaque événement observé par l'architecture.

7.4.2 La base des données de SGNET

Dans ce travail, nous proposons une analyse des données générées par le prototype durant la période d'expérimentation de 8 mois. Cette analyse se concentre sur deux perspectives spécifiques d'intérêt à ce travail : comprendre le comportement de ScriptGen face à des attaques réelles (Section 4.4) et étudier la structure des attaques d'injection de code observées (Chapitre 5).

L'information générée par SGNET nous offre la possibilité d'étudier et valider la capacité de ScriptGen à modéliser l'interaction des protocoles dans des conditions réalistes. En particulier, cette information nous permet de répondre aux questions suivantes :

- **Est-ce que ScriptGen améliore l'évolutivité ?** En théorie, l'interaction entre les capteurs et les générateurs d'échantillons est seulement nécessaire pour gérer les activités qui ne font pas déjà partie du modèle de la machine à états finis. L'augmentation progressive de la " connaissance" des activités au cours de ce processus d'apprentissage devrait permettre de réduire la fréquence d'usage des coûteux générateurs d'échantillons. Cette capacité dépend de la diversité des activités observées, et de la rapidité avec laquelle ScriptGen est capable d'apprendre. En étudiant notre système dans des conditions réelles, nous pouvons confirmer si la pratique confirme les hypothèses théoriques.
- **Quel type d'activités ScriptGen peut il apprendre ?** Nous montrons comment les activités observées par ScriptGen appartiennent à des catégories différentes. Nous observons, en particulier, des processus d'attaque présents sur une large échelle, impliquant un nombre considérable de victimes et d'agresseurs mais nous voyons aussi un certain nombre d'activités de courte durée associées à un nombre limité de victimes et d'agresseurs. Grâce à ces données réelles, nous pouvons montrer comment ScriptGen est en mesure d'apprendre correctement les deux types d'activités, bien que, dans le dernier cas, le temps disponible pour l'apprentissage soit limité et le nombre

d'échantillons soit remarquablement réduit.

– **Comment évolue la taille de la machine à états finis ?** Il est intéressant d'étudier, grâce aux données réelles, l'évolution de la taille de la machine à états finis. Si la diversité des activités Internet était excessive, la taille de cette machine pourrait exploser et rendre la méthode inopérante. Nous montrons que, en pratique, la taille de la machine à états finis est le résultat du chevauchement d'un processus de mise en place de nouvelles branches et de "mort" d'autres. Le résultat de cette combinaison donne un nombre relativement constant de branches actives à tout moment.

En parallèle à la validation de ScriptGen, nous avons présenté dans le chapitre 5 une technique d'extraction de données pour l'étude des attaques d'injection de code dans leurs 4 étapes : epsilon, gamma, pi et mu. La nécessité de l'introduction de la technique de l'extraction de données découle de la nécessité d'identifier correctement l'invariant dans différentes dimensions.

Par exemple, l'identification correcte d'une famille de logiciels malveillants n'est pas un problème trivial. Récemment, nous avons observé un recours croissant à des techniques de polymorphisme dans les logiciels malveillants. Chaque diffusion de logiciels malveillants comme Allaple [URL 12] modifie son contenu. En prenant deux échantillons différents, déterminer leur appartenance au même type devient un problème complexe. Nous proposons donc une technique d'analyse qui étudie les différentes caractéristiques de chaque observation (dans le cas des logiciels malveillants leur structure et leur comportement lors de l'exécution) et tente d'identifier l'invariant de chaque classe.

En appliquant cette technique à chaque dimension epsilon-gamma-pi-mu d'une attaque, nous pouvons étudier les relations entre les différentes dimensions et identifier, par exemple, les logiciels malveillants qui peuvent se propager en exploitant différents exploits. De plus, on identifie un nombre de cas dans lesquels un seul type d'exploitation est utilisé pour la propagation d'un grand nombre de différents types de logiciels malveillants, suggérant la réutilisation de code entre les différents développeurs de logiciels malveillants.

7.5 Conclusion

Ce travail trouve son origine dans la nécessité de disposer de bases de données quantitatives et riches sur les menaces d'Internet. Nous avons considéré un problème qui n'était que partiellement résolu par les techniques de collecte de données appartenant à l'état de l'art. Ce problème est associé au besoin conjoint d'installer un grand nombre de capteurs en différents réseaux de l'Internet et d'obtenir un niveau de sophistication suffisant pour générer des déductions significatives sur l'origine des menaces. Nous avons proposé une technique d'apprentissage automatique de protocoles, appelée ScriptGen, en utilisant des algorithmes de bio-informatique pour trouver une solution à ce problème.

Nous avons montré que que l'apprentissage des techniques de protocole per-

met de déduire automatiquement la sémantique à partir d'un ensemble des échantillons d'interaction et de générer des émulateurs capables de gérer correctement les instances futures d'une même activité. Nous avons également montré comment ces techniques peuvent être utilisées pour réagir dynamiquement à l'apparition de requêtes encore inconnues et d'affiner, ainsi, les connaissances sur le protocole de manière incrémentale. Nous avons mis en oeuvre ces techniques dans une architecture en mesure d'exploiter une série de capteurs installés dans plusieurs réseaux de différents continents afin de contribuer à la création de modèles d'interaction. Grâce à la combinaison de plusieurs techniques existantes, nous avons bâti une infrastructure capable d'émuler correctement les attaques d'injection de code et de recueillir des informations détaillées sur leur structure.

L'infrastructure résultant de ce travail nous permet d'obtenir des informations extrêmement intéressantes sur les logiciels malveillants et sur la façon dont ils sont diffusés. Tout d'abord, les observations sont basées sur des techniques agnostiques en ce qui concerne la structure du protocole. Il n'y a pas d'hypothèse sur la structure ou la sémantique des interactions qui seront étudiées. Deuxièmement, le système démontre une grande évolutivité et est adapté à l'installation d'un grand nombre de capteurs. La base de données utilisée dans ce travail est basé sur les observations de 23 capteurs, mais ce nombre peut être augmenté considérablement à l'avenir pour générer une perspective unique sur les attaques Internet. Enfin, le fonctionnement de cet environnement pour la collecte de données et son intégration avec des sources extérieures offrent une vaste perspective sur les caractéristiques des différentes phases d'attaques par injection de code. Nous proposons dans ce travail une étude détaillée des attaques observées en tirant parti des simples techniques de regroupement qui vont ouvrir des perspectives intéressantes pour les futurs travaux de recherche.

Appendix A

The Peiros protocol

The Peiros protocol is an ad-hoc HTTP-like protocol generated as a consequence to our need to close coordination among the different SGNET entities. As described in Section 4.3.3, we can identify three main types of remote interaction among the different components of the SGNET distributed deployment.

- SGNET sensors are independent entities for most of their operation. We have although presented two cases in which a sensor in unable to continue its interaction independently and needs to rely on an external entity. The sensor needs to interact with a sample factory to handle previously unknown activities, and needs to interact with a shellcode handler whenever a code injection attack is observed. This type of interaction thus involves a sensor actively requesting a *service* to a remote entity.

- In both the previously described interactions, we need to be able to tunnel raw packets among the sensors and the involved entities. We need the tunneling to be flexible and friendly with respect to any possible firewall on the path between the two endpoints.

- The SGNET gateway needs to keep the state of the sensor up-to-date by pushing to them any update of the current FSM knowledge as soon as it is produced. Moreover, for the ease of maintenance we want to centralize the configuration of the sensors (open ports, high interaction profile associated with each IP, ...). The gateway thus needs primitives to inform the sensors of their configuration upon their appearance in the infrastructure.

We decided to integrate all these different requirements into the specification of a single distributed protocol developed ad-hoc for this specific scenario. This choice allowed us to achieve the maximum amount of freedom in the definition of the requirements without being bound to the specifications or limitations of any existing general-purpose solution. The reader might wonder if this freedom is sufficient to justify the increased overhead of developing an ad-hoc protocol with respect to reusing existing solutions. As is described in the appendix, the complexity of the Peiros protocol has been kept to a minimum, making the implementation of Peiros-enabled components extremely easy.

The Peiros interaction mimics the syntax and the structure of the HTTP protocol. Every Peiros interaction is composed of the following phases:

1. Connection establishment. During this phase, the client connects to a Peiros-enabled service and queries it about its *capabilities*. The capabilities indicate what kind of services the Peiros service is willing/enabled to offer to the client.

2. Service request. According to the capabilities discovered in the connection establishment, the Peiros client requires a given service to the server. The server will accept the service request if the capability is implemented and if it has enough resources to handle such request. If a Peiros entity implements a capability, it can in fact refuse to serve a certain client to prevent resource exhaustion.

3. Service interaction. Upon acceptance of the service request, the interaction between the client and the server is dependent on the type of capability being served.

4. Connection drop. Either party can terminate at any moment the service by sending a Peiros-level termination message. The other participant to the interaction is forced to acknowledge the termination and stop interacting. The TCP connection does not need to be terminated and can be used for further service requests.

Within this work, we have taken advantage of 4 distinct capabilities to implement all the required interaction among the different SGNET entities.

C1: sample-generation. This capability is associated with the basic operation of a sample factory, and provides to the sensors the ability to instantiates a high interaction profile with a certain network configuration and proxy raw packets to and from its network interface.

C2: shellcode-detection. This capability is complimentary to capability C1, and consists in the ability to take advantage of memory tainting techniques for the identification of successful code injection exploits and the position of the first byte of the associated shellcode. Capability C2 was separated by C1 to allow coexistance of high interaction techniques supporting memory tainting with other more scalable high interaction techniques unable to perform tainting of the control flow. While this diversification exists in the protocol specification, it is not currently implemented in the running prototype. We will thus refer in the rest of the document to the combined capability C1+C2.

C3: shellcode-handling. A Peiros-enabled service implementing this capability offers to the clients the ability to analyze and emulate the behavior of a binary shellcode sample. The client can thus provide to the server a byte sequence corresponding to the identified shellcode as well as some contextual information about its network configuration, and receive: 1) a notification of recognition of the shellcode; 2) the required network conversation to emulate the shellcode behavior taking advantage of the same tunneling capabilities offered by capability C1.

C4: update-subscriber. A client can use this capability to subscribe to a "feed" of FSM updates for a given honeypot IP address. Such subscription implicitly

allows the client to receive information on the open ports: upon reception of a FSM model for a given port, the client will bind a socket on that port and associate it to the received model. This allows the gateway to control the configuration of all the sensors and the profiles associated with each IP.

A.1 Protocol syntax

The Peiros protocol specification defines a standard set of messages. Each message is defined according to an HTTP-like syntax. Depending on the message type, the message might require an answer from the recipient of the message. The grammar of a Peiros message can be generically be described as follows:

literal ::=
(letter | digit | accepted_special) *

spaced_literal ::=
literal (‘ ’ literal) *

number ::=
digit *

letter ::=
lowercase | uppercase

lowercase ::=
‘a’...‘z’

uppercase ::=
‘A’...‘Z’

digit ::=
‘0’...‘9’

accepted_special ::=
‘#’ | ‘.’ | ‘-’ | ‘_’

newline ::=
‘\r\n’

literal_list ::=
literal (‘,’ literal) *

option ::=
literal ‘:’ (spaced_literal | literal_list)

option_list ::=
option (newline option) * newline

directive ::=
'HEAD'
| 'HELO'
| 'BYE'
| 'TRANS'
| 'ALERT'
| 'PSH'

message ::=
directive [literal_list] newline [option_list] newline data

answer ::=
number spaced_literal newline [option_list] newline data

Each Peiros message is composed of a header, containing a directive identificative of the message type and of one or more lines of options, and of a data section separated from the header by an empty line. The data section has no terminator, and its end is deduced by a "Content-Length" option in the header that is mandatory if the length of the data section is different from 0. Depending on the message type, a message can expect a mandatory anser from the other party involved in the conversation. The answer has a structure similar to that of the message, exception made for its first line, in which the directive is replaced by a numerical code following the same conventions used in the HTTP protocol.

Follows a brief overview of the different Peiros message types, the associated required options and their utilization context.

A.1.1 HEAD message

The HEAD message is used by Peiros client to query a server about its capabilities. In the HEAD options, the client declares its hostname and its client version.

```
HEAD\r\n
Name:  woody\r\n
Version:  SGNET sensor v2.0\r\n
\r\n
```

The response to a HEAD message is mandatory, and if successful provides in its options the list of capabilities implemented by the server.

```
201 Features Available\r\n
Name: woody\r\n
Version: sgProxy v1.0.0\r\n
Offers: sample-generation,shellcode-detection,shellcode-processing\r\n
\r\n
```

A.1.2 HELO message

The HELO message requires the instantiation of a given capability and, in case of positive response from the Peiros service provider, starts a Peiros session. The client is required to provide its hostname and its IP address in order to allow its identification.

```
HELO sample-generation+shellcode-detection\r\n
Name: woody\r\n
Address: 28.34.38.188\r\n
Version: ScriptGen v1.0.0\r\n
\r\n
```

The Peiros service can respond to such request with a 2xx code in case of availability to offer the required service, with a 4xx code in case of unknown capability, or with a 5xx code in case of insufficient resources to accept the request.

A.1.3 BYE message

The BYE message can be generated by any of the participants to the Peiros session as a request to terminate it. After having emitted a BYE message, the corresponding peer is not allowed any more to transmit messages inherent to the session. Upon reception of a BYE message, a peer needs to acknowledge the termination of the session by replying with a BYE message. No option is required for this message.

A.1.4 TRANS message

TRANS messages are used within Peiros sessions related to capabilities C1 and C3 to tunnel raw IP packets among the two Peiros peers. Differently from the previously analyzed messages, TRANS messages normally have a payload (the binary content of the packet being transferred) and do not require an answer. Every TRANS message must be associated with the transmission of a single packet.

```
TRANS\r\n
Content-length: 31\r\n
\r\n
32154435 46543wywetg sdgsdg swd
```

A.1.5 ALERT message

Alert messages are generated by Peiros service providers implementing capability C2 to alert a sensor about a successful code injection attack during the sample generation phase. Thanks to memory tainting techniques, sample factories implementing capability C2 are able to identify the location in the network stream of the first byte of the shellcode. This message is used to notify the sensor about its location. Optionally, the C2 service provider can provide in the message data the whole shellcode payload π identified through tainting techniques. While this shellcode identification capability is not yet supported by the current Argos-based sample factory, it is likely to be exploited in the future as an attempt to use memory tainting techniques for the identification of the whole shellcode. This should overcome the limitations of the heuristics discussed in Section 4.3.2.2 on page 4.3.2.2.

`ALERT`$\backslash r \backslash n$
`ID: 2684cd46694cf98e195042739902a34f`$\backslash r \backslash n$
`Offset:  22`$\backslash r \backslash n$
$\backslash r \backslash n$

Through the utilization of the *ID* and *Offset* attributes the service provider is able to notify the sensor about the MD5 hash of the packet containing the first byte of the shellcode as well as the offset within the packet of that first byte. It is up to the client to cache the packets and their hashes to reconstruct from this information the exact location in the network stream.

A.1.6 PSH message

PSH messages have exactly the same structure of the TRANS messages. While TRANS messages are used to exchange raw network packets over the Peiros layer, the PSH messages are used to transfer serialized representations of the FSM knowledge. In implementing the capability C4, the service provider uses these messages to asynchronously push to the subscribed client updates of the FSM knowledge.

A.1.7 ANALYZE message

Analyze messages are used to provide to a shellcode handler a sample of shellcode. The shellcode handler can respond to these requests with a `2xx` code if the shellcode was recognized correctly, with a `4xx` code if the shellcode was not recognized, or with a `5xx` code if busy. In the submission of the shellcode the client needs to provide to the shellcode handler contextual information on the network setup in which the shellcode was collected. This information is used by the heuristics for the analysis of the shellcode sample.

```
ANALYZE\r\n
Source:  1.2.3.4:1342\r\n
Destination:  192.88.33.142:445\r\n
Content-length:  22\r\n
\r\n
```

A.2 Protocol features

The Peiros protocol as presented until now has two interesting characteristics worth being detailed more in depth: its support for application level proxies as capability aggregators and its extensibility to support new capabilities.

In the definition of the Peiros protocol we have tried to be as generic as possible in the definition of each capability, decoupling the protocol interaction from the practical implementation of each functionality. The protocol definition allows an extremely interesting concept: that of capability aggregators. The SGNET architecture described in Figure 4.6 on page 79 proposes the concept of a gateway as an application level proxy between the sensors and the internal service providers. From a Peiros point of view, the gateway is a Peiros-enabled server providing to sensors a set of capabilities that is the *union* of the capabilities of the managed service providers. If the gateway manages three sample factories (capability C1+C2) and two shellcode handlers (capability C3) it will be seen by the sensor as an entity offering capabilities C1+C2+C3. Upon a request for capability C3, the gateway will forward the Peiros communication to one of the managed shellcode handlers in a completely transparent way.

The flexibility of the Peiros protocol with respect to this kind of aggregations shows its potential ability to support more complex infrastructures. For instance, multiple gateways might be deployed in different locations of the Internet and each installed sensor might choose among the gateways the one with lowest load or lowest network delay.

Moreover, the structure of the Peiros protocol is extensible to support other types of interactions among the different peers involved in SGNET. For instance, in the current prototype all the information collected by the sensors on the observed attacks is stored into local log files that are collected and parsed for the inclusion in the central database on a daily basis. Such data collection pattern is not compatible with the requirements of an early warning system, in which the reaction time to new threats needs to be in the order of minutes. In such a scenario, we could envisage the creation of a capability C5, realtime-logger, which allows the sensor to push in real time all the collected information to the central gateway.

Appendix B

Horasis: the Python API

The dataset taken into consideration in this work is offered to all the partners of the project. Every research or industrial entity interested in taking advantage of the information collected by SGNET is welcome to join the project by installing a sensor. This pattern, carried out by the Leurré.com project [Pouget 2006] and now extended to SGNET as an evolution of the initial Leurré.com deployment has attracted numerous participants from all over the world.

A time consuming task in the management of the these datasets consists in providing support to the users. Historically, the access to the these datasets was provided to interested partners either through a web interface or through direct SQL access via an SSH account. While the latter was adequate to the task of building automated analysis scripts, a set of inconvenients were identified.

- The full SQL schema of the SGNET database is rather complex. Also, it often rapidly evolves: new analysis methods are generated and new data feeds are continuously added. The impact on the external user of such complexity is noticeable, and often leads to misunderstanding of the semantics of the various concepts or of the content of the tables.

- Errors of a single user can have repercussions on the availability of the whole system. For instance, an erroneous SQL query can lead to an excessive cost on the system draining the DBMS resources.

For the above reasons, an alternate solution was investigated. This led to the generation of an API, called *horasis*, and based on python. The horasis API allows users to access most of the information stored in the database through a set of object instantiations and method calls. No knowledge is required on the underlying SQL schema: the library transparently converts all the python interaction into SQL queries used to reply to the user. The horasis library provides a unified interface to both the Leurré.com "first generation" dataset and to the additional information generated by the SGNET deployment.

All the information presented in this work was generated through python scripts exploiting the features of this API.

The horasis library provides to the user three main concepts: DB objects, iterators and predicates.

B.1 DB objects

The horasis DB objects are python objects wrapping the main concepts defined in the SGNET DB schema. Examples of these objects are, for instance, the `Malware` object or the `TinySession` object. Each of these objects is instantiated using an identifier, for instance the MD5 hash for a Malware object or the internal identifier for a TinySession object. Upon instantiation, the library queries the database to retrieve information on the existance of the instance in the dataset, and retrieve those informations available at low cost (e.g. without complex joins among tables). For instance, the `MALWARE` table in the SGNET schema contains the MD5 of each sample, its size in bytes and similar attributes. When verifying the existance of the MD5 in the table, the library can retrieve all the content of the row without significantly affecting the performance. All this information thus appears as an *attribute* of the object instance. Each object also provides a set of methods that, once invoked, generate more expensive SQL queries and eventually link to other DB objects. For instance, a malware object provides a method to retrieve the list of all the `InjectionAttack` events that led to its download.

B.2 Iterators

The horasis iterators are used to iterate in time over a collection of DB objects. For instance, the `CodeInjectionIterator` allows the iteration over all the code injection objects detected by the deployment over a certain period of time.

B.3 Predicates

Finally, the horasis library provides a set of predicates used to query the database. Two different types of predicates are provided:

- Interrogative predicates. They are used to query the database about the available knowledge on a certain event. For instance, the predicate `whois_ip` queries the database about all the available knowledge on a certain IP address (if known). These predicates return an activity identifier, an opaque identifier used within the SGNET database to identify a certain class of activities.

- Explicative predicates. They are used to retrieve additional information about an activity identifier generated by the interrogative predicates. For instance, the predicate `activity_srcnetblocks` provides the list of CIDR network prefixes containing at least an attacking source that performed a given activity class.

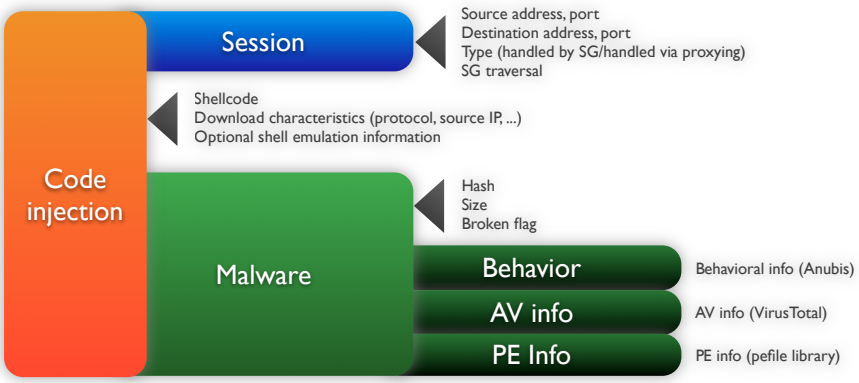

Figure B.1: Horasis library

B.4 Examples

Figure B.1 shows an example of the type of information that can be retrieved from the SGNET dataset taking advantage of the horasis library. Taking advantage of an iterator, the data consumer can have access to all the code injections observed by the infrastructure for a given timeframe. Each of these events is associated with three main objects: the code injection itself, the SG session and the malware object. Through this information, the user can retrieve detailed knowledge on the various phases of a code injection attack.

Summarizing, the horasis library allows data consumers to easily take advantage of most of the SGNET dataset without an in-depth knowledge of the underlying SQL schema.

Bibliography

[Akritidis 2005] Periklis Akritidis, Evangelos P. Markatos, Michalis Polychronakis and K. Anagnostakis. *Stride: Polymorphic sled detection through instruction sequence analysis*. In 20th IFIP International Information Security Conference, 2005. 76

[Baecher 2005] Paul Baecher, Thorsten Holz, Markus Koetter and Georg Wicherski. *Know your Enemy: Tracking Botnets*. Know Your Enemy Whitepapers, March 2005. 1, 17

[Baecher 2006] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif and Felix Freiling. *The Nepenthes Platform: An Efficient Approach to Collect Malware*. In 9th International Symposium on Recent Advances in Intrusion Detection (RAID), September 2006. 8, 60, 76, 144, 145

[Bailey 2005] Michael Bailey, Evan Cooke, Farnam Jahanian, Jose Nazario and David Watson. *The Internet Motion Sensor: A Distributed Blackhole Monitoring System*. In 12th Annual Network and Distributed System Security Symposium (NDSS), San Diego, February 2005. 19, 70

[Bailey 2007] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian and Jose Nazario. *Automated Classification and Analysis of Internet Malware*. In 10th International Symposium on Recent Advances in Intrusion Detection (RAID). Springer, 2007. 109, 117

[Bayer 2005] Ulrich Bayer, Christopher Kruegel and Engin Kirda. *TTAnalyze: A Tool for Analyzing Malware*. PhD thesis, Master's Thesis, Technical University of Vienna, 2005. 77, 107, 146

[Bayer 2006] Ulrich Bayer, Andreas Moser, Christopher Kruegel and Engin Kirda. *Dynamic Analysis of Malicious Code*. Journal in Computer Virology, vol. 2, no. 1, pages 67–77, 2006. 107

[Beddoe 2005] Marshall Beddoe. Network Protocol Analysis Using Bioinformatics Algorithms,. available on `http://www.4tphi.net/~awalters/PI`, 2005. 22, 25, 27, 95, 139

[Bellard 2005] Fabrice Bellard. *QEMU, a Fast and Portable Dynamic Translator*. In USENIX Annual Technical Conference, FREENIX Track, pages 41–46, 2005. 12, 14, 73, 138

[Caballero 2007] Juan Caballero, Heng Yin, Zhenkai Liang and Dawn Song. *Polyglot: automatic extraction of protocol message format using dynamic binary analysis*. In 14th ACM conference on Computer and Communications Security, pages 317–329. ACM New York, NY, USA, 2007. 97

[Cheswick 1990] B. Cheswick. *An Evening with Berferd in which a cracker is Lured, Endured, and Studied*. In USENIX, volume 20, page 1990, January 1990. 5

[Chinchani 2005] Ramkumar Chinchani and Eric van den Berg. *A fast static analysis approach to detect exploit code inside network flows*. In 8th International Symposium on Recent Advances in Intrusion Detection (RAID), September 2005. 76

[Chowdhary 2004] Cishal Chowdhary, Alok Tongaonkar and Tzi-cker Chiueh. *Towards Automatic Learning of Valid Services for Honeypots*. In Controls, Automation of Communication Systems (ICCACS2004), 2004. 95

[Cohen 1998] Fred Cohen. *A note on the role of deception in information protection*. Computers & Security, vol. 17, no. 6, pages 483–506, 1998. 7

[Cooke 2004] Evan Cooke, Michael Bailey, Z. Morley Mao, David Watson, Farnam Jahanian and Danny McPherson. *Toward understanding distributed blackhole placement*. In ACM workshop on Rapid malcode (WORM '04), pages 54–64, New York, NY, USA, 2004. ACM Press. 2, 17, 135

[Costa 2005] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang and Paul Barham. *Vigilante: end-to-end containment of internet worms*. In Twentieth ACM Symposium on Operating Systems Principles, pages 133–147. ACM Press New York, NY, USA, 2005. 14

[Crandall 2005] Jediah R. Crandall, S. Felix Wu and Frederic T. Chong. *Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities*. In GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA). Springer, 2005. 14, 64, 144

[Cui 2006a] Weidong Cui. *Automating Malware Detection by Inferring Intent*. PhD thesis, University of California, Berkeley, Fall 2006. 16, 95, 96

[Cui 2006b] Weidong Cui, Vern Paxson and Nicholas Weaver. *GQ: Realizing a System to Catch Worms in a Quarter Million Places*. Technical report, ICSI Tech Report TR-06-004, September 2006. 16, 95

[Cui 2006c] Weidong Cui, Vern Paxson, Nicholas Weaver and Randy H. Katz. *Protocol-Independent Adaptive Replay of Application Dialog*. In The 13th Annual Network and Distributed System Security Symposium (NDSS), February 2006. 16, 27, 67, 95

[Cui 2007] Weidong Cui, Jayanthkumar Kannan and Helen J. Wang. *Discoverer: Automatic Protocol Reverse Engineering from Network Traces*. In 16th USENIX Security Symposium, 2007. 97

[Cui 2008] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang and Luiz Irun-Briz. *Tupni: Automatic Reverse Engineering of Input Formats*. In 15th ACM conference on Computer and Communications Security, Alexandria, VA, October 2008. 97

[Dacier 2004a] Marc Dacier, Fabien Pouget and Hervé Debar. *Honeypots, a Practical Means to Validate Malicious Fault Assumptions*. In Proceedings of the 10th Pacific Rim Dependable Computing Conference (PRDC04), Tahiti, February 2004. 2, 17, 135

[Dacier 2004b] Marc Dacier, Fabien Pouget and Hervé Debar. *Towards a Better Understanding of Internet Threats to Enhance Survivability*. In Proceedings of the International Infrastructure Survivability Workshop 2004 (IISW'04), Lisbonne, Portugal, December 2004. 17

[Gu 2007] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong and Wenke Lee. *BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation*. In 16th USENIX Security Symposium, August 2007. 13, 21

[Gusfield 1997] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. ACM SIGACT News, vol. 28, no. 4, pages 41–60, 1997. 25

[Halme 1995] L.R. Halme and R.K. Bauer. *AINT misbehaving-a taxonomy of anti-intrusion techniques*. In Proceedings of the 18th National Information Systems Security Conference, pages 163–172. Baltimore, MD, USA: NIST, National Institute of Standards and Technology/National Computer Security Center, 1995. 5, 137

[Hong 2005] G. H. Hong and S. Felix Wu. *On interactive internet traffic replay*. In 8th International Symposium on Recent Advances in Intrusion Detection (RAID), LNCS, Seattle, September 2005. Springer. 69

[Jiang 2004] Xuxian Jiang, Dongyan Xu and Wang Yi-Min. *Collapsar: A VM-Based Architecture for Network Attack Detection Center*. In 13th USENIX Security Symposium, pages 15–28, 2004. 20, 94

[Julisch 2003] Klaus Julisch. *Clustering intrusion detection alarms to support root cause analysis*. ACM Transactions on Information and System Security (TISSEC), vol. 6, no. 4, pages 443–471, 2003. 11, 118, 119, 120, 121

[Leita 2005] Corrado Leita, Ken Mermoud and Marc Dacier. *ScriptGen: an automated script generation tool for honeyd*. In 21st Annual Computer Security Applications Conference, December 2005. 46, 47, 95

[Leita 2006] Corrado Leita, Marc Dacier and Frédéric Massicotte. *Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based*

honeypots. In 9th International Symposium on Recent Advances in Intrusion Detection (RAID), Sep 2006. 41, 47, 65, 71

[Leita 2008a] Corrado Leita and Marc Dacier. *SGNET: a worldwide deployable framework to support the analysis of malware threat models*. In 7th European Dependable Computing Conference (EDCC 2008), May 2008. 65, 72

[Leita 2008b] Corrado Leita and Marc Dacier. *SGNET: Implementation Insights*. In IEEE/IFIP Network Operations and Management Symposium, April 2008. 100

[Leita 2008c] Corrado Leita, Van Hau Pham, Olivier Thonnard, Eduardo Ramirez-Silva, Fabien Pouget, Engin Kirda and Marc Dacier. *The Leurre.com Project: Collecting Internet Threats Information using a Worldwide Distributed Honeynet*. In 1st Wombat Workshop, 2008. 18, 100

[Lin 2008a] Zhiqiang Lin and Xiangyu Zhang. *Deriving Input Syntactic Structure From Execution*. In 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Atlanta, GA, USA, November 2008. 97

[Lin 2008b] Zhiquiang Lin, Xuxian Jiang, Dongyan Xu and Xiangyu Zhang. *Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution*. In 15th Annual Network and Distributed System Security Symposium, San Diego, CA, February 2008. 97

[Moore 2002] David Moore. *Network Telescopes: Observing Small or Distant Security Events*. In 11th USENIX Security Symposium (Invited Talk), 2002. 16

[Moore 2006] D. Moore, C. Shannon, D.J. Brown, G.M. Voelker and S. Savage. *Inferring Internet denial-of-service activity*. In ACM Transactions on Computer Systems (TOCS), volume 24, pages 115–139. ACM Press New York, NY, USA, 2006. 16

[Nazario 2007] Jose Nazario. *Botnet Tracking: Tools, Techniques, and Lessons Learned*. In Blackhat, 2007. 1, 134

[Needleman 1970] Saul Needleman and Christian Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol. 48(3):443-53, 1970. 27

[Newsome 2006] James Newsome, David Brumley, Jason Franklin and Dawn Song. *Replayer: automatic protocol replay by binary analysis*. In 13th ACM conference on Computer and Communications Security, pages 311–321. ACM Press New York, NY, USA, 2006. 97

[Overton 2003] Martin Overton. *Worm charming: taking SMB Lure to the next level*. In Virus Bulletin Conference, 2003. 11

[Pang 2004] Pang, Yegneswaran, Barford, Paxson and Peterson. *Characteristics of Background Radiation*. In 4th ACM conference on the Internet Measurement, 2004. 16

[Polychronakis 2006] Michalis Polychronakis, Kostas G. Anagnostakis and Evangelos P. Markatos. *Network-Level Polymorphic Shellcode Detection Using Emulation*. In GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), July 2006. 76, 130

[Polychronakis 2007] Michalis Polychronakis, Kostas G. Anagnostakis and Evangelos P. Markatos. *Emulation-based Detection of Non-self-contained Polymorphic Shellcode*. In 10th International Symposium on Recent Advances in Intrusion Detection (RAID). Springer, 2007. 72, 76, 130

[Portokalidis 2006] G. Portokalidis, A. Slowinska and H. Bos. *Argos: an emulator for fingerprinting zero-day attacks*. In ACM Sigops EuroSys, 2006. 14, 73, 144, 145

[Pouget 2006] Fabien Pouget. *Distributed System of Honeypots Sensors: Discrimination and Correlative Analysis of Attack Processes*. PhD thesis, Institut Eurecom, 2006. 3, 18, 100, 136, 157

[Provos 2004] Niels Provos. *A Virtual Honeypot Framework*. In 12th USENIX Security Symposium, pages 1–14, August 2004. 7, 16

[Provos 2006] Niels Provos, Joe McClain and Ke Wang. *Search worms*. In 4th ACM Workshop on Recurring Malcode, pages 1–8. ACM Press New York, NY, USA, 2006. 96

[Ramirez-Silva 2007] Eduardo Ramirez-Silva and Marc Dacier. *Empirical study of the impact of Metasploit-related attacks in 4 years of attack traces*. In 12th Annual Asian Computing Conference focusing on computer and network security (ASIAN07), December 2007. 52, 70

[Rieck 2008] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel and Pavel Laskov. *Learning and Classification of Malware Behavior*. In GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2008. 109

[Riordan 2006] James Riordan, Diego Zamboni and Yann Duponchel. *Building and deploying Billy Goat, a Worm Detection System*. In 18th Annual FIRST Conference, 2006. 11

[Roesch 1999] M. Roesch. *Snort-Lightweight Intrusion Detection for Networks*. In USENIX LISA Systems Administration Conference, 1999. 12, 72, 100, 146

[Shannon 2004] Colleen Shannon and David Moore. *The spread of the Witty worm*. In IEEE Symposium on Security and Privacy, volume 2, pages 46–50, 2004. 16

[Small 2008] Sam Small, Joshua Mason, Fabian Monrose, Niels Provos and Adam Stubblefield. *To Catch a Predator: A Natural Language Approach for Eliciting Malicious Payloads*. In 17th Usenix Security Symposium, 2008. 96

[Smith 1981] TF Smith and MS Waterman. *Identification of common molecular subsequences*. J. Mol. Bwl, vol. 147, pages 195–197, 1981. 30

[Song 2001] Dug Song, Rob Malan and Robert Stone. *A Snapshot of Global Internet Worm Activity*. Technical report, Arbor Networks, November 2001. 16

[Spafford 1989] Eugene H. Spafford. *Crisis and aftermath*. Communications of the ACM, vol. 32, no. 6, pages 678–687, 1989. 1

[Spitzner 2002] Lance Spitzner. Honeypots: Tracking hackers. Addison-Welsey, Boston, 2002. 5, 6, 137, 138

[Staniford 2004] Stuart Staniford, David Moore, Vern Paxson and Nicholas Weaver. *The top speed of flash worms*. In ACM workshop on Rapid malcode (WORM '04), pages 33–42, New York, NY, USA, 2004. ACM Press. 17

[Tan 2004] Chew Keong Tan. *Defeating Kernel Native API Hookers by Direct Service Dispatch Table Restoration*. Technical report, SIG2 G-TEC Secure Code Study Research Paper, July 2004. 14

[Tateno 1982] Y. Tateno, M. Nei and F. Tajima. *Accuracy of estimated phylogenetic trees from molecular data*. Journal of Molecular Evolution, vol. 18, no. 6, pages 387–404, 1982. 26, 28

[The Honeynet Project 2005a] The Honeynet Project. *Know your Enemy: GenII Honeynets*. Know Your Enemy Whitepapers, May 2005. 12

[The Honeynet Project 2005b] The Honeynet Project. *Know your Enemy: Honeywall CDROM Roo*. Know Your Enemy Whitepapers, August 2005. 12, 13

[The Honeynet Project 2005c] The Honeynet Project. *Know your Enemy: Tracking Botnets - Bot Commands (`http://www.honeynet.org/papers/bots/botnet-commands.html`)*. Know Your Enemy Whitepapers, March 2005. 105

[Toth 2002] Thomas Toth and Christopher Kruegel. *Accurate buffer overflow detection via abstract payload execution*. In 5th International Symposium on Recent Advances in Intrusion Detection (RAID). Springer, 2002. 76

[Turner 2008] Dean Turner, Marc Fossi, Eric Johnson, Trevor Mack, Joseph Blackbird, Stephen Entwisle, Mo King Low, David McKinney and Candid Wueest. *Symantec Global Internet Security Threat Report*. Technical report XIII, Symantec, 2008. 1, 2, 134

[Vanderavero 2004] Nicolas Vanderavero, Xavier Brouckaert, Olivier Bonaventure and Baudouin Le Charlier. *The HoneyTank: a scalable approach to collect malicious Internet traffic*. In International Infrastructure Survivability Workshop, volume 4, 2004. 11, 42, 142

[Vanderavero 2008] Nicolas Vanderavero, Xavier Brouckaert, Olivier Bonaventure and Baudouin Le Charlier. *The HoneyTank: a scalable approach to collect malicious Internet traffic*. International Journal of Critical Infrastructures, vol. 4, no. 1/2, pages 185–205, 2008. 11

[Vrable 2005] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker and Stefan Savage. *Scalability, fidelity, and containment in the Potemkin virtual honeyfarm*. ACM SIGOPS Operating Systems Review, vol. 39, no. 5, pages 148–162, 2005. 13, 15, 16, 94

[Wang 2006] Xinran Wang, Chi-Chun Pan, Peng Liu and Sencun Zhu. *SigFree: A Signature-free Buffer Overflow Attack Blocker*. In 15th USENIX Security Symposium, 2006. 76

[Welte 2000] H. Welte. *The Netfilter framework in Linux 2.4*. In Linux Kongress, 2000. 10

[Wondracek 2008] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel and Engin Kirda. *Automatic Network Protocol Analysis*. In 15th Annual Network and Distributed System Security Symposium (NDSS'08), 2008. 97

[Yegneswaran 2004] Vinod Yegneswaran, Paul Barford and Dave Plonka. *On the Design and Use of Internet Sinks for Network Abuse Monitoring*. Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID) 2004, pages 146–165, 2004. 3, 11, 16, 42, 135, 142

[Zhuge 2007] J. Zhuge, Thorsten Holz, X. Han, C. Song and W. Zou. *Collecting Autonomous Spreading Malware Using High-Interaction Honeypots*. In Information and Communications Security, volume 7, pages 438–451. Springer, 2007. 10, 14

[Zurutuza Ortega 2007] Urko Zurutuza Ortega. *Data Mining Approaches for Analysis of Worm Activity Toward Automatic Signature Generation*. PhD thesis, Mondragon University, November 2007. 11, 106

[URL 1] *Nmap - Free Security Scanner For Network Exploration and Security Audits*, `http://www.nmap.org`. 7

[URL 2] 29A Labs. `http://vx.org.ua/29a/`. 24

[URL 3] Paul Baecher and Markus Koetter. *libemu -shellcode detection*, `http://libemu.mwcollect.org/`. 76

[URL 4] George Bakos. *Tiny Honeypot - resource consumption for the good guys, http://www.alpinista.org/thp/*. 7

[URL 5] John Bambenek. *SANS ISC diary,http://isc.sans.org/diary.html?storyid=4963&rss*. 1

[URL 6] Caida Project. *The UCSD Network Telescope, www.caida.org*, 2008. 16

[URL 7] Ero Carrera. *Pefile, http://code.google.com/p/pefile/*. 110, 146

[URL 8] CERT. *Advisory CA-2003-20 W32/Blaster worm*, August 2003. 1, 70, 105, 111

[URL 9] CWSandbox - Automated Behavior Analysis of Malware. *www.cwsandbox.org*, 2007. 77, 107

[URL 10] Frederic Deletang. *Windows SMB Nuker, http://securityvulns.com/files/smbnuke.c*. 57

[URL 11] DShield. *Distributed Intrusion Detection System, www.dshield.org*, 2007. 17

[URL 12] F-Secure. *Malware Information Pages: Allaple.A, http://www.f-secure.com/v-descs/allaplea.shtml*, December 2006. 14, 103, 105, 109, 124, 147

[URL 13] Jan Gerrit Göbel. *Amun: Python Honeypot, http://zero.ram.rwth-aachen.de/amun/*. 8, 60

[URL 14] Thorsten Holz and Frederic Raynal. *Defeating Honeypots: System Issues, Part 2, http://www.securityfocus.com/infocus/1828*, April 2005. 14

[URL 15] Honey@Home. *http://www.honeyathome.org/*. 20

[URL 16] K2. *ADMmutate: polymorphic shellcode API, http://www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz*. Talk at Defcon 9, 2001. 72

[URL 17] Kerberos. *The Network Authentication Protocol, http://web.mit.edu/kerberos/www/*. 46

[URL 18] Tom Liston. *LaBrea, http://labrea.sourceforge.net*. 7

[URL 19] Maxmind. *IP Geolocation and Online Fraud Prevention, www.maxmind.com*. 18

[URL 20] Microsoft. *Security Bulletin MS04-011, http://www.microsoft.com/technet/security/Bulletin/MS04-011.mspx*. 9, 75

[URL 21] Milw0rm. *MS Windows NetpIsRemote() Remote Overflow Exploit (MS06-040),http://www.milw0rm.com/exploits/2223*. 57

[URL 22] Milw0rm. *MS Windows XP/2K Lsasrv.dll Remote Universal Exploit (MS04-011),http://www.milw0rm.com/exploits/295*. 9, 53, 75

[URL 23] Milw0rm. *http://www.milw0rm.com/*. 24, 52

[URL 24] Mitre. *Common Vulnerabilities and Exposures, CVE-2003-0533,http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0533*. 9

[URL 25] mwcollect Alliance. *http://alliance.mwcollect.org*. 21

[URL 26] netfilter/iptables project. *http://www.netfilter.org/*. 12

[URL 27] Laurent Oudot and Thorsten Holz. *Defeating Honeypots: Network Issues, Part 2*, *http://www.securityfocus.com/infocus/1805*, October 2004. 14

[URL 28] PEiD. *Plugins*, *http://www.PEiD.info/BobSoft/Downloads.html*. 77

[URL 29] Niels Provos. *Honeyd contributions*, *http://www.honeyd.org/contrib.php*. 8

[URL 30] ShadowServer Foundation. *http://www.shadowserver.org*. 1

[URL 31] Snort signature database. *NETBIOS DCERPC NCACN-IP-TCP IActivation remoteactivation little endian overflow attempt*, *http://www.snort.org/pub-bin/sigs.cgi?sid=8690*. 105

[URL 32] Snort signature database. *WEB-IIS view source via translate header*, *http://www.snort.org/pub-bin/sigs.cgi?sid=1042*. 104

[URL 33] Snort signature database. *WEB-MISC WebDAV search access*, *http://www.snort.org/pub-bin/sigs.cgi?sid=1070*. 104

[URL 34] Snort signature database. *WEB-PHP test.php access*, *http://www.snort.org/pub-bin/sigs.cgi?sid=2152*. 104

[URL 35] Lance Spitzner. *Honeypot Farms*, *http://www.securityfocus.com/infocus/1720*, August 2003. 20, 93

[URL 36] SRI International. *Cyber-TA Research and Development Project,http://www.cyber-ta.org/releases/malware-analysis/public/*. 21

[URL 37] Strongbit technology. *EXECryptor*, *http://www.strongbit.com/execryptor.asp*. 114

[URL 38] Symantec. *W32.SQLExp.Worm,http://www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99*. 1, 16

[URL 39] Symantec. *Deepsight Threat Management System*, *http://tms.symantec.com*, 2008. 17

[URL 40] Team Cymru. *http://www.team-cymru.org/Services/darknets.html*. 16

[URL 41] The Honeynet Project. *Honeymole*, *http://www.honeynet.org.pt/index.php/HoneyMole*. 20, 94

[URL 42] The Honeynet Project. *Sebek*, *http://www.honeynet.org/tools/sebek/*. 13

[URL 43] The Honeynet Project. *http://www.honeynet.org/*. 19

[URL 44] The Metasploit Project. *www.metasploit.org*, 2007. 52, 70

[URL 45] VirusTotal. *www.virustotal.com*, 2007. 111, 146

[URL 46] VMware Inc. *The VMWare software package*, *http://www.vmware.com*, 2007. 12, 138

[URL 47] Tillman Werner. *Honeytrap - Trap attacks in your network*, *http://honeytrap.mwcollect.org/*. 10

[URL 48] M. Zalewski and W. Stearns. *Passive OS Fingerprinting Tool*. 7, 18