A lightweight framework to build honeytanks

Nicolas Vanderavero

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

December 2007

Faculté des Sciences Appliquées
Département d'Ingénierie Informatique
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Prof. Baudouin **Le Charlier** (Advisor)	UCL/INGI, Belgium
Prof. Olivier **Bonaventure**	UCL/INGI, Belgium
Prof. Marc **Dacier**	Eurecom, France
Prof. Mireille **Ducassé**	INSA/IRISA, France
Dr. Abdelaziz **Mounji**	Dexia Group, Belgium
Prof. Yves **Deville** (Chair)	UCL/INGI, Belgium

Abstract

As the Internet becomes an ubiquitous medium of communication, it carries more and more malicious activities like spam, worms or denial of service attacks. One solution to detect and collect such malicious traffic is to use honeypots. They are devices or pieces of information that are not part of the usual production system. Their goals are to lure the attackers into a trap to study them, divert their attention from another target or collect statistics.

In this work, we propose a lightweight framework to build honeytanks, which are very efficient low-interaction honeypots. We present and evaluate techniques and algorithms to simulate the presence of a large number of hosts with various degrees of realism and scalability, from a completely stateless approach to a stateful approach able, amongst other things, to mimic the behavior of various TCP/IP stacks.

Our framework is based on ASAX, a generic and lightweight data stream analyzer. We instantiate ASAX to build powerful traffic handlers. We introduce several extensions to ASAX and to RUSSEL, its programming language. These extensions allow us to develop new concurrent programming techniques to simulate hosts and protocols in a simple and modular way. We use a recently optimized version of ASAX that makes it possible to simulate tens of thousands hosts while keeping the simulation at a high level of realism.

To show the benefits of our approach, i.e., greater simplicity, flexibility, and independence of other technologies, we compare our honeytanks to Honeyd and Nepenthes, two well-known low-interaction honeypots.

A tous ceux qui m'ont aidé
A ma famille
A Karine

Remerciements

Je tiens tout d'abord à remercier mon promoteur, le professeur Baudouin Le Charlier. Durant ces six années de recherche, il a toujours été disponible pour m'aider à trouver des solutions à mes problèmes, pour me soutenir et pour m'encourager. Il a su m'écouter et me remonter le moral lorsque j'en avais besoin et s'est toujours montré patient envers moi. Il s'est investi personnellement dans cette thèse et y a sacrifié beaucoup de temps. Grâce à lui, j'ai appris qu'il est essentiel de "perdre" du temps à faire les choses correctement dès le début. Baudouin, sans toi, cette thèse n'aurait jamais pu aboutir.

Je remercie également les autres membres du jury dont les remarques et les commentaires m'ont permis d'améliorer grandement la qualité de la version finale de cette thèse. Je remercie plus particulièrement le professeur Olivier Bonaventure qui m'a proposé l'idée du honeytank, concept duquel découle toute cette thèse.

Durant six ans, j'ai été entouré de nombreux collègues et amis au département INGI qui ont fait de cette période un moment inoubliable. Merci à Renaud, Stéphane, Grégoire, Raphaël, Isabelle, François, Pierre, Nguyen, Yves et à tous les autres pour toutes les discussions et échanges si drôles, intéressants et souvent politiquement incorrects que nous avons eus. Merci également à Viviane, Chantale, Stéphanie, Fabienne et Pierre pour vos nombreux papotages et encouragements.

Lors des dernières semaines de cette aventure, j'ai trop souvent délaissé mes amis et ma famille pour me consacrer à la rédaction de cette thèse. Je les remercie pour leur patience, leur gentillesse, la confiance inébranlable qu'ils ont eue en moi et surtout pour leur soutien lors des moments difficiles que j'ai traversés.

Merci à Xavier, mon compagnon d'armes, qui a partagé avec moi les bons comme les mauvais moments de cette aventure. Nous avons commencé, traversé et terminé ensemble nos thèses respectives. Merci pour ton amitié.

Enfin, merci à Karine, ma tendre moitié, de m'avoir soutenu, encouragé et supporté durant tout ce temps. Tu as accepté sans (trop) broncher que je passe mes soirées, mes week-ends et, sur la fin, mes nuits sur cette thèse. Tu as partagé mes joies et tu as toujours su me réconforter lorsque j'en avais besoin. Karine, sans toi, je n'aurais jamais tenu le coup. *Te quiero* ♡

Contents

Part I

Introduction

Chapter 1

Introduction

> A beginning is the time for
> taking the most delicate care
> that the balances are correct.
>
> Frank Herbert
> *Dune*

1.1 Context of this Thesis

Nowadays computers from the whole world can potentially communicate with each other through the Internet. The same standard protocols are implemented by most operating systems allowing computer users to access the information from the World Wide Web in a simple way. Unfortunately, this nice picture does not completely reflect reality: Internet protocols may be flawed and/or incorrectly implemented. Computer users may have to face various efficiency and correctness problem when using Internet and the World Wide Web. More sadly, hackers may and do exploit these weaknesses for fun and/or profit.

As the Internet is difficult to tune and to master, it cannot be only a powerful tool to be transparently used by high-level users, it has to be studied for itself much like a living being. And, as living beings, it seems to suffer from various kinds of diseases due to the existence of unwanted programs known, for instance, as viruses and worms. To "internally" analyze the Internet, several special pieces of software, not aimed at normal users, have been designed and implemented. For instance, the `libpcap` library [61] contains low-level C routines allowing a programmer to capture all packets (i.e., technically, frames) passing on a network link. Such a program is independent of –and it allows one to by-pass – higher-level Internet protocols. Symmetrically, the `libnet` library [52] allows a program to "forge" arbitrary packets and to "inject" them on a network link. TCPDump [61] uses

the `libpcap` to save packets passing on a network link to a file, for further analysis, at the bit level. Based on those low-level programs, more complex applications are proposed: TCPReplay [65] and TCPopera [65] are programs able to "replay" a sequence of packets previously captured on a network, at various speeds. They can be used to check the performances of a network. NMap [71] is a so-called network scanner, which sends specific packets on a network to "discover" which hosts are accessible through the network and what are their main characteristics, such as the operating systems they are running. Scapy [3] is another library allowing a Python programmer to read and forge packets as well as analyzing them. We only cite here the few tools that we use and we analyze in this thesis. Of course, much more of them are available.

The tools we presented just before are general programs that can be used as a basis for more specific applications. In this thesis, we consider, in particular, honeypots [54], which are special programs devoted to the discovery and analysis of Internet threads such as external (human) attackers and worms. There are many different kinds of honeypots, depending on their objective, as we explain it in Chapter 3. Roughly speaking, high interaction honeypots are special programs or computers simulating or running a complete operating system, possibly containing known vulnerabilities. High interaction honeypots are closely monitored to observe and analyze the attacks they are subject to. Low interaction honeypots only partially simulate computer systems but they try to simulate a higher number of them, to collect information on a much larger scale.

1.2 Contributions of this thesis

The work we present in this thesis is at the crossroads of low-level Internet analysis tools and low interaction honeypots: we propose a methodological framework and an implemented system that allow one to conveniently build low level Internet tools and we use them to build efficient low interaction honeypots, called honeytanks, as well as traffic generators to test the honeypots. The framework and the implemented system are however tightly related.

Internet protocols can be conveniently described by finite state automata or, more realistically, by parametrized automata annotated with executable code. We use RUSSEL, an efficient programming language that allows us to describe such enhanced automata in a straightforward way. The RUSSEL language is part of the ASAX system [16, 34], which has been originally designed to allow stateful intrusion detection. ASAX is generic as it can be applied to any data stream (or event flow), and it has a two layers structure:

1. The input data stream is first converted, normally on-the-fly, to a normalized and flexible format, called NADF. In our applications, the

input stream is just a sequence of packets returned by the `libpcap` module.

2. A set of RUSSEL analysis rules are applied in non determinate to each NADF record produced by the format adapter.

It should be observed that RUSSEL is a complete programming language in which analysis rules can be viewed as a kind of parallel processes. However this theoretical power is usually not needed for –even stateful– intrusion detection since analysis rules generally are almost completely independent from each other. As a consequence, the full problem of concurrent programming in RUSSEL has not been considered in previous work on ASAX [16, 34, 32]. In this work, we improve on this work in three ways.

1. We introduce a new programming methodology to write truly communicating processes in RUSSEL. These processes can, for instance, simulate different (virtual) hosts and different TCP sessions opened on these hosts. Although possible in theory, concurrent programming with RUSSEL (as described in previous work) is not as convenient as it should be. Thus, we introduce the new notion of *frozen variable* that allows concurrent RUSSEL rules to cleanly communicate.

2. We also introduce a new way to use the format adapter, which much facilitates the writing of concurrent rules: instead of converting each packet to NADF format independently of the previous and next one, we enhance the description of each packet with information about the next one to be analyzed. This allows the rules to take decisions about the next packet before it is actually analyzed and to communicate the decisions to other rules, for the next analysis cycle, through frozen variables.

3. As a third improvement to ASAX original philosophy, we propose to use the format adapter not only as a passive routine that converts data but possibly as an active one that generates artificial NADF records playing the role of clock ticks. This allows our system to be used not only to simulate "servers" but also to simulate "clients" initiating connections.

The benefits of our approach are the following.

1. It is simple and general. The methodological framework that we propose allow us to simulate any kind of Internet protocols as well as the services from the application layer in a uniform and clear way.

2. It is efficient. The ASAX implementation that we use is especially light and efficient. We simulate thousands hosts with many sessions

opened on them in a single process without relying on any support from the operating system to handle concurrency. This makes our approach more scalable than previous ones. This claim is supported by various experiments conducted in Chapter 10 where we notably compare our implementations of honeytanks with Honeyd [47], a generic low interaction honeypot, written in C.

3. It is independent of other technologies. Most proposals for honeypots reuse some existing technologies by modifying their original use. This can be either problematic or inefficient. For instance, in its normal configuration, Honeyd forks processes to handle application services. This prevents it from simulating a large number of services. In another configuration, services can be simulated by Python modules but Python is much less efficient than ASAX as we show it in Section 7.3. As another example, Nepenthes [1] uses the TCP implementation of the OS it is running to simulate many opened ports on many computers. This is less practical than our approach: less computers can be simulated and the authors of [1] reports system crashes when too many hosts are simulated. Similar problems are reported for other systems that modify existing technologies to simulate many hosts more efficiently [69].

In this thesis, we provide various experiments to support the claims made above. A preview of these experiments is given in the next section, which summarizes the content of this thesis. The version of ASAX that we use for the experiments is an improvement to the original version, which is due to Xavier Martin [27]. Some of the objectives reached in this thesis would not have been possible without using this improved implementation as demonstrated in Chapter 10. Thus Xavier Martin's thesis and this one are distinct but complementary works. The only common contribution is the notion of frozen variable, which both facilitates concurrent programming and contributes to a more efficient implementation. Xavier Martin's thesis also contains a detailed discussion of Bro [38], a well-known intrusion detection system, which bears similarities with ASAX. In particular, our enhanced use of the format adapter must be related to Bro "event engine". We do not compare our work to Bro here since it would be redundant with Xavier Martin's presentation.

1.3 Content of this thesis

The rest of this thesis is mainly composed of two parts. The first of these two contains background material. The second one describes the original work. There is also a general conclusion as well as several appendices.

1.3.1 Background

Chapter 2 describes the ASAX system and its programming language RUSSEL as they were designed before our work. This chapter is necessary to understand our technical work.

Chapter 3 discusses related work. We first give a general overview of honeypots, the main application area of our framework, in this thesis. We also give information about existing traffic generators, the second application we investigate. Finally, we study two systems in more details. The first one is Nepenthes [1], a low interaction honeypot that "simulates only the vulnerable parts of services" to collect information about worms. Later on, in the thesis, we show how Nepenthes services can be implemented within our framework and we report on interesting traffic collected with our implementation. The second low interaction honeypot we present is honeyD. HoneyD is a configurable system with a lot of functionalities. In Chapter 10, we perform some efficiency comparisons between HoneyD and a stateful honeytank built with our approach.

1.3.2 Our work

Chapter 4 explains how we instantiate ASAX to handle network traffic. We build a format adapter on top of the `libpcap` library and we extend the RUSSEL language with new primitives to access the `libnet` library.

Chapter 5 presents the stateless honeytank. This chapter is based on a joint work with Olivier Bonaventure [67] and it takes a rather different approach than the rest of this thesis since it uses a purely stateless approach, which is a priori more efficient and is more based on a deep knowledge of Internet protocols.

Chapter 6 presents a method to allow a honeytank to successfully respond to some fingerprinting tools. This method needs maintaining stateful information in RUSSEL rules. It is used by all the honeytanks we build in this thesis except in Chapter 5.

Chapter 7 describes the concurrent programming techniques we use in the two next chapters. It introduces the notion of frozen variable. It also provides an efficiency comparison with Scapy to support our claim that our approach, as supported with the optimized version of ASAX [27], outperforms more classical approaches for the analysis of network traffic.

Chapter 8 explains how we build stateful honeytanks. Stateful information is needed to correctly implement some aspects of Internet protocols such as IP and TCP. To the contrary of usual implementations of TCP/IP as in operating systems or, to a less extent, in honeyD, we do not hide the stateful information to the application layer by just providing the data (or payload) contained in the packets. Different RUSSEL rules may take into account different layers corresponding to different protocols since all rules

have access to all the information in the original packet, in NADF format. Actual parameters of the rules are used to keep the relevant information that is needed to simulate protocols up to the desired level of realism. Thus we do not "fully implement" TCP/IP, since we do not take into account fragmentation or data segment re-sequencing but we show how this can be done. We make precise what part of the protocols are simulated in the honeytank actually implemented. In this chapter, we also explain how (stateful) Nepenthes services can be integrated to our stateful honeytank.

Chapter 9 is about traffic generators. We show that the same techniques as above can be used to program various kinds of traffic generators that can be used to check that a honeytank behaves properly. The only "trick" is that the ASAX format adapter must now be able to generate "fake" packets when "real" ones are not immediately available. This is in fact a very general technique that we use in any traffic handler to ensure that active rules are never delayed too long.

Chapter 10 provides a thorough assessment of the work described in the previous chapters. We first explain how a traffic generator can be written to test the functioning of a honeytank and we perform several such tests on our honeytanks and on Honeyd. Then we provide an efficiency comparison of two stateful honeytanks executed with both the original [32] and the optimized [27] versions of ASAX. This explains why the optimized version is actually needed and it provides the basis for a theoretical model that allows one to predict the performance of a stateful honeytank or generator in a number of situations. We then provide concrete experiments to confirm the applicability of the model. Afterward, we compare the efficiency of our stateful honeytank to Honeyd in a fair context where Honeyd is only configured to use internal services and where it does not even need to use them. We show that approach compares well with Honeyd. Lastly, we use TCPReplay to perform stress tests against our stateless and stateful honeytanks, and Honeyd. This confirms the results of all tests conducted before and, in particular, the fact that the traffic generator can be safely used to test honeypots within the limits drawn by the mathematical model.

Chapter 11 describes two experimentations where the honeytanks are used to collect real traffic. These experiments show that all versions are able to collect interesting traffic but that stateful honeytanks provide more interesting results. In particular, stateful services adapted from Nepenthes have been contacted and they have performed all steps of their simulation. On other simulated addresses, where the services are not present or where the simulation is less faithful, less information is collected.

Part II

Background

Chapter 2

ASAX

> The person who takes the banal and ordinary and illuminates it in a new way can terrify. We do not want our ideas changed. We feel threatened by such demands. "I already know the important things!" we say. Then Changer comes and throws our old ideas away.
>
> Frank Herbert
> *Chapterhouse: Dune*

In this chapter, we present ASAX, the sequential event analyzer used to implement our prototype honeytanks. We present the general architecture of ASAX and explain its operation by executing a simple example step-by-step. The ASAX system described in this chapter is the standard version of ASAX. The extensions we contributed to the ASAX system are described later in Chapter 4 and Chapter 7.

The remainder of this chapter is organized as follows. Section 2.1 gives an overview of ASAX and Section 2.2 presents its architecture. In Section 2.3 we present the functioning of the ASAX analyzer by running an example step-by-step. Section 2.4 concludes this chapter by presenting the advantages of using ASAX.

2.1 Overview of ASAX

ASAX, the **A**dvanced **S**equential **A**nalyzer on uni**X**, is a sequential event analyzer [16, 34]. It can sequentially process a flow of events and take various actions according to the events encountered. As information about the past can be maintained, it can perform a stateful analysis of the event flow. ASAX has been originally used as an intrusion detection system for

analyzing the security audit trails of BS2000 systems. However, it is not limited to this task. ASAX has been developed to be **universal**, **efficient** and **powerful**. It is universal because it can be adapted to process any type of event flow. ASAX is efficient because the analysis process is done in one pass and the implementation design uses efficient data structures and algorithms [14]. Finally, ASAX is powerful because the RUSSEL language used to write analysis rules for ASAX is a complete language.

2.2 Architecture of an ASAX system

The architecture of a typical ASAX system is depicted by figure 2.1 :

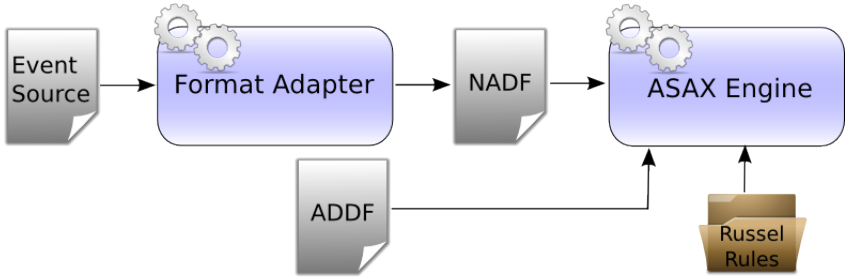

Figure 2.1: Architecture of an ASAX system

- **The format adapter**. This module fetches events from an event source and translates them into a specific format (the NADF format) understandable by the ASAX analyzer. Thus, adapting ASAX to support a new type of event source is easy : a new format adapter for this event source must be created but the ASAX analyzer is left untouched. Currently, two main format adapters have been implemented: one for the Solaris BSM format used to analyze security audit trails of Sun Solaris systems and the other for the libpcap[61] format used to capture network packets. The format adapter is not limited to simply translating events in the NADF format. It can be used as a pre-processor to apply general operations to each record or to enhance the information given to the ASAX analyzer. For example, it can number the events, add a time stamp, compute some statistics, ...

  The **N**ormalized **A**udit **D**ata **F**ormat is a simple format, able to contain any type of data. Figure 2.2 explains its structure. Each NADF record represents one event for the ASAX analyzer. A NADF record is made of fields which represent the various attributes of the event. A NADF field contains an identification number, the length of the data and the data itself.

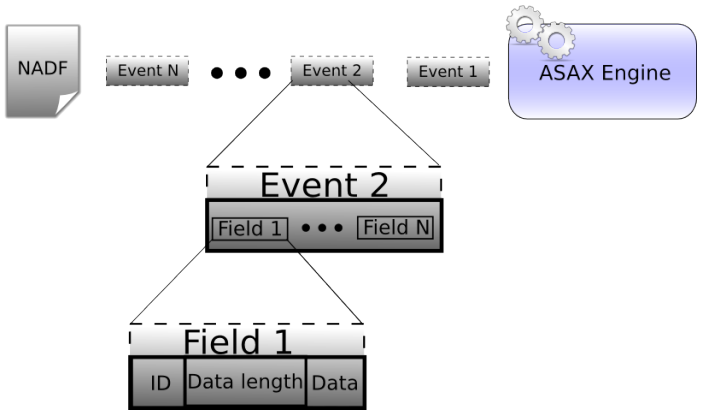

Figure 2.2: Structure of an NADF record

- **The ADDF file**. Each format adapter comes with its corresponding ADDF file. The **A**udit **D**ata **D**escription **F**ile makes the correspondence between the identification numbers used to identify NADF fields and a human-readable name, which can be used by the programmer to identify those fields inside a RUSSEL program.

- **RUSSEL files**. They contain the event analysis rules written in RUSSEL, an imperative and rule-oriented language. A RUSSEL rule can be seen as a lightweight and parametric program. A rule defines how to inspect an event, how to take decisions based on the analyzed event and how to perform actions. Since rules are parametrized, they are never directly executed: only rules **instances** are executed, i.e., rules where the formal parameters are replaced by actual parameters. A detailed explanation on how rules instances are created and executed is presented in section 2.3.

  Rules can perform various actions, e.g., increments counters, compute means, print alerts or instantiate rules. Moreover, RUSSEL supports extensions through a plug-in system which allows one to call C-written functions from a RUSSEL rule. For example, all input/operations in RUSSEL are done through external C functions. We have implemented a binding with the libnet[52] library, allowing RUSSEL rules to forge packets and inject them on a network.

- **The ASAX analyzer**. This module is the `engine` of an ASAX system: it uses NADF events as input data and processes them according to the rules defined in the RUSSEL source files.

2.3 The ASAX analyzer in action

Globally speaking, the ASAX analyzer fetches and analyzes events **one at a time**. When an event is available, all rule instances active at this moment are executed. When there is no rule instance anymore to execute for the current event, a new event is fetched and, again, all rule instances active at this moment are executed.

More specifically, the ASAX analyzer maintains three bags (or multisets) of rule instances: the `for_current`, the `for_next` and the `at_completion` bag. The `for_current` bag contains the rule instances to execute for the current event. The `for_next` bag contains the rule instances to execute for the next event. Finally, the `at_completion` bag contains the rule instances to execute at the end of the analysis.

The ASAX analyzer asks the format adapter for a new event. When it becomes available, all rule instances present in the `for_current` bag are executed. It is important to note that the execution order of the rule instances is unspecified. When a rule instance is executed, it has a read-only access to the contents of the current event. It can analyze it and perform various actions. It can also decide to create a new rule instance and place it in the `for_current`, the `for_next` or the `at_completion` bag. Once a rule instance has been executed, it is removed from the `for_current` bag. Thus, each rule instance has a lifetime limited to the current event. If a rule wants to live longer to analyze the next event, it must explicitly create a new instance of itself and place it in the `for_next` bag.

Once the `for_current` bag is empty, all rules instances present in the `for_next` bag are moved to the `for_current` bag (the `for_next` bag thus becomes empty). Then, the ASAX analyzer asks for a new event to the format adapter and the analysis process continues. The period of time in which an event is fetched, converted in NADF and analyzed is called an **ASAX cycle**. When the `for_current` and `for_next` bags are both empty or when the format adapter signals that there are no more events left, the ASAX analyzer executes the rules instances present in the `at_completion` bag and then exits.

To bootstrap the analysis process, the ASAX analyzer starts by looking for rules named `init_action`. It places an instance of them in the `for_current` bag and executes them. The `init_action` rules are rules without parameters that are used to bootstrap the analysis process by typically performing various initializations and instantiating some rules.

Let us illustrate the operation of the ASAX analyzer on a simple example. Figure 2.3 shows a simplified version of a RUSSEL code searching for all TCP `SYN` segments sent to a given IP address. The `init_action` rule initializes the global counter to zero, creates a new instance of the rule `print_found_syn` and schedules its execution at the end of the analysis. Finally, it creates a new instance of the rule `find_syn` and schedules its

```
global internal syn_count: integer;

init_action;
begin
    syn_count := 0;
    trigger off at_completion print_found_syn;
    trigger off for_next find_syn('10.0.0.1')
end;

rule find_syn(monitored_ip : string);
begin
    trigger off for_next find_syn(monitored_ip);
    if
       present tcp_flags_syn and event_destination_ip = monitored_ip
       --> syn_count := syn_count + 1
    fi;
end;

rule print_found_syn;
begin
    println('SYN found: ', syn_count)
end.
```

Figure 2.3: Simplified RUSSEL code for searching `SYN` segments

execution for the next event.

The rule `find_syn` is a parametrized rule. It starts by creating a new instance of itself with the same parameters and schedule its execution for the next event. This way, the rule will live until the end of the analysis. Then it tests if the current event representing a network packet possesses a TCP `SYN` flag and if the destination IP of the packet is the same as the IP passed as a parameter. If it is the case, the global counter `syn_count` is incremented.

The rule `print_found_syn` simply prints to the standard output a message followed by the value of the global counter `syn_count`.

As an illustration, let us explain the step-by-step execution of this example on an event trace composed of tree elements: a `RST` segment, a `SYN` segment sent to the 192.168.0.1 IP and finally a `SYN` segment sent to the 10.0.0.1 IP. The ASAX analyzer puts an instance of the `init_action` rule in the `for_current` bag and executes it. The execution of this rule creates an instance of the rule `find_syn` in the `for_next` bag and an instance of the rule `print_found_syn` in the `at_completion` bag. Once this rule is executed, it is removed from the `for_current` bag. This bag is now empty. The contents of the bag `for_next` is moved into the bag `for_current`. The next event is fetched and the rule instances in the `for_current` bag are executed.

This bag contains only one rule instance of the rule `find_syn`. Its exe-

cution creates an instance of the rule `find_syn` in the `for_next` bag. Since the current event does not have a TCP `SYN` attribute, the global counter `syn_count` is not modified. At the end of its execution, this rule instance is removed from the `for_current` bag which is now empty. It receives the rule instances moved from the `for_next` bag, a new event is fetched and its rule instances are executed.

The `for_current` bag contains only one rule instance of the rule `find_syn`. Its execution creates an instance of the rule `find_syn` in the `for_next` bag. The current event has a TCP `SYN` attribute but its destination IP does not match the IP passed as parameter to the rule instance. The global counter `syn_count` is thus not modified. This rule instance is then removed from the `for_current` bag which is now empty. It receives the rule instances moved from the `for_next` bag, a new event is fetched and its rule instances are executed.

The instance of the rule `find_syn` in the `for_current` bag is executed. It places an instance of itself in the `for_next` bag. The current event has a TCP `SYN` flag and is intended to the IP address monitored by the rule instance. Hence, the global counter `syn_count` is incremented. The rule instance is removed from the bag which is now empty. The rule instances in the `for_next` bag are moved in the `for_current` bag. A new event is fetched but the format adapter signals that the end of the audit trail has been reached. Consequently, the analyzer executes the rule instances present in the `at_completion` bag and then exits.

To make an analogy, the rule instances are like lightweight process and the ASAX analyzer is like the scheduler of an operating system choosing the process execution order. The execution order of the rule instances is undefined and the programmer should not rely on it to write his programs. Moreover, we have seen that all rule instances present in the `for_current` bag are executed at each ASAX cycle. It means that the execution time of an ASAX cycle is linear with regard to the number of rule instances. We present in section 7.2 an optimized version of ASAX for which the cycle execution time is constant under some conditions.

2.4 Advantages of using ASAX

In this chapter, we have presented ASAX, the system used to build our prototype honeytanks. We explained its working and presented its programming language, RUSSEL. Although the paradigm used by ASAX can seem unfamiliar, it has several advantages for the user. The RUSSEL programmer is freed from annoying and error-prone tasks. He does not need to explicitly handle the main loop of its program as it is done by the ASAX analyzer. He does not need to query the event source to check if an event is available, fetch the event and parse it as this is done by the format adapter.

When the RUSSEL programmer develops rules, he can directly access all fields of the event. He does not have to manipulate complex data structures in order to extract the relevant fields.

Moreover, the rule-oriented approach coupled with the sequential analysis encourages the developer to solve a large and complex problem by writing rules that focus on smaller and simpler problems and combine them together. Finally, the modularity of the ASAX system allows the developer to specify, develop, test and validate rules independently while executing them together in parallel.

Chapter 3

Related Work

> You did something because it had always been done, and the explanation was "but we've always done it this way." A million dead people can't have been wrong, can they?
>
> Terry Pratchett
> *The Fifth Elephant*

In this chapter, we describe prior and ongoing work related to honeypots and traffic generators and compare them to our approach. In this thesis, we present tools and techniques to design and implement efficient traffic handlers, i.e., tools for analyzing and replying to network traffic as well as generating it. We use them to create efficient honeypots as well as traffic generators used to assess the performance of our honeypots.

We first present the concepts and definitions related to the field of honeypots in Section 3.1. In Section 3.2 we present the criteria traditionally used to classify honeypots. The seminal works on honeypots are described in Section 3.3. Techniques for collecting traffic sent to unused IP addresses are presented in Section 3.4. We present some low-interaction and high-interaction honeypots in Section 3.6 and 3.5 and compare them to our approach. Some examples of honeytokens are presented in Section 3.7. Then, we compare our traffic generators with other tools in Section 3.8. Finally, in Sections 3.9 and 3.10, we give a detailed presentation of two honeypots: Nepenthes and Honeyd. We detail the architecture and functioning of those systems and we present an in-depth comparison of those systems with our approach.

3.1 Definitions

Simply put, a honeypot can be seen as a bait for computer attackers. It tries to lure the attackers into a trap in order to study them, divert their attention from another target, collect statistics, ... More specifically, the definition generally accepted by the community has been given by Lance Spitzner in [54, p. 40] :

> "A honeypot is a security resource whose value lies in being probed, attacked and compromised".

The author then precises that the goal of a honeypot is to be probed and attacked : a honeypot system which is never probed or attacked has little or no value.

Indeed, a honeypot is not part of the usual production systems and thus should normally never be contacted. Consequently, **every interaction with a honeypot is by definition suspicious**. In the same way, a honeypot should never initiate communications by itself. If it is the case, it probably means that the honeypot has been compromised by an attacker.

For the sake of completeness, we point another definition of a honeypot given in [9] : "A honeypot consists in an environment where vulnerabilities have been deliberately introduced in order to observe attacks and intrusions". This definition has the merit to be rigorous, for the terms attack, vulnerability and intrusion are well-defined and reused from the European MAFTIA project [62].

The broadness of the definition of a honeypot reflects the fact that, as for real hunting baits, honeypots can take many forms and use many different technologies. A honeypot is generally a computer workstation connected to the Internet. However, the definition of a honeypot does not state it has to be a computer. For example, it could be a file with a tentative name put on a production server. Any attempt to access to this file would mean that someone is going through the server. This particular case of honeypot is called a **honeytoken** [55]. A honeypot can also be a complete network of computers, possibly mimicking a real production network. This is called a **honeynet** [63]. The reader is invited to refer to [43] for a complete survey of honeypots systems.

This diversity is due to the fact that the objectives of a honeypot deeply influence its design. Clearly, the requirements for a honeypot trying to attract an insider stealing trade secrets and a honeypot trying to attract the latest Internet worm are not the same. In the same way, designing a honeypot for collecting statistics about attacks is not the same as designing a honeypot understanding and learning the techniques used by the attackers. Mainly, the goals of a honeypot can be summarized by answering the following two questions : "who has to be attracted by the honeypot ?" and "why it is important that they rise to the bait ?".

3.2 Classifications of honeypots

Honeypots are usually classified [54] according to two criteria : their high-level goal and their level of interaction. When looking at their goal, the distinction is made between *research* and *production* honeypots. When looking at the level of interaction of a honeypot, two classes of honeypots are generally considered: *low-interaction* and *high-interaction* honeypots.

High-interaction honeypots High-interaction honeypots are real systems with actual vulnerabilities. The simplest form of a high-interaction honeypot is a regular computer placed on a network and used as a *sacrificial lamb* to lure attackers. Some variants use virtualization techniques such as VMWare, User-Mode Linux or Xen to execute several operating systems on a single computer. The main advantage of high-interaction honeypots is that they offer the most reliable source of information about the attacks performed. Indeed, as they execute real operating systems, a forensics analysis can be made to understand what vulnerabilities has been exploited, what files have been altered, ... However, high-interaction honeypots have some drawbacks. First, as they consist in regular operating systems and applications, their execution uses a lot of processing and memory resources. Second, they can be actually exploited by an attacker which can then use the compromised honeytanks to launch other attacks. Consequently, high-interaction honeypots must be closely monitored and containment techniques must be used to isolate a compromised honeypot.

Low-interaction honeypots A low-interaction honeypot is a software tool that simulates a vulnerable service, a vulnerable host or even a network of vulnerable hosts. Since they are not complete operating systems running actual vulnerable applications, low-interaction honeypots are usually easier to deploy and administrate than high-interaction honeypots. As they only simulate vulnerable hosts and applications, they can normally not be compromised and thus do not require containment techniques. Moreover, they use less resource than high-interaction honeypots. A single computer running a low-interaction honeypot can simulate the presence of several hosts on a network. However, low-interaction honeypots only simulate a subset of the behavior of a real vulnerable host. Consequently, the information collected about the attacks is less precise and complete than the information collected by a high-interaction honeypot.

Research honeypots The goal of a research honeypot is to attract attackers and malicious traffic in order to study them and gain a better understanding of their behavior. They are used by the research community to understand how the attackers work, what tools they use, what vulnerabilities they exploit .... As the goal is to obtain the most precise information

about the behavior of attackers, high-interaction honeypots are usually used to implement research honeypots.

Production honeypots Learning from the attackers is not the primary goal of a production honeypot. Indeed, its goal is to mitigate the risks of a production network to be compromised. Production honeypots are typically closely monitored systems. Since no one should contact them, an interaction with a production honeypot is an anomaly and the source of the interaction can be further monitored or blacklisted, preventing the compromising of other resources in the production network. Moreover, a production honeypot can be used to divert the attention of the attackers from other real production resources. While the attacker is attacking a production honeypot, this give some times to the security officer to monitor and protect other hosts.

3.3 Seminal works on honeypots

This section presents the articles which are generally accepted as the *seminal works* in the field of honeypots.

3.3.1 Stalking the wily hacker

In *Stalking the wily hacker* [59], the author describes how the Lawrence Berkeley Laboratory (LBL) handled an intrusion in their computer system. Instead of trying to prevent further access to the intruder, they deliberately let their systems open. They monitored him, printing all his activities in order to study him and his methodology and to trace him back to the source.

The lab started to suspect an intrusion when an accounting program reported an accounting error caused by a newly created account which has no billing address. The intrusion was confirmed when they detected someone trying to modify accounting records. The paper explains thoroughly how they monitored the attacker to study his behavior.

One interesting aspect of this paper is how they managed to force the attacker to stay connected long enough to trace him. Indeed, the attacker generally did short connexions, lasting only a few minutes. But in order to trace precisely the attacker's phone connexion, the phone technicians needed a relatively long connexion. Looking at the log printouts, it was clear that the attacker was looking for military documents. So they baited him by placing several fictitious text documents describing research for the Strategic Defense Initiative. These file were readable only by their owner and by the system administrator and an alarm was set to trigger when they were accessed. The attacker eventually found those files and spent more than

an hour reading them, allowing the phone technician to identify precisely his location[1].

The use of bogus tempting files placed in a strategic place in order to lure attackers is now known as using honeytoken. This paper is the first known publication to describe the use of such honeytokens.

3.3.2 An Evening with Berferd ...

In [6], the author explain how he studied an attacker trying to exploit a vulnerability in his computer. The author was in charge of an Internet gateway and knew that it was certainly being probed or attacked on a regular basis. So he installed a few fake services (FTP, Telnet, SMTP, Finger, Rlogin and an unusable guest account) and monitored their usage.

Eventually, an attacker tried to exploit a vulnerability in the SMTP server. This vulnerability allows an attacker to send shell commands embedded into a mail and let the SMTP server execute them as the root user. Since the SMTP server installed was a fake one, the computer was not vulnerable. The attacker was trying to download the `/etc/passwd`, so the author fooled the attacker, pretending his attack worked by sending him a fake password file.

A few days later, the attacker returned and tried to exploit the same vulnerability to change the password of one of the users (called Berferd) defined in the fake password file he retrieved. In order to study the attacker, the author set up an account for the bogus Berferd user, and put it in a chroot jail so the command issued under this account would be harmless. Luring the attacker was a good strategy as he was connected long enough to trace him. By studying the attacker, the author discovered that the attacker was targeting other computers. It allowed the author to warn those other victims and mitigate the damages.

This paper is the first to describe how a fake but realist enough computer environment was set up to lure an attacker in order to study him and mitigate his attacks. Hence, this paper is considered as a seminal work in the field of honeypots.

3.4 Collecting traffic sent to unused IP addresses

Seminal work on honeypots uses fake services that are run alongside with real services on a production host. The connexions and the traffic exchanged with those fake services is then analyzed. Another solution to collect malicious traffic is to listen to traffic sent to unused IP addresses. Since those addresses

[1]The original article did not give all the details of the story because the case was still under investigation at the time of publishing. The full story was later published in the book *The Cuckoo's Egg* [60] where we learn that the attacker was trying to sell the results of his crackings to the Soviet KGB.

are unused, no one should try to contact them. Even if collecting traffic sent on unused addresses is done passively, i.e. without replying to the received traffic, it still fits the global definition of a honeypot. Indeed, those IP addresses have no production value and we can learn a lot by monitoring the traffic they receive. This can be considered as a *zero-interaction honeypot.*

Two methods for collecting malicious traffic have been proposed in [8] and [29, 31]: Darknets and "network telescopes". They use a monitoring station running a network sniffer and a default network route announced toward this station. When a packet is sent to an unallocated subnet of the network, it takes the default route and is analyzed by the monitoring station. However, those methods for collecting malicious traffic are completely passives. They do not respond to the received packets. This implies that in the case of malicious traffic sent over TCP, it only collects the TCP `SYN` segments. This is not sufficient to determine precisely the type of malicious packets received. Since those methods do not reply to incoming connections, they are limited to the analysis of ICMP and UDP traffic or TCP port scans.

3.5 High-interaction honeypots

Collecting the traffic sent to unused IP address does not give enough details on the attack attempts performed. To collect more accurate data, traffic sent to unused IP addresses must not only be collected but also replied to. This allows the attacker to open TCP sessions and to try to perform his attack. To achieve this, honeypots can be used to reply to the traffic sent to unused IP addresses.

A first approach is to use high-interaction honeypots [46], that is, real computers intended to be compromised. A solution using a virtual network of three machines built on top of VMWare has been described in [10] and in [9]. The drawbacks of this approach are that usually, only a small number of IP addresses are used and exposed, and managing several machines is not always trivial.

In the Potemkin Virtual HoneyFarm [69], the authors try to address this problem by proposing a solution for running high-interaction honeypots on a large set of IP addresses without using huge hardware resources. They rely on Xen[36], a virtualizer which allows multiple operating systems to be run in parallel on a single workstation. The authors implemented in Xen a copy-on-write mechanism allowing to quickly create a new virtual machine by cloning and modifying a reference virtual machine. This reduces drastically the cost of instantiating a new virtual machine. This approach looks promising but the authors stated that they are still debugging their prototype as they experienced various crashes during their deployments.

Compared to our honeytanks, those systems offer a higher level of realism since they execute real services and operating systems. Nonetheless, the

realism of the service simulators used in our honeytanks could be improved by using ScriptGen [24]. It generates a state machine mimicking the behavior of a real service based from real samples traces of network traffic. This work is described later in this chapter. High-interaction honeypots are not well suited to collect traffic from a large number of IP addresses. Indeed, they would require to use an important amount of real machines. However, The HoneyFarm approach is interesting as it tries to merge the benefits from low and high interaction honeypots: simulating real applications and operating systems on a limited number of computers. Its implementation is quite complex as it requires important modifications to the Xen virtualizer and the use of a packet filter written in Click [22]. Click is a modular software router. The user can combine together various predefined modules which process network packets. The user can also write its own modules in C++. Once combined, the modules can be run in kernel space under Linux to achieve maximum efficiency.

3.6 Low-interaction honeypots

Instead of executing real operating systems, another solution consists of using low-interaction honeypots to simulate a subset of the behavior of a real host.

Honeyd Honeyd [47] is an example of such a low-interaction honeypot. It can fake the behavior of many TCP/IP stacks implementations. A complete network topology can be simulated by Honeyd, including routing, link latency or bandwidth. It handles by itself ICMP messages and the opening and clearing of TCP connections. The emulated services are not provided by Honeyd. They must be implemented either as external programs which are spawned by Honeyd on each TCP connection or as Python modules interpreted by the Python interpreter linked with Honeyd. Honeyd can also intercept network system calls of standard applications and virtualize their execution inside the honeypot. We provide a detailed comparison between Honeyd and our framework at the end of this chapter. Moreover, in Chapter 10, we show experiments where a stateful honeytank written using our framework outperforms Honeyd.

Internet Sinks Another method has been proposed in [72] that aims at simulating a large number of hosts. The iSink architecture is composed of three parts. The first is a passive monitoring component analyzing flow-information. The second part is an active component based on a VMWare Honeynet. This component replies to part of the incoming traffic. The last part is an active component called the Active Sink. This component can be seen as a low-interaction honeypot. Like Honeyd, the Active Sink handles

TCP connection by itself without using the TCP/IP stack of the underlying operating system. It also contains a set of responders for various protocols (HTTP, Windows RPC Service, ...) implemented in Click. To achieve scalability, the handling of TCP connections as well as the responders are implemented in a stateless way.

Like our honeytanks, iSink simulates a large amount of hosts. However, the main advantage of our method is that it is conceptually simpler. Our framework is self-sufficient and does not require the modification and configuration of other tools like Click. Since it is based on RUSSEL, the user can easily modify the behavior of the honeytanks. Moreover, we show in Chapter 8 how to efficiently implement a stateful simulation of the TCP/IP stack as well as stateful service simulators. Performance tests presented in Chapter 10 shows that maintaining this state as little impact on the performance of our honeytanks.

Labrea Labrea [26] is presented as a "sticky" honeypot. Its goal is to slow down the propagation of worms by accepting and handling TCP connections in a way that forces the attacker to stay connected a long period of time. The technique used by Labrea consists in accepting incoming connections and advertising a low Maximum Segment Size (MSS) option. This forces the worm to send lots of small segments instead of a unique, larger, one. Once the 3-way handshake has been performed, Labrea do not acknowledge the received segments. This forces the worm to retransmit the segments and it thus slow down its propagation rate.

Honeytanks We have presented in [67, 68] the concept of *honeytanks*. A honeytank is a low-interaction honeypot that simulates a large number of hosts over unused IP addresses. The first prototype implemented a stateless emulation of a TCP stack and of service simulators. In this thesis we further develop this concept and present the design and evaluation of several prototype honeytanks that simulate hosts with varying degrees of realism, including a stateful simulation of TCP and of several service simulators.

Nepenthes Nepenthes [1] is a honeypot that aims at collecting self replicating malware such as worms. The approach used by Nepenthes consists in simulating only the vulnerable part of services. Specifically, it only simulates the subset of the service that leads to exploiting the vulnerability. Once a vulnerability is exploited, Nepenthes analyzes the malicious payload and attempt to download the malware as a real infected computer would do. Since Nepenthes does not only replies to the incoming traffic but also mimics a successful vulnerability exploitation by downloading the malware, Nepenthes is sometimes labeled as a "medium-interaction" honeypot.

The simulation of services in Nepenthes is often simple and, in some

cases, it does not require to maintain state. Unlike Honeyd or iSink, Nepenthes does not handle TCP connections by itself. It uses regular TCP sockets and thus relies on the underlying operating system. We provide a detailed comparison between Nepenthes and our framework at the end of this chapter. Moreover, we show in Chapter 8 how to translate the service simulators shipped with Nepenthes into RUSSEL and to integrate them within our stateful honeytank. In Chapter 11, we describe an experiment in which we deployed a honeytank running such translated services from Nepenthes.

Generation of service simulators Low-interaction honeypots simulate a subset of the behavior of real hosts. It means that the simulation of services like SMTP or HTTP must usually be reimplemented from scratch. For example, Honeyd propose various services simulators which have been reimplemented as shell, Perl or Python scripts. Similarly, the authors of Nepenthes provide simulators for the vulnerable parts of known services.

Implementing service simulators is a delicate and time-consuming task which sometimes requires to reverse-engineer the application or the protocol used if they are not documented. The authors of ScriptGen [24, 23] propose to address this problem by automatically generating the service simulators. They define a algorithm called *region analysis*. With this algorithm, given a network traffic trace capturing the interactions between a real service and real clients, they are able to generate a state machine that abstract the observed behavior of the real service. Then they implement the state machine as a Honeyd service.

Another approach for creating service simulators from real samples of network traffic is proposed by Roleplayer [7]. It uses as input two sample traces of a dialog between a client and server. The first one is used as reference, the second is used to discover the location of the dynamic fields of the analyzed protocol. However, the samples must be carefully chosen to be quite similar but still different enough in order for Roleplayer to find the location of the dynamic fields. Unlike ScriptGen, Roleplayer does not export the resulting services as Honeyd modules. Instead, Roleplayer is a standalone application which handles the execution of the generated service simulators by itself. With ScriptGen and Roleplayer, the communication with a low-interaction honeypot can be as loquacious as high-interaction honeypots.

ScriptGen is an interesting complement to our framework. In this thesis, we apply our framework to build, amongst other thinks, a stateful honeytank. We developed a set of stateless service simulators and we show in Chapter 8 how to reimplement the state machine of some stateful service simulators of Nepenthes for our honeytanks. Our framework uses the RUSSEL language which is well-suited for implementing state-machines. Imple-

menting the state machines produced by ScriptGen as service simulators for our stateful honeytank would thus be an easy way to greatly increase the amount and the quality of the service simulators in our honeytanks.

3.7 Honeytokens

In [55], the author introduces the term honeytoken and defines it as a honeypot which is not a computer. The concept is not new (see [59, 13]), but the author was the first to define it and give it a name. The author then proposes two examples of a honeytoken implementation.

In the first example, a bogus e-mail is stored in the senior management's email system. This e-mail seems to be sent by the security help desk and contains a login, a password and the address of a secured web server. The given web address is in fact a honeypot faking a web server and monitoring the usage of the login/password given in the bogus e-mail. If this login/-password is used, this would probably mean that the management's e-mail has been compromised.

In the second example, the author suggests to embed a specific credit card number in a repository like a database or a file server. Then, an intrusion detection system monitors the network traffic and trigger an alert if the byte sequence representing the given specific credit card number is found. The main disadvantage of monitoring the network traffic to detect accesses to the honeytokens embedded in a database is that it will not work if the database outputs its results in an encrypted form. In [4], the authors propose that accesses to the honeytokens should be detected by the DBMS itself. They use a regular table as honeytoken and the fine grained auditing mechanism present in Oracle9i to log all accesses to the honeytoken and launch an external access script. Thanks to the auditing mechanism, the exact query used to access the honeytoken can be saved in the alert log.

In [13], the authors propose to initiate unsecured FTP and telnet sessions using logins and passwords that act as baits for potential attackers eavesdropping the network traffic. Then, they monitor the network with a network sniffer to detect if those logins or passwords appear. If it is the case, it means that an attacker has sniffed a pair of login/password and tries to use it to gain access to a computer.

3.8 Network Traffic Generators

We presented in this thesis techniques for programming efficient and versatile traffic generators. We use them to assess the performance and functionality of honeytanks. Other tools can be used to generate network traffic but we believe that they do not match our needs.

NMap[71] is a port scanner that is able to forge and send arbitrary packets. It can scan multiple hosts and ports and generate packets at various rates. It can also spoof the source IP of the generated traffic. However, its main purpose is to perform a basic request/reply scan. More elaborated test scenarios such as opening TCP sessions or performing OS fingerprinting scans are hardcoded in the program and the user cannot create its own test scenarios. All features of NMap can be reimplemented with our framework.

hping[51] is primarily a network scanning tool but it evolved in a tool able to forge and inject various types of network packets. It can be used to inject bogus packets to test the behavior of firewalls and intrusion detection systems. It is a versatile tool, but it must be scripted in order to perform advanced tests or to contact multiple targeted systems.

Another traffic generator is iPerf [64]. Its main goal is to analyze bandwidth and throughput of a network in order to fine-tune the options of the TCP/IP stacks. It is composed of a server and a client. The client generates TCP or UDP traffic with various options towards the server which can analyze the received traffic. iPerf is a very complete tool with regard to the analysis of TCP and UDP traffic. Unfortunately, it cannot be easily modified to match our needs as we must be able to contact multiple hosts and to spoof the source IP addresses of the generated traffic. Indeed it does not allow to spoof IP addresses or to forge arbitrary packets. Moreover, it cannot be programmed to generate traffic and reply to it according to a complex scenario.

Other traffic generators like [53] are specialized in testing the behavior of firewalls and intrusion detection systems. They come with a complete test suite of scenarios known to pose some problems to firewalls or IDS. Those systems are generally ad-hoc systems and are not designed to be easily modified.

Tcpreplay [65] is a set of tools for replaying previously captured traffic. It can read a tcpdump capture file and replay its contents at arbitrary speeds. Packets cannot be modified on-the-fly while being replayed. However, Tcpreplay can pre-process and modify the contents of a capture file before replaying it. Tcpreplay is a passive tool: it blindly sends the packets read from the capture file and cannot be configured or programmed to interactively react to the traffic it receives in return. In Chapter 10, we use our traffic generators as well as Tcpreplay to assess the performance of our honeytanks and Honeyd.

Tcpreplay could be reimplemented in our framework as we are able to read packets from a capture file and inject arbitrary packets at a given rate. However, the approach used by Tcpreplay is intrinsically more efficient as it only replays packets as they are without having to build them layer-per-layer or modify them. However, our framework allows the creation of more versatile traffic replayers as, for example, packets read from the capture file could be modified on-the-fly before being injected.

TCPopera [18] aims at overcoming some of the limitations of Tcpreplay. When replaying a capture file, it tracks the TCP connections and uses a TCP/IP state machine to store their state. According to information extracted from the trace and according to the received traffic, TCPopera can adapt the way the trace is replayed. For example, it can detect that a replayed packet has not been acknowledged by the receiver and retransmit it. TCPopera is designed to replay packets from a captured trace. It thus cannot be programmed to build packets from scratch and react to the incoming traffic according to a given scenario.

Scapy [3] is an interactive network packet manipulation tool. It can be used as an interactive command-line tool or it can be imported as a module in a Python [66] program. Scapy offers a pleasant API to capture packets, decode and explore their layers, change their contents and inject them on a network. It is a versatile and programmable tool that allows the creation of various types of network applications. Our framework offers the same versatility as Scapy. However, the rule-based paradigm used in our framework is more modular: the developer can solve complex problems by writing rules that focus on independent or loosely coupled tasks that are executed in parallel. Also, the programming system we use is more efficient: for instance, we show in Chapter 7 that our framework outperforms Scapy by a factor 65 on a simple traffic analysis task.

3.9 A Study of Nepenthes

3.9.1 Presentation of the Nepenthes framework

Nepenthes [1] is a honeypot framework that aims at collecting *self replicating malware* such as worms. This kind of malware usually behaves as follows: an infected computer is running the malware; so it contacts other computers to infect them by exploiting a known vulnerability (usually a buffer overflow) in a service running on them. If the vulnerability can be exploited, an executable code — called shellcode — is run into the process of the exploited service. Executing the shellcode causes the exploited computer to download a copy of the complete malware and to execute it. The exploited computer is now infected: it is running the malware and it starts infecting other computers.

The goal of Nepenthes is to collect such malware to study it and to send it to anti-virus manufacturers. The specificity of Nepenthes lies in its approach for simulating application-level services. Other honeypots simulate the normal behavior of services and they attempt to be as close as possible to a real implementation. To the contrary, Nepenthes only simulates the known vulnerabilities of services. More precisely, it only simulates the subset of the service that leads to exploiting the vulnerability.

If we look at the behavior of a vulnerable service as a state machine, *simulating the vulnerabilities* boils down to implementing only the path taken in the state machine by an attacker when he exploits the vulnerability. Most of the time, the resulting "simulation" contains only a few states. For example, the simulation of the Universal Plug and Play (UPnP) service implemented by Nepenthes is trivial as the exploitation of the vulnerability is performed at the first request sent by the attacker. Thus simulating the vulnerability just means waiting for a specific request sent by the attacker.

Different malwares can exploit the same vulnerability. Thus, by simulating a single vulnerability, Nepenthes can collect several malwares or even new ones as long as they exploit a simulated vulnerability. The limitation of this approach is that the Nepenthes team must closely monitor security vulnerability announcements to rapidly create new simulators.

3.9.2 Architecture of Nepenthes

The architecture of Nepenthes consists of a core module and of several other modules implementing specific functionality such as simulating the vulnerabilities, analyzing the shellcodes, and downloading the malware.

Nepenthes core

The Nepenthes core is the main component of the Nepenthes architecture. It is a daemon, i.e., a background process, which manages network connexions and controls other Nepenthes modules. The other modules are C++ classes compiled as dynamic library objects, which are loaded by the core at startup. The core manages a large number of regular network sockets corresponding to well-known vulnerable services for which a vulnerability module exists. The computer running Nepenthes is given a large range of IP addresses so that many instances of the same service (i.e., of the same vulnerability module) are simulated in parallel. Each instance is associated to a different network socket. When a connexion is established on one of these sockets, the core calls a specific method of the vulnerability module associated to the port. This method returns a new instance of a C++ class (called the *Dialogue* class) implementing the state machine of the vulnerable service simulation. When some data is available on the socket, the Nepenthes core reads it into a buffer. Then, it calls another method of the class instance associated to the socket, passing the buffer as a parameter to the method. The method analyzes the buffer and replies according to the state machine simulating the vulnerable service.

It is important to observe that, to the contrary of usual networking applications, such as servers, vulnerability modules are not run as independent processes or threads. The Nepenthes core is run in a single process that monitors the status of all sockets using the `poll()` system call. It reads

data from the relevant sockets and then calls the corresponding methods to process the data. This approach entails much less overhead than the usual approach but it also has some drawbacks that we discuss later on.

Vulnerability modules

The vulnerability modules simulate the vulnerable services. They are usually simple and implement a state machine. The simulation of the services is thus stateful. When the dialog with the malware reaches the point where the vulnerability is exploited and a shellcode is injected, this shellcode is passed to a shellcode parsing module.

Shellcode parsing modules

The shellcode parsing modules analyze the shellcode sent by the malware in order to know how the malware executable must be downloaded. The analysis is done by looking for known patterns of download methods, e.g., URLs. When the method of downloading the malware is found, this information is passed to the adequate fetch module.

Fetch modules

The fetch modules download the malware executable code according to the information received when the shellcode parsing modules. The goal of those modules is to download the malware as a real infected computer would do. Once the malware is downloaded, it is passed to a submission module.

Submission modules

The submission modules may store the downloaded malwares into a database, send them to a centralized point of collect, send them by mail to anti-virus manufacturers, ...

3.9.3 Strengths of Nepenthes

The Nepenthes framework allows the programmer to create new vulnerability modules quickly and easily. The authors of Nepenthes announce that, on the average, less than 500 lines of C++ code are required to write a vulnerability module [1]. In practice, the programmer has to write a C++ class implementing a well-defined interface. This interface contains, among other things, the method `incomingData()` that is called by the Nepenthes core and receives as parameter the data read from a TCP socket. The programmer only has to write code to analyze the data, to wait for additional data if all the bytes of the request have not been transmitted yet, and to maintain state. It does not have to take care about managing the sockets

and reading from them. This ease of programming fosters the quick creation of modules when new vulnerabilities are announced.

3.9.4 Limitations of Nepenthes

Nepenthes uses regular TCP sockets for its network operations. It means that the TCP connexions are handled by the TCP/IP stack of the operating system running Nepenthes. This implies three limitations.

1. First, Nepenthes cannot simulate various types of TCP/IP stacks to fool OS fingerprinting tools as it has no control over the TCP layer.

2. Second, when data become available from a socket, the Nepenthes core iterates over all managed sockets to find which ones have to be read. This entails an efficiency issue when dealing with a large number of sockets.

3. Third, Nepenthes can only reply to traffic sent to the IP addresses managed by the operating system. Thus, deploying a honeypot exposing several thousand IP addresses implies that the operating system running Nepenthes must be configured to handle all those addresses. In [1], the authors of Nepenthes state that the Linux kernel sometimes crashed when it has to handle 16,000 IP addresses in parallel.

3.9.5 Comparison with our framework

The Nepenthes framework is similar to our approach in two important aspects. However our own framework, based on ASAX, has additional strengths. We now elaborate on these issues.

Similarities between Nepenthes and our framework

Single process The Nepenthes framework offers an environment where the programmer can focus on writing the simulation of a vulnerable service without taking care of the interactions with other vulnerability modules or managing the socket as this is done by the Nepenthes core. Moreover, Nepenthes is implemented by a single process, which avoids the overhead caused by a multi-process or multi-threaded approach. The ASAX system is based on a similar philosophy. It offers an environment for parallel programming where the programmer can focus on writing RUSSEL rules. The format adapter takes care of reading and pre-processing the data and the ASAX analyzer takes care of scheduling the execution of rules. To be efficient, the ASAX analyzer is implemented within a single process. Our honeytank framework heavily relies on this feature as it creates lots of rule instances that are managed under a single process by the ASAX analyzer.

Modularity Implementing vulnerability modules in our honeytank framework is as easy as in Nepenthes. The programmer has only to define a RUSSEL module containing the RUSSEL equivalent of the `incomingData()` method. The programmer must write in the body of this rule the code analyzing the payload of the current TCP segment. He can store as many state as wanted through parameters of the RUSSEL rule.

A new instance of this `incomingData` rule is created by the honeytank rules following the evolution of TCP sessions when they detect that a TCP session has been established. This is similar to the Nepenthes core creating a new instance of the *Dialogue* class of a vulnerability module when a new session is established on a socket. The Nepenthes core reads the sockets and passes the data to the vulnerability modules trough a method call. The RUSSEL equivalent is somehow different. Once a `incomingData` rule is created, it will live until the simulation is ended. The rule instance checks each received packet and it tests if it is a TCP segment belonging to the TCP session associated to the rule instance. If it is the case, it directly reads the payload of the current TCP segment and analyzes it as the C++ `incomingData()` method would do.

However, before analyzing the payload, the rule instance must ensure that the current TCP segment is not an out-of-sequence or a retransmitted segment. Hopefully, the rule must not perform this check by itself. All it as to do is to check the value of a given variable of a particular kind. This variable is what we call a global frozen variable. It is described in Chapter 7. Frozen variables are used to pass information between RUSSEL rule instances. The honeytank rules following the evolution of TCP sessions perform this validity check and guarantee that this frozen variable will be set to 1 if the current segment is valid and its payload can be processed.

In Chapter 8, we give a detailed explanation on how to translate a Nepenthes vulnerability module into RUSSEL rules into our framework. Most vulnerability modules are based on the same template; the translation in RUSSEL is thus direct and systematic.

Better efficiency of our approach

Our framework has some advantages over Nepenthes. In our framework, we bypass totally the TCP/IP stack of the operating system to handle the network traffic. It allows us to simulate various types of TCP/IP stacks and to easily and reliably expose large amounts of IP addresses. In [67], we deployed a honeytank that handled up to 40,000 IP addresses. In fact, there are virtually no limits to the number of IP addresses and TCP sessions that can be handled by our honeytanks. Indeed, with the optimized version of ASAX [28, 27], the ASAX analyzer only selects the rule instances that are concerned by the received network packet without having to iterate over all the instances to find them. This claim is demonstrated by experiments

presented in Chapter 10.

3.10 A Study of Honeyd

3.10.1 Presentation of Honeyd

Honeyd [47] is a honeypot framework for deploying low-interaction honeypots. Honeyd aims at simulating a network of several hosts on a single physical computer. The simulated hosts can be organized in a virtual network. Several parameters of this network can be customized such as the latency, the packet loss or the bandwidth of the virtual network links.

Moreover, the virtual hosts can be configured to mimic the behavior of known types of TCP/IP stacks and thus deceive some remote OS fingerprinting tools like Nmap[71] or Xprobe[20]. Honeyd implements a subset of the TCP/IP state machine. It does thus not use the TCP/IP stack of the underlying operating system and handles by itself the TCP connections established to and from the virtual hosts.

Simulation of services can be bound to the simulated hosts. Services can be implemented in three different ways. First, services can be implemented by external programs communicating with Honeyd through their standard input and output. Second, they can be implemented by a Python script executed by the Python interpreter linked with Honeyd. Third, they can be regular network programs using standard TCP sockets. In this case, Honeyd intercepts the network system calls of the service and manages the network communication by itself.

Honeyd is also able to redirect incoming connections to real hosts. This way, it is possible to add a real host running regular services inside the virtual network.

3.10.2 Architecture of Honeyd

The architecture of Honeyd consists in the following components.

Routing component

The routing component is in charge of simulating the routing of the incoming and outgoing packets inside the virtual network. For example, this component simulates the packet loss by dropping packets according to the configured packet loss rate or delay the transmission of packets to simulate the link latency.

Configuration database

The configuration database stores the configuration of the virtual hosts: IP addresses, TCP/IP stack type, simulated services, ... This database also

stores the configuration of the virtual network.

Packet dispatcher

The packet dispatcher receives the incoming packets and inspect them. When a packet is received, its type is first verified. If the type of the packet is not ICMP, TCP or UDP, the packet is logged and discarded. Otherwise, the packet dispatcher verifies its length and checksum. If the destination of the packet is one of the simulated hosts, the packet dispatcher queries the configuration database to retrieve the configuration of the destination host. Then, the host configuration and the packet are transferred to the adequate protocol handler.

Protocol handlers

The protocol handlers analyze and reply to ICMP, UDP and TCP packets received from the packet dispatcher. The ICMP protocol handler replies to all `echo request` messages with the adequate `echo reply` messages. Other type of ICMP messages are handled according to the configuration of the simulated host.

The UDP protocol handler sends the received UDP datagrams to the adequate UDP service simulator specified in the simulated host configuration.

The TCP protocol handler process the received TCP segments with the simplified TCP/IP state machine implemented by Honeyd. This state machine supports the TCP 3-way handshake for connection establishments as well as normal or abrupt connection clearings. However, it does not fully implements receiver and congestion window management [47]. The TCP handler then sends data segments to the adequate TCP service simulator according to the simulated host configuration.

Service simulators

Service simulators are in charge of simulating application-level services such as a mail server or an HTTP server. They receive the incoming data from the protocol handlers. Honeyd supports three types of service simulators: the *services*, the *subsystems* and the *internal services*.

Services The *services* are external programs. Those service programs receive their input on `stdin` (the standard input) and write their output to `stdout` (the standard output). When a connection is established on a port handled by a *service*, Honeyd runs the service program as an external process. Thus, when using *services*, there is one process per active TCP session on the host running Honeyd.

Once a *service* is running, Honeyd extracts the payloads from the received packets and feeds them into the standard input of the service programs. Similarly, Honeyd reads the standard output of those programs and prepares the data for transmission onto the network. The external service is terminated when the associated connection is ended.

Subsystems A subsystem is a regular network program that uses standard TCP sockets to initiate or wait for TCP connections. A subsystem program is bound to a virtual host. When this host is initialized, Honeyd runs the corresponding subsystem program as an external process. Thus, there can be up to one process used per active simulated host.

Subsystem programs are executed by Honeyd using a technique called shared library preloading. This technique allows Honeyd to replace some of the shared libraries used by the subsystem programs by modified versions of those libraries. In practice, the modified versions intercepts system calls related to TCP/IP sockets and redirects them to Honeyd. This way, Honeyd replaces the operating system as far as TCP/IP sockets are concerned. The subsystem can create sockets, initiate or accept connections as usual, but the data exchanged through those sockets is managed by Honeyd instead of the operating system.

Internal services Internal services are Python[66] scripts executed by the Python interpreter linked to Honeyd. The internal services are based on an event-driven model where they must notify Honeyd when they are ready to read or write data. Unlike subsystems, internal services cannot initiate TCP connections. To implement an internal service, the programmer must create a Python module containing four functions:

1. `honeyd_init()`. This function is called when a connection has been established on a port handled by the internal service. It takes no parameters and returns an object. This object stores the state to maintain during the connection. The programmer must create and initialize this object inside the `honeyd_init()` function. Then, Honeyd will always pass this object as a parameter when it calls the other functions of the internal service. This way, the programmer is able to read the stored state and to modify it if needed.

2. `honeyd_readdata()`. This function is called if Honeyd is ready to send data to the internal service and if the internal service has notified Honeyd that it is ready to read data. It takes two parameters: the data sent by a protocol handler and the maintained state (i.e., the object returned by `honeyd_init()`).

3. `honeyd_writedata()`. This function is called if Honeyd is ready to read data from the internal service and if the internal service has noti-

fied Honeyd that it is ready to write data. It takes as a parameters the maintained state and returns the data to transmit onto the network.

4. `honeyd_end()`. This function is called when the connection is ended.

The Python interpreter is only instantiated once, at the startup of Honeyd. Moreover, the Python modules implementing the internal services are loaded once at startup. Then, Honeyd calls the functions of the internal services according to the received traffic. The four functions defined in an internal service are static and are not part of a Python class. Thus, they are not instantiated multiple times. The state maintained for a TCP connection is stored in a Python object initialized and returned by the `honeyd_init()` function. Thus, Honeyd must store and manage those objects. When some data must be read or written on a given connection, Honeyd must retrieve the object associated to this connection and pass it as a parameter to the `honeyd_readdata()` or `honeyd_writedata()` functions.

Personality engine

Before a packet is sent onto the network, it is processed by the personality engine. This component queries the configuration database to retrieve the type of TCP/IP stack associated with the virtual host that has sent this outgoing packet. If needed, the personality engine modifies the packet so that it seems to originate from the associated TCP/IP stack.

3.10.3 Strengths of Honeyd

The main strength of Honeyd is its ability to simulate a realistic virtual network. The user can create a whole virtual topology and configure various aspects such as the packet loss or the latency of the links. Moreover, the virtual hosts can mimic the behavior of known TCP/IP stacks and fool some remote operating system fingerprinting tools.

Additionally, another strength of Honeyd is its versatile support of application-level services. With the *services*, the programmer can write a service simulator in the language of its choice. The only constraint is that the service must communicate through the standard input and output. Thanks to subsystems, Honeyd can use a regular implementation of a service to process the data received by the virtual hosts. For example, it is possible to use Nepenthes[1] as a Honeyd subsystem. However, there is a security risk when using the subsystem mechanism: if the service used is vulnerable to attacks, the host running Honeyd might be compromised. With the mechanism of internal services, Honeyd offers to the programmer an environment to easily create service simulators. The programmer must create a Python module and implements four functions. Most of the simulation must be implemented in the `honeyd_readdata()` function, which receives as a parameter the data to process.

3.10.4 Limitations of Honeyd

While Honeyd provides a versatile support of application-level services, most of them are not adequate for deploying a large-scale honeypot. Indeed, when the *services* mechanism is used, Honeyd creates one process per active TCP session. Honeyd is shipped with several services simulators. Most of them are shell scripts but some of them are Perl scripts. In that case, it means that one Perl interpreter is instantiated per active TCP session. Similarly, the subsystem mechanism uses the regular implementation of a server. With this technique, there is one process per active virtual host. The overhead generated by the use of multiple processes prevent the use of service and subsystem mechanisms for deploying a large number of virtual hosts.

Thus, the only viable mechanism for deploying a large-scale honeypot is the internal service mechanism. However, internal services can only receive data from the protocol handlers of Honeyd and cannot initiate connections by themselves. It is thus not possible to correctly implement some protocols.

For example, the FTP[42] protocol cannot be fully implemented using internal services. When used in active mode, the FTP client opens a connection to the FTP server. Then, the client opens a random port, notifies it to the server and waits from a connexion originating from the server to this port. Thus, an FTP server implemented with the internal service mechanism of Honeyd could not initiate this connection to the client. Since internal service are Python scripts, the programmer might be tempted to use the Python socket library to initiate connections. But such TCP connections would originate from the IP address of the host running Honeyd instead of the IP address of the virtual host and would be rejected by the FTP client.

Similarly, it is not possible to reimplement a honeypot like Nepenthes[1] with the internal services of Honeyd. Nepenthes simulates the vulnerable parts of a service. Once a vulnerability has been exploited, Nepenthes mimics the behavior of an infected host by downloading a copy of the malware. Since internal service cannot initiate connections, this functionality of Nepenthes cannot be implemented.

3.10.5 Comparison with our framework

We have used our framework, based on ASAX, to create a stateful honeytank, i.e., a low-interaction honeypot simulating a large number of hosts. Honeyd shares some functionality with this particular application of our framework.

Like Honeyd, our framework implements a limited TCP state machine to handle by itself the network traffic without using the underlying operating system. Similarly, we have reused the ideas and algorithms used by Honeyd to simulate various types of TCP/IP stacks. However, Honeyd offers

more immediate functionality than our stateful honeytank. For example, the user can customize various parameters of the virtual network such as bandwidth or link latency. Moreover, simulation of services can be achieved through external programs or through Python scripts that must respect a given template.

Better versatility of our approach

While Honeyd offers several configurable functionality, it is not possible to easily add new functionality. For example, the packet handler component or the protocol handlers cannot be easily modified to change their default behavior or to support a new type of protocol. The only programmable aspect of Honeyd lies in the service simulators that can be implemented as external programs or as Python scripts. However, the Python scripts suffers from some limitations as they do not have access to the full captured packets and they cannot initiate connections by themselves. Thus, some services like FTP cannot be correctly implemented by internal Python scripts.

The Honeyd framework is a *configurable* framework whereas our framework is a *programmable* framework. The user can easily modify or add new functionality to our stateful honeytank. Moreover, the stateful honeytank is only one particular application of our framework. The RUSSEL language used by our framework is particularly well suited for implementing state machines. We used it to create traffic generators or traffic analysis programs. Those kind of applications are not feasible with the Honeyd framework.

Better efficiency of our approach

Through the use of the subsystems or service mechanism, Honeyd can use external programs to simulate application-level services. Unfortunately, the overhead of creating an managing multiple process renders those mechanisms useless when deploying a honeypot simulating a large number of hosts. To be efficient, simulation of services must be implemented in Python through the internal service mechanism of Honeyd. However, we show in Chapter 10 that our stateful honeytank using services implemented in RUSSEL outperforms Honeyd and its Python internal services.

Part III

Traffic handlers : honeytanks and traffic generators

Chapter 4

Handling network traffic with ASAX

> One day a tortoise will learn
> how to fly.
>
> Terry Pratchett
> *Small Gods*

In this chapter, we present the modifications needed to add networking capabilities to the ASAX system. First, we give to ASAX the ability to read and analyze network traffic. As the network traffic is made of packets which appear one at a time, the natural solution to read it from ASAX is to create a format adapter which converts those packets in NADF and gives them to the analyzer. We explain how we constructed such a format adapter and how the network packet information is structured inside the NADF records. Second, we modify ASAX to allow it to reply to the received traffic. We use its plug-in system and create a binding with a C library that allows us to create and inject any type of packet on a network. We explain how to use those functions from RUSSEL programs and we present an example of a simple rule replying to ICMP echo requests.

The remainder of this chapter is organized as follows. Section 4.1 presents the format adapter allowing ASAX to analyze network traffic. It explains how the RUSSEL programmer can access the network packet information through the NADF records. Section 4.2 presents the binding with a C library allowing us to create and inject arbitrary packets onto a network and explains how to use it from RUSSEL programs. Finally, Section 4.3 concludes this chapter.

4.1 A format adapter for reading and analyzing network traffic

ASAX has been designed to be flexible and to be able to analyze any type of sequential events through the use of a specialized format adapter. A format adapter converts and possibly enhances the original events into a format understandable by the ASAX analyzer. In this section, we explain how we build a format adapter for the network traffic.

4.1.1 Reading network traffic

To convert network traffic, we must first be able to read it. We could have developed a format adapter that opens various TCP sockets and converts the incoming traffic into NADF. However, this solution is strongly limited. Indeed, a socket is entirely managed by the operating system and the application level data is the only part exposed to the user. It is thus not possible to access the lower layers. Moreover, a socket can only listen to an IP address bound to the host. It is thus not possible to listen to the traffic sent to other hosts. In the case of honeytanks, those limitations are problematic as we need to access all layers of the network traffic to handle the simulation of the various stacks by ourselves. Moreover, we need to analyze the traffic sent to multiple unused IP addresses.

The `libpcap` library. The chosen solution uses the libpcap[61] library. It allows us to put a network interface in a mode called *promiscuous*. This mode allows the user to *sniff* the traffic, i.e. to listen to all the network traffic seen by the network card on the physical link even if it is not intended for the host. Moreover, the data returned by this library are the complete network packets as they are seen by the network interface. It means that the user has a complete view over the various network layers of the packet.

Each network packet returned by the libpcap is converted into a NADF event by the format adapter. The original packet contains layered information that cannot be accessed directly. To access a layer, the lower layers must be first parsed. On the contrary, a NADF record is a flat structure where each field can be accessed independently. For each field of each layer of the original network packet, a NADF field is created in the NADF record. The RUSSEL programmer has thus a direct access to all information contained in the original packet and does not need to parse complex structures. For example, to know if the network packet contains a given high-level protocol, the programmer usually has to parse the whole packet. With ASAX, if a given protocol is not present in the original packet, its corresponding fields are not present in the NADF record. So, to test if a protocol is present, the RUSSEL programmer has just to test the presence of one of the protocol fields in the NADF record through the built-in `present` operator.

The libpcap returns a buffer of bytes containing the captured frame on the data link layer. The format adapter parses this buffer layer by layer. For each layer, the length announced in the header is checked against the size of the captured packet in order to detect truncated packets or corrupted size fields in the header. If it is the case, the translation is stopped at the current layer reached.

Currently, the following protocols are supported, i.e., each field of the protocols are translated into an NADF field. At the data link layer, only the Ethernet protocol is supported. For the network layer, the format adapter supports the ARP, IP and ICMP protocols. Finally, it supports TCP and UDP as protocols of the transport layer. For example, the RUSSEL programmer has access to NADF fields named tcp_source, tcp_dest, tcp_sequence, tcp_ack, tcp_flags which respectively represent the source and destination port, the sequence and acknowledgment numbers and the flags of the original packet. The ADDF file containing the complete name and the description of the NADF fields which can be used by a RUSSEL programmer is given in appendix F.

Enhancing information with the format adapter. The format adapter can also enhance the information contained in the original packet. For example, it adds a field representing the timestamp when the packet was captured. The format adapter can also be used as a pre-processor. We will see in Section 7.2.2 that in some cases a hash of identifying parameters of a signature occurrence must be computed to fully benefit from the optimized version of ASAX. Our format adapter computes this hash on the 4-uple identifier of each TCP segment, relieving the RUSSEL programmer from this task.

Similarly, for some protocols, the format adapter does not simply perform a field-by-field conversion. It also presents the information in a format more convenient for the RUSSEL programmer. For example, when a TCP datagram is translated, its `tcp_flags` field is translated into an NADF field. Moreover, for each flag (`SYN`, `RST`, ...) set in the `tcp_flags` field, a NADF field (named `tcp_syn`, `tcp_rst`, ...) representing the TCP flag is also created. So, in order to check the setting of the `SYN` flag in a TCP datagram, the user then has the choice to either check its value in the translated `tcp_flags` field or to simply test the existence of the `tcp_syn` NADF field.

Finally, the format adapter can also convert the information present in the original network packet. For example, if a field in the original packet stores information in a different endianness than the host, our format adapter converts this information to the correct endianness.

4.1.2 Buffering the received network packets

The format adapter converts a network packet into a NADF event only if the ASAX analyzer asks for a new event to process, i.e., when it has finished to

process all rule instances for the current event. Thus, the time between each call to the format adapter is the execution time of the ASAX cycle. It means that if both the arrival rate of network packets on the network interface and the ASAX cycle time are high, some network packets can be lost. In this situation, the format adapter is called late and its call to the libpcap library to obtain a network packet occurs too late. Hopefully, the mechanism used by the libpcap to retrieve packets uses a buffer storing a few packets. However, in case of important network traffic, we have observed that packets are not taken out rapidly enough from this buffer, which eventually gets overflowed.

To avoid this packet loss, we also maintain a buffer (implemented by a queue) inside the format adapter. Each time it is called by the analyzer, the adapter retrieves all the packets stored in the libpcap buffer and appends them to the queue. This way, the libpcap buffer is emptied at each ASAX cycle, thus reducing the risk of packet loss. Nevertheless, a packet loss could still occur if the incoming rate of packets is greater than the rate of their removal from the libpcap buffer. However, such a packet loss has only been observed under heavy load with artificially generated traffic. Every network device has a limited buffer size and can sometimes lose packets. We believe that the performance achieved by this solution is satisfactory. Moreover, on some occasions we have observed that the call to the `libpcap` library lasts longer than usual and reaches a few thousands microseconds. When it occurs, the call to the format adapter lasts longer than usual and the availability of the current packet is delayed. The ASAX cycle time is thus increased.

We will see in section 7.1.1 that the format adapter must be able to generate fake events and delay the translation of events to simulate foreseeing abilities. We call such a format adapter an *oracle format adapter*. Thus, Algorithm 1 shows how an oracle format adapter can manage a queue of network packets while providing information about the next record. The use of such a format adapter is detailed in Chapter 7.

4.2 Injecting network traffic: binding the `libnet` library

ASAX is hitherto only used passively, i.e., it analyzes traffic without replying to it. However, in the context of honeytanks, ASAX must simulate hosts and thus reply to the received traffic. To achieve this, the RUSSEL program must be able to create a packet seemingly originating from the simulated host. In the previous section we exposed that a traditional network socket could not be used to receive traffic as it is handled by the operating system and hides most of information about the received packet. Similarly, opening and writing to a network socket from a RUSSEL program is not ideal as it

Algorithm 1 Managing a network queue with an oracle format adapter

At initialization:
$first\_event \leftarrow$ create a fake event

When the analyzer asks for a new event:
while <event is available from `libpcap`> **do**
  event ← <get event from `libpcap`>
  <append *event* to the packet queue>
end while

if <the packet queue is empty> **then**
  second_event ← <create a fake event>
else
  second_event ← <pop event from event queue>
end if

nadf ← <convert $first\_event$ and give information about *second_event* >
$first\_event \leftarrow$ *second_event*
return *nadf*

does not allow us to send arbitrarily data. Indeed, contrary to a traditional application which only handles the communication at the application layer, a honeytank must bypass any support from the underlying operating system. It has to handle all by itself the creation of a valid network packet from the data link layer up to the application layer. To ease the task of forging and injecting arbitrary packets on a network, we use the plug-in system of RUSSEL and create a binding with the `libnet`[52] library.

The `libnet` library. This library offers an easy-to-use API, allowing the programmer to forge a packet layer by layer in a top-down way. It automatically computes and fills in the various checksums fields of the forged packet. Using the `libnet` binding is rather straightforward. Figure 4.1 illustrate the use of this library within a RUSSEL program. It is a simple RUSSEL rule replying to every ICMP echo request by an appropriate ICMP echo reply packet. To forge the reply, the rule reuses the values of the identifier, sequence and data fields of the request, here accessed through the NADF fields named `icmp_echo_request_id`, `icmp_echo_request_sequence` and `icmp_echo_request_data`. Similarly, the rule swaps the source and destination IP of the request to forge the IP layer of the reply.

In this example, the programmer must first initialize the libnet library. This is done by calling the `LIBNET_init()` function in the `init_action` rule. This function returns a "libnet context" that must be passed as a parameter to further calls to other libnet functions. Here, this context is stored in

```
global internal libnet_context : string;

init_action;
var outIFace, err_buffer : string;
begin
    err_buffer := GLIB_malloc(256); outIFace := 'eth1';
    libnet_context := LIBNET_init(0, outIFace, err_buffer);

    trigger off for_next echo_reply
end;

rule echo_reply
begin
    trigger off for_next echo_reply;
    if present icmp_echo_request -->
       begin
           LIBNET_build_icmpv4_echo(
           0,                          /* Type */
           0,                          /* Code */
           0,                          /* Checksum */
           icmp_echo_request_id,       /* ID */
           icmp_echo_request_sequence, /* Sequence */
           icmp_echo_request_data,     /* Payload */
           libnet_context,  0          /* Libnet context */);

           LIBNET_build_ipv4(
           20 + 8 + stringlen(icmp_echo_request_data), /* Length */
           0,                  /* TOS */
           0,                  /* IP ID */
           0,                  /* IP fragmentation */
           64,                 /* TTL */
           1,                  /* Protocol (icmpv4 : 1)*/
           0,                  /* Checksum */
           ip_dest,            /* IP Source */
           ip_source,          /* IP Destination */
           '',                 /* Payload */
           libnet_context,  0 /* Libnet context */);

           LIBNET_build_ethernet(
           ethernet_source,  /* Ethernet Destination */
           ethernet_dest,    /* Ethernet Source */
           2048,             /* Ethernet type */
           '',               /* Payload */
           libnet_context, 0 /* Libnet context */);

           LIBNET_write(libnet_context);
           LIBNET_clear_packet(libnet_context)
        end
     fi
end.
```

Figure 4.1: RUSSEL for an ICMP ECHO Request replier

the global variable `libnet_context`. The rule `echo_reply` creates an ICMP echo reply packet. This is done in a top-down way. The rule first forges the content of the upper-layer protocol, ICMP. This is done by calling the `LIBNET_build_icmpv4_echo()` function. Its parameters represents the various fields of the ICMP protocol: the ICMP type, code, ... In this example, the checksum parameter is set to zero. It means that the libnet library will compute the correct checksum by itself. However, the user can override this behavior and set the checksum to a specific value. This is useful for creating incorrect packets in order to test the behavior of a host. Then, the rule traverses the lower layers. It forges the contents of the IP layer and the Ethernet layer. Once the whole packet has been forged, it is sent into the network by calling the `LIBNET_write()` function. Finally, the rule calls the `LIBNET_clear_packet` rule which frees some data internally used by the libnet library.

4.3 Conclusion

We have presented in this chapter extensions that add networking capabilities to ASAX. First, we presented a format adapter based on the `libpcap` library allowing the RUSSEL programmer to access all layers of information contained in a network packet through NADF records. Then, we presented an extension to the ASAX analyzer based on the `libnet` library which allows the RUSSEL programmer to call external functions to create arbitrarily network packets and inject them onto a network. Those extensions are essential to implement honeytanks with ASAX as the simulation must analyze network traffic and reply to it.

This chapter illustrates the versatile nature of ASAX. The network format adapter makes it easy to access packet content and thus to analyze network traffic. Moreover, the role of the format adapter is not limited to a mere translation of the original events but can also enhance or pre-process the information they contain in order to ease the programming tasks of the user. Finally, the binding with a packet creation and injection library shows that ASAX is not limited to passively analyze packets. It can be used to reply to the received traffic and fully interact with other hosts.

Chapter 5

Stateless honeytanks

> Any path that narrows future possibilities may become a lethal trap. Humans are not threading their way through a maze; they scan a vast horizon filled with unique opportunities. The narrowing viewpoint of the maze should appeal only to creatures with their noses buried in the sand.
>
> Frank Herbert
> *Children of Dune*

As the honeytank must emulate a potentially large number of hosts, the first prototype honeytank has been implemented as a completely stateless system in order to maximize its performance. We rely on the fact that, for most protocols, it is possible to infer a suitable response by simply looking at the contents of the request. Indeed, protocols often maintain state in order to detect errors and recover from them. Most of the time, the state stored is not useful if no problems occurred during an exchange between a client and a server. Our goal is to simulate at best the behavior of a host by looking independently at each received network packet. We do not follow the common layered approach where each layer is responsible for a different facet of the communication.

In this chapter, we first describe how to accept TCP session establishments and clearing by exploiting some particularities of the TCP protocol. Then we show how to support standard applications running over TCP. The source code of those simulated applications is presented in Appendix C. We also explain how we can use a host contacting the honeytank to maintain information instead of maintaining it inside the honeytank. An early version of this work has been published in [67].

5.1 Emulation of the TCP three-way handshake

To establish a TCP connection, a client and a server exchange three TCP segments. The first segment sent by the client has its `SYN` flag set and contains a sequence number chosen by the client (x). The server replies with a segment with the `SYN` and `ACK` flags set and a sequence number (y) chosen by the server. This segment acknowledges the received `SYN` segment by containing $x + 1$ as its acknowledgment number. The client will reply to this `SYN+ACK` with a segment whose acknowledgment number is equal to $y + 1$.

Real TCP implementations create state for the new connection either upon arrival of the first (`SYN`) or third (`ACK`) segment and maintain state during the entire TCP session. Our Honeytank establishes a new TCP connection by sending the appropriate `SYN+ACK` segment to each `SYN` segment received. All TCP segments containing only an acknowledgment and no data such as the third segment of the three-way handshake are ignored. For some protocols, a "welcome banner" must be sent by the server directly after the three-way handshake. For those protocols, in addition to the packet containing the `SYN+ACK`, we create and send immediately to the client a packet containing the banner.

The implementation in RUSSEL depicted by Figure 5.1 is rather straightforward. Two rules are running in parallel. The first rule tests the TCP flags of the received segment exported by the format adapter built on top of `libpcap` [61]. If the packet is a TCP segment with the `SYN` flag set, then we select an initial sequence number and send the corresponding `SYN+ACK` segment in reply with the adequate sequence number. The second rule tests if the current record is an `ACK` segment and observes the contents of the TCP payload. If it is empty, the received segment is silently discarded. Otherwise, we analyze its content and we send the required reply segment (this part is explained later). A new instance of both rules are retriggered for the next received packet.

So far, a human interacting with the honeytank could easily detect that the servers are emulated. A simple method to detect the honeytank is to send a TCP segment containing data on a non-established TCP connection. A normal TCP implementation would return a `RST` segment, a stateful firewall would drop the segment and the honeytank will acknowledge the segment as if the connection was already established.

This problem can be solved by relying on the acknowledgment numbers. The initial sequence number can be freely chosen by the honeytank and is used as a reference point to number the bytes sent by the honeytank. When the client sends a message to the honeytank, it must also include an acknowledgment number indicating the expected byte number of the next transmission. Our solution maintains state outside the honeytank by carefully choosing the initial sequence number. During the

```
/* Initialisation rule */
init_action;
begin
        trigger off for_next wait_for_syn;
        trigger off for_next wait_for_ack
end;
rule wait_for_syn;
var sequence : integer;
begin
     if
        present tcp_flags_syn  and not present tcp_flags_ack and
        not present tcp_flags_rst and not present tcp_flags_fin -->
        <Choose the initial sequence number>
        <Send a SYN+ACK reply>
     fi;
     trigger off for_next wait_for_syn
end;

rule wait_for_ack;
var response : string;
begin
     /* Is it an ACK alone ? */
     if
        present tcp_flags_ack and not present tcp_flags_syn and
        not present tcp_flags_rst and not present tcp_flags_fin -->
        if
           tcp_payload not present --> skip; /* Ignore ACK without payload */
           true --> <Handle the protocol emulation here>
         fi
     fi;
     trigger off for_next wait_for_ack
end;
```

Figure 5.1: The TCP three-way handshake in RUSSEL

three-way handshake, when the first `SYN` segment arrives, we use a hash function to select the sequence number to put in the `SYN+ACK` segment sent in reply. Let H be such a hash function. This sequence number is $H(IP_{src}, IP_{dst}, port_{src}, port_{dst}, Secret)$ where $Secret$ is a configuration parameter of the Honeytank. When receiving a TCP segment with the ACK flag set, the hash value $H(IP_{src}.IP_{dst}.port_{src}.port_{dst}.Secret)$ is computed. This hash value is then compared with the acknowledgment number present in the packet. We define as a configuration parameter an "acceptance window" around the hash value. If the ack number falls inside this window (i.e. is "near" the hash value), this segment is assumed to be part of an initiated TCP session. Otherwise, the segment is silently discarded.

5.2 TCP connection clearing

A TCP connection is composed of three phases : the three-way handshake, the data transfer phase and the clearing phase. TCP supports two types of connection clearing : a graceful clearing with the exchange of segments with the `FIN` flag set and an abrupt clearing with the utilization of the `RST` flag.

In the stateless honeytank, we emulate the clearing of a TCP connection the following way. If the client initiates the clearing of a TCP session and sends a TCP segment with the `RST` flag, it means that it will not send other TCP segments belonging to this session after this and it is not expecting any reply. All the honeytank has to do is to ignore this kind of segments. If the honeytank receives a `FIN` segment, it means that the client is gracefully closing the TCP session. The honeytank replies with a `FIN+ACK` segment. When the client receives this last segment, it will reply with an `ACK` segment confirming that the connection has been closed. The honeytank must not reply to this final `ACK`. As `ACK` segments without payload are already ignored, there is nothing more to do.

The stateless honeytank can also initiate the clearing of a TCP session. This decision is taken by the rules simulating application protocols. However, they should terminate TCP sessions abruptly by sending a `RST` segment. Indeed, if they initiate a graceful closing with a `FIN` segment, the client will reply with a `FIN+ACK` segment. As no state is maintained, this reply will be mistaken for a TCP session clearing initiated by the client and a `FIN+ACK` reply will be sent by the honeytank. However, it is not mandatory to initiate the clearing of session. In this case, the client will sooner or later be in timeout and will take the initiative to clear the session. As we do not maintain state, leaving a TCP session opened for a long time has no performance impact on the honeytank.

5.3 Emulation of stateless protocols

The first server we implemented in this stateless honeytank provides the echo service. This standard service is supported by most TCP/IP implementations. The client establishes a TCP connection to the server and all the information sent by the client is echoed by the server. To emulate this service, the stateless honeytank simply replies to each received data segment with a data segment constructed as follows : the payload of the reply segment is equal to the payload of the received segment. The sequence number of the reply segment is set to the acknowledgment number of the received segment plus one. The acknowledgment number of the reply segment is set to the sequence number of the last byte in the received segment plus one. This implies that we assume that all segments were received correctly and in sequence. If TCP segments are lost, the honeytank will not send the acknowledgments, allowing the sender to retransmit the lost segments. As the objective of a honeytank is to receive malicious traffic, this is not a serious problem.

Another example of a stateless application-level protocol is the HyperText Transfer Protocol (HTTP) [11]. HTTP is widely used and several worms spread through vulnerabilities in various HTTP servers [5]. A HTTP server accepts TCP connections established by clients on port 80. After the three-way handshake, the client sends a request. This request can be a single command with one parameter, for example `GET index.html` or a one line command with parameters and additional information in subsequent lines. The server analyzes the received request and sends a response composed of a status line, a header with optional information and often a MIME document.

The simplest approach to emulate a HTTP server without maintaining any state is to assume that when the client sends a HTTP request, it will send it inside a single TCP segment. In this case, the server can use regular expressions to match the content of the received TCP segments with the standard HTTP commands and send the corresponding responses. If no matching rule is found for the received TCP segment, the honeytank acknowledges the received segment. Table 5.1 shows the regular expressions used in our stateless honeytank to emulate HTTP. The configuration of an emulated HTTP server can be tailored to fake different types of HTTP servers by providing different `Server` identification strings in the HTTP responses.

Using regular expressions to determine the command sent by the client is not perfect from a theoretical viewpoint. A first possible problem is that the request may be sent inside several TCP segments instead of a single one. In practice, when testing the honeytank with several HTTP clients, we found the emulation to be sufficient. A second possible problem is that a MIME document containing HTTP commands may be attached to a HTTP request. This would happen for example when a client is using a `POST`

Table 5.1: Emulation of HTTP/1.1

Regular expression	Reply
GET .* HTTP/1.1\n	HTTP/1.1 200 OK\nHTML page
HEAD .* HTTP/1.1\n	HTTP/1.1 200 OK\nLast-modified:...
CONNECT .* HTTP/1.1\n	HTTP/1.1 200 OK\n\n
TRACE .* HTTP/1.1\n	HTTP/1.1 200 OK\n Content-type:message/HTTP\n\n...
OPTIONS .* HTTP/1.1\n	HTTP/1.1 200 OK\n Allow: CONNECT,... \n\n
PUT .* HTTP/1.1\n	HTTP/1.1 201 Created\n\n
DELETE .* HTTP/1.1\n	HTTP/1.1 200 OK\n\n
POST .* HTTP/1.1\n	HTTP/1.1 200 OK\n\n

request to send to the emulated server the RFC defining HTTP. However, in practice this is unlikely to happen. As our objective is to collect malicious traffic, an incorrect reply to a malicious request is not a major problem.

Although many application-level protocols use ASCII encoded commands and replies, some important applications exchange binary messages. As an example of such protocols, we consider the Remote Procedure Call (RPC)[57] based services which are often targeted by worms. Among the available RPC services, the portmapper/rpc.bind [56] is important as it listens to a standard TCP port and provides the list of RPC services supported on the local system and the TCP/UDP ports used to reach those services. A request to the portmapper is a standard RPC call addressed to the service number 100000 on port 111. To emulate the portmapper service in our Honeytank, we created a rule waiting for a RPC message on the default portmapper. This request is encoded in XDR format and contains the service number and a transaction identifier. We extract this identifier and use it to build a matching RPC reply containing a list of fake RPC services emulated on the honeytank.

The source code of those RUSSEL application simulators are presented in Appendix C.

5.4 Emulation of stateful protocols

As another example of protocol implemented on the stateless honeytank, we consider the Simple Mail Transfer Protocol (SMTP) [41]. SMTP is the standard protocol used to send electronic mail. SMTP servers are subject to different types of attacks such as buffer overflows or spammers using them as relays to send unsolicited emails. SMTP is a stateful application level protocol running over TCP on port 25.

During a SMTP session, an email is transferred in a few steps. After the TCP three-way handshake, the server sends its banner. The client sends a

`HELO` message and the server confirms by sending a one-line reply containing a number and a comment. After this exchange, a mail transaction can start. The client issues a `MAIL FROM` command indicating the sender's email address. The server replies and indicates whether the sender is acceptable or not. Then, the client issues one or several `RCPT TO` commands that are confirmed by the server. Finally, the client uses the `DATA` command to send the entire email followed by a single line containing ".". The arrival of the email is confirmed by the server. From a theoretical point of view, the server should refuse a `DATA` command if the client has not correctly sent a `HELO`, `MAIL FROM` and `RCPT TO` command previously. As we propose a stateless implementation, we will not be able to detect misbehaving clients. However, in practice such a situation is unlikely to happen.

To implement this exchange, the first problem to solve is the SMTP banner. This banner must be sent after the arrival of the TCP `ACK` segments that concludes the three-way handshake. To correctly implement SMTP without maintaining any state inside honeytank, we need to distinguish between the first `ACK` segment on a connection and another `ACK` sent by a client later. This problem can be solved by relying on the acknowledgment number contained in the `ACK` segment of the three-way handshake and by adapting the solution presented in Section 5.1. We use a different method to choose the initial sequence number if the 3-way handshake concerns an application-level protocol which needs to send a banner. When the first `SYN` segment arrives, we use a hash function to select the sequence number to place in the `SYN+ACK` segment sent in reply. If the application-level protocol does not send a banner, this sequence number is $H(IP_{src}.IP_{dst}.port_{src}.port_{dst}.Secret_1)$ where $Secret_1$ is a configuration parameter of the Honeytank. If the application-level protocol needs to reply with a banner to the third segment of the three-way handshake, the sequence number of the SYN+ACK is $H(IP_{src}.IP_{dst}.port_{src}.port_{dst}.Secret_2)-1$ where $Secret_2$ is different from $Secret_1$. When emulating an application-level protocol requiring a banner, we compare the acknowledgment number of each ACK segment received. As the SYN flag consumes one TCP sequence, a banner will need to be sent if the received acknowledgment number is equal to $H(IP_{src}.IP_{dst}.port_{src}.port_{dst}.Secret_2)$. Otherwise, no banner is necessary.

The main problem to emulate an SMTP server is to correctly reply to the mandatory SMTP commands [41]. In our prototype, we implemented a simple solution based on regular expressions. When a TCP segment arrives, we check whether its payload matches one of the regular expressions of table 5.2. If so, a TCP segment containing a positive reply is sent and the received segment is acknowledged. Otherwise, we assume that the received segment is part of the body of an email message being received after a successful `DATA` command and we simply acknowledge the segment. The source code of this SMTP server simulator is presented in Appendix C.

Table 5.2: Emulation of SMTP with regular expressions

Regular expression	Reply
HELO .*\n	250 Requested mail action okay, completed
MAIL FROM:.*\n	250 Requested mail action okay, completed
RCPT TO:.*\n	250 Requested mail action okay, completed
DATA\n	354 Start mail input; end with <CRLF>.<CRLF>
\n.\n	250 Requested mail action okay, completed
NOOP\n	250 Requested mail action okay, completed
QUIT\n	221 Closing
TURN...	502 Command not implemented

We verified that this implementation based on regular expressions was sufficient to interact correctly with several SMTP clients. When deployed on a campus network, a stateless honeytank running this SMTP simulation was able to collect spam sent by a spammer searching for open SMTP servers. Yet, since no state is maintained, a human attacker can easily detect that he is interacting with a fake SMTP server. For example, our server will accept a `DATA` command even if it is not preceded by a `MAIL FROM` and `RCPT TO` command. Similarly, if the string "`RCPT TO: \n`" appears in the DATA section of a mail our server will send a "250 OK" reply rather than ignore it.

5.5 Using TCP extensions to maintain state outside the honeytank

During the three-way handshake, current TCP implementations include TCP options to negotiate parameters of the TCP connection such as the Maximum Segment Size (MSS) [40] or the utilization of TCP extensions such as the window scale option, the timestamps option [21] or the selective acknowledgments. The MSS option indicates the maximum size of the segments to be sent on the TCP connection. Each TCP end-system indicate the largest segments it is able to accept on the connection. In practice, most TCP implementations use the minimum value proposed by both end-systems with a minimum of 536 bytes. Our current implementation of honeytank assumes this default MSS value but will accept any unfragmented TCP segment.

The most useful TCP extension for the honeytank is the TCP timestamp option. Most current TCP implementations support this option [21]. This option was developed to improve the accuracy of the round-trip-time measurement and to protect high bandwidth-delay product TCP connections against wrapped sequence numbers. Each TCP segment may contain a timestamp option. If the client supports this extension, it includes a null timestamp option in the `SYN` segment. If the server also supports this op-

tion, it includes it in the `SYN+ACK` segment. A timestamp option contains two 32 bits timestamps : `TSval` and `TSecr`. When a TCP entity sends a TCP segment, it will place in the `TSval` field a 32 bits integer, usually derived from its clock. The `TSecr` will contain the last value received from the other entity in the `TSval` field.

For example, consider an emulated server willing to verify that the client correctly finishes the three-way handshake. Without maintaining state in the stateless honeytank, we need to distinguish between the `ACK` segment finishing the three-way handshake and a normal `ACK` in the flow of an established connection. With the timestamp option, a simple solution to this problem is to copy the sequence number of the `SYN+ACK` segment inside the `TSval` field of the timestamp option. When replying to this `SYN+ACK` segment, the client will echo this value in the `TSecr` field of the `ACK` segment. By comparing the `TSecr` field and the acknowledgment number, the honeytank will easily determine whether it is the third segment of a three-way handshake or a segment sent during the normal data flow. We will provide more details about the TCP timestamp options in the Honeytank in section 5.4.

The timestamp option could also be used to maintain state in order to simulate stateful application-level protocols in a more realistic way. The regular expression approach is sufficient to emulate a simple SMTP server. However, its behavior is far from a normal server maintaining state. To emulate such a stateful server without maintaining any state in the stateless honeytank, we need to determine the state of the emulated server from each TCP segment received from the client. With the normal TCP protocol we used until now, it is not possible. However, the TCP timestamps can be used to store the state of the SMTP in the client as follows. We will use the `TSval` field of the TCP timestamp to store information about the state of the emulated server. When a packet is received by the server, the `TSecr` field contains the last value of the `TSval` field which has been received by the client. In order to respect the TCP timestamp specification, the `TSval` value must be strictly increasing between each packets. To achieve this, we consider that the first 24 most significant bits of the field are seen as a counter that will be incremented by one each time a packet is sent. We call this the "counter" part of `Tsval`. This counter is incremented for each TCP segment sent. The last 8 least significant bits are used to store the state of the emulated server. We call this the "state" part of `Tsval`

First, we place the value 1 in the state part of `TSval` in the TCP segment containing the banner. This timestamp value indicates that the server is expecting the `HELO` command. Upon arrival of a TCP segment with the state part of `TSecr` set to 1, the SMTP server will only accept a `HELO`. The reply message sent in response to this command will contain a state set to 2. This timestamp indicates that the STMP server is expecting a `MAIL FROM` command. A state value of 3 is used when the SMTP server expects the

first `RCPT TO` command. After this command (state=4), the client can either send another `RCPT TO` command or an email by using the `DATA` command. During the transfer of the email message, the server sends TCP segments with the state part of `TSval` set to 5. In any state besides the email message transfer, the server will correctly reply to a `QUIT` and an invalid command. Table 5.3 shows an emulated SMTP server using TCP timestamps.

Table 5.3: Emulation of SMTP with TCP timestamps

`State received`	Regular expression	`State sent`	Reply message
1	HELO .*\n	2	250 Requested mail action OK
1	MAIL|RCPT|DATA	1	503 Bad sequence of commands
2	MAIL FROM:.*\n	3	250 Requested mail action OK
2	HELO|RCPT|MAIL|DATA	2	503 Bad sequence of commands
3	RCPT TO:.*\n	4	250 Requested mail action OK
3	HELO|MAIL|RCPT|DATA	3	503 Bad sequence of commands
4	RCPT TO:.*\n	4	250 Requested mail action OK
4	DATA\n	5	354 Start mail input; end with <CRLF>.<CRLF>
4	HELO|MAIL	4	503 Bad sequence of commands
5	\n.\n	2	250 Requested mail action OK
5	!(\n.\n)	5	*no reply, received segment is acknowledged*
<5	QUIT\n	999	221 Closing
<5	VRFY|EXPN|HELP...	`TSecr`	502 Command not implemented

The TCP timestamps can also be used to recover from some segment loss and reordering of TCP segments. For example, consider a SMTP client sending the `MAIL FROM:` command in two segments, the first containing `MAIL` and the second `FROM: a@a.com`. If the first segment is lost, the Honeytank will receive a segment with the state part of `TSecr` set to 2 that does not match. The received segment should be ignored (and thus not acknowledged) by the Honeytank. As the TCP implementation on the client maintains state, it will retransmit the two missing segments. Furthermore, as TCP implementations support the Nagle algorithm, the client will retransmit all its unacknowledged data in a single segment.

5.6 Conclusion

We have presented in this chapter a stateless honeytank. We have explained how to handle the TCP 3-way handshake and clearing phases without maintaining state on the honeytank side. We have presented techniques for implementing stateless service simulators for some application-level protocols such as HTTP or SMTP.

Chapter 11 presents an experiment where a stateless honeytank was deployed on a campus network. It shows that it was able to establish TCP

connections and collect various types of attacks. The stateless SMTP simulator was accurate enough to collect SPAM mails.

However, stateless honeytanks suffer from some limitations. First, they can be detected by an attacker. Since the TCP/IP protocols and the other application-level protocols are neither correctly nor fully implemented, a human attacker could easily find out that he is not interacting with a real computer. In Chapter 10 we compare the functionality of stateless and stateful honeytanks and show some examples where a stateless honeytank can be detected by an attacker. Moreover, a stateless honeytank is unable to fool remote operating system fingerprinting tools like Nmap [71]. We explain in Chapter 6 how maintaining state allows a honeytank to fool such tools.

Second, a stateless honeytank does not maintain state about the established TCP connections and is unable to detect most retransmitted or out-of-sequence segments. This is problematic since application-level simulators presented in this chapter process the payload of all received segments regardless of the validity of those segments. We present in Chapter 8 the design of a stateful honeytank and stateful application simulators that overcome this problem. The stateful approach explicitely maintains state and simplifies the conception of honeytanks. Indeed, the stateless approach heavily relies on specificities of the TCP protocol in order to avoid to maintain state in the honeytank. The programmer must thus have a deep knowledge of Internet protocols to implement a honeytank using the stateless approach presented in this chapter.

Chapter 6

Responding realistically to fingerprinting tools

'Questions don't have to make sense, Vincent,' said Miss Susan. 'But answers do.'

Terry Pratchett
Thief of Time

The TCP and IP protocols have many different implementations (called TCP/IP stacks) that can be found in different operating systems or in various devices such as networked printers, wireless access points or even cellphones. However, the behavior of the different TCP/IP stacks is not always identical. For instance, it can be due to some degrees of freedom offered by the protocol specifications or simply to an incorrect implementation.

Thus it is possible to identify a particular TCP/IP stack by observing its behavior in specific cases. This technique is called *Remote Operating System fingerprinting* and has been presented in [12] and implemented in some network scanners like NMap[71] or XProbe[20]. This technique tests the behavior of the stack in a few specific and often borderline cases. The results are then matched against a database of known TCP/IP stacks. While this testing approach clearly cannot capture the full complexity of a TCP/IP implementation, it is often and unfortunately the only possible approach to characterize the behavior of a TCP/IP stack since the source code of the stacks is rarely public.

As a honeytank aims at emulating a lot of devices, it can be interesting to give it the ability to simulate various TCP/IP stack for those devices. To differentiate such a honeytank from the stateless honeytank, we call it an *extended* honeytank. The first benefit is that the network of machines simulated by the honeytank seems to be much more realistic when scanned by a tool doing OS fingerprinting. Indeed, when that kind of tool scans the

network simulated by the original stateless honeytank, it reports that all the machines are of an unknown type. The second benefit is that it might improve the quality of the malicious traffic collected. At this time we are not aware of the existence of a worm or other form of automated attack which performs an OS fingerprinting before contaminating its target. However, if such a worm appears in the future, emulating various TCP/IP stacks will be a mandatory step in order to catch this kind of malicious traffic.

Since NMap is a commonly used network scanner, we use it as a reference point for designing and testing our implementation. The TCP/IP stacks behavior simulated by our extended honeytank shall be correctly identified by NMap. More specifically, we will reuse the database of NMap in order to reply correctly to its tests. This approach has first been proposed by Niels Provos in [47].

In the remainder of this section we describe how we modify the stateless honeytank to give it the ability to simulate the behavior of various TCP/IP stacks. We first explain how the behavior of a TCP/IP stack is characterized by NMap. Then we show what must be changed in the honeytank in order to simulate the various TCP/IP stacks profiles. An early version of this work has been presented in [68].

6.1 Characterizing a TCP/IP stack (NMap)

In order to identify a TCP/IP stack, NMap tests the behavior of the stack in a few specific cases. Firstly, it sends data to the stack to trigger those specific cases. The characteristics of the replies (or the absence of replies) to those tests are then matched against a database of known operating systems. NMap performs nine tests divided into three categories.

The first category contains only one test whose purpose is to analyze the TCP 'initial sequence number' sequentiality. This test is known as the *TSeq* test. In the TCP protocol, each side of a TCP connection uses *sequence numbers* to keep track of the number of bytes sent and received. However, the sequence numbers do not have to start at zero. So each time a new TCP connection is established, the TCP stacks are free to choose a new initial sequence number (ISN). Choosing always the same ISN is not strictly forbidden by the TCP specification but it can pose a security risk as some type of attacks rely on the predictability of the TCP sequence numbers [2]. So most of the stacks use different ISN. The way they choose the ISN can be used to identify the stacks.

For this test, NMap initiates several new TCP connections and analyzes the evolution of the ISN chosen by the target. Five classes of behavior are recognized: constant (the ISN is always the same), constant increments (the ISN are incremented by a constant factor), time dependent (the ISN are incremented every x milliseconds), random increments (the ISN are in-

Test name	**TCP flags set**	**Destination port**
T1	SYN	Opened port
T2	No flags set	Opened port
T3	SYN, FIN, URG, PSH	Opened port
T4	ACK	Opened port
T5	SYN	Closed port
T6	ACK	Closed port
T7	FIN, PSH, URG	Closed port

Table 6.1: NMap General behavior tests summary

cremented by random factors) and truly random (the ISN are randomly chosen).

The second category contains seven tests whose purpose is to analyze the general behavior of the TCP/IP stack. Those tests are known as the *T1* to *T7* tests. They consist in sending TCP segments with various flags configurations to opened and closed ports. They are summarized in Table 6.1. For each test, the following characteristics of the answer are analyzed: the presence of an answer to the test, the value of the *Don't fragment* flag, the value of the TCP window size, the value of the acknowledgment number, which TCP flags are set and which TCP options are used.

The last category contains one test whose purpose is to analyze the reaction of the stack when an UDP packet is sent to a closed port. It is known as the *PU* test. For this test, an UDP packet is sent to a closed port. The normal behavior for the targeted stack is to reply with a ICMP Port Unreachable message. Such ICMP error messages normally include the first bytes of the original request which triggered the error. The length and the integrity of the original message included in the reply often varies from one stack to another. This can be used as a test to differentiate the stacks.

The following characteristics of the answer are analyzed: the presence of an answer to the test, the value of the *Don't fragment* flag, the value of the *Type of Service* field in the IP packet and the length of the ICMP message. Then NMap checks if the original message included in the reply was modified by the stack. The following values are checked : the length of the original IP and UDP packets, the original IP ID, the original IP and UDP checksum and the original UDP payload. With those nine tests, NMap is able to identify about 1.300 different TCP/IP implementations.

6.2 Adding TCP/IP stack emulation to the honeytank

The stateless honeytank presented in the previous chapter simulates the TCP/IP stack of all hosts with only two RUSSEL rules. Each rule is instantiated once and the two rules instances are executed in parallel. The first one waits for a TCP segment with the `SYN` flag set. Then it chooses an initial sequence number and sends the appropriate `SYN+ACK` segment. The second one waits for a TCP segment with the `ACK` flag set and some data present in the TCP payload. Then it analyzes the payload and infers some suitable reply according to the emulated application-level protocol.

In order to create an 'extended version' of a honeytank able to emulate various TCP/IP stacks, we must change this design. Instead of coding the TCP/IP simulation in two rules, we must define one rule per type of TCP/IP stack to be simulated. This way, each rule declaration embed the behavior of a specific type of stack. Moreover, multiple IP addresses in the network handled by the honeytank can be bound to the same type of TCP/IP stack. To handle this, one instance of a stack emulation rule is created per IP address bound to this type of stack. So, this extended honeytank is no longer stateless as it has to maintain a rule instance per emulated IP address.

To reply correctly to the NMap tests, we follow the same approach and algorithms as [47] by reusing the entries in the fingerprinting database of NMap. The behavior of a type of TCP/IP stack is coded in the stack emulation rules as follows.

The tests *T1* to *T7* as well as the *PU* test are easy to handle. Each test sends a specific request to the target and expects a specific answer. Since there is no dependency between the tests, the Honeytank does not need to maintain state to reply correctly to them. In terms of implementation, a stack emulation rule contains some tests instructions to determine if the current packet has the characteristics of a packet sent for a *T1* ... *T7* or *PU* test. In this case, we forge an answer whose parameters comply with the expected values for this test for the chosen TCP/IP stack to emulate. The expected values for each test can be found in the fingerprinting database shipped with NMap.

To reply correctly to the *TSeq* test, the emulation rule must choose a correct initial sequence number each time a TCP 3-way handshake is initiated. It is initiated by sending a `SYN` segment to an opened port, which has exactly the characteristics of a packet sent for the `T1` test. Thus, choosing the correct ISN will be handled in the same time as replying to the `T1` test. This way, each rule instance has to maintain a state holding the value of the last chosen ISN.

Thus, unlike the stateless honeytank, the extended honeytank cannot choose freely the ISN. It means that it cannot reuse the technique presented

in Section 5.1 that computes a specific ISN via a hash function. This technique allows the stateless honeytank to know in most cases if a received segment is part of an established session. Without this technique, a problem might arise in the extended honeytank. This problem occurs when the incoming traffic rate is high and causes two TCP sessions openings on the same host to interleave. When the extended honeytank receives the `SYN` segment of the first TCP session, it chooses an initial sequence number, remember it, and replies with a `SYN+ACK`. The honeytank must then receive an `ACK` segment to complete the TCP session opening. If at this moment the honeytank receives a `SYN` segment from a second TCP session opening, it will choose another initial sequence number, remember it – thus overwriting the previous value saved – and send a `SYN+ACK` reply. When the honeytank receives the `ACK` segment finalizing the 3-way handshake of the first TCP session, it detects that the acknowledgment number of the segment does not match the last initial sequence number and thus it rejects the TCP session opening by issuing a `RST` segment. However, in practice, this situation occurs rather infrequently.

The skeleton in pseudo-code of a simplified stack emulation rule is given in Figure 6.1. We developed a Python script which completely automates the process of creating the TCP/IP stack emulation rules. The script gives to the user the choice of the stacks to emulate. Then it parses the fingerprinting database of NMap and generates the stack emulation rules by filling the skeleton with the appropriate data for the selected TCP/IP stacks.

Note that even if we have to maintain some state to be able to simulate various TCP/IP stacks, the design and implementation of the rules emulating application protocol (such as `HTTP`, `SMTP` and `rpcbind`) can be reused just as they are. Indeed, the fingerprinting done by NMap does not involve fully opening TCP connections nor analyzing the behavior of higher-level application protocols. The emulation of the application protocols is stateless in this extended honeytank.

```
/* Emulation of the XYZ TCP/IP stack */
rule IP_stack_emulator_XYZ(simulated_IP_address : string; last_ISN : integer);
  begin
    if
       <Destination IP == simulated_IP_address>
       --> if
              <Packet is ICMP Echo Request>
              --> <Forge and send ICMP Echo Reply>

              <Packet is UDP>
              --> <Reply to the NMap PU test>

              <TCP segment to an opened port>
              --> if
                     <SYN>
                     --> <Choose a correct ISN>
                         <Reply to the NMap T1 test>

                     <ACK>
                     --> if
                            <TCP Payload is empty>
                            --> if
                                   /* Empty ACK not from a 3-way handshake ? */
                                   <TCP ACK number != last_ISN+1>
                                   --> <Reply to the NMap T4 test>
                            else
                            --> <Handle payload>

                     <FIN> --> <Close the connection>

                     <NULL> --> <Reply to the NMap T2 test>

                     <SYN FIN PSH URG> --> <Reply to the NMap T3 test>

              <TCP segment to a closed port>
              --> if
                     <SYN> --> <Reply to the NMap T5 test>

                     <ACK> --> <Reply to the NMap T6 test>

                     <FIN PSH URG> --> <Reply to the NMap T7 test>

    retrigger IP_stack_emulation_XYZ(simulated_IP_address, last_ISN)
  end
```

Figure 6.1: Simplified skeleton of a TCP/IP stack emulation rule

Chapter 7

Parallel programming with ASAX

> It is well known that a vital ingredient of success is not knowing that what you're attempting can't be done.
>
> Terry Pratchett
> *Equal Rites*

In Chapter 2 we introduced the original version of ASAX, the sequential event analyzer on which our framework is based. In Chapter 4, we presented the modifications we contributed to ASAX to give it the ability of reading and writing arbitrary network packets. In this chapter, we present a methodology for writing efficient parallel programs with ASAX.

As the approach used by ASAX can seem unfamiliar, it can be difficult at first to write correct and efficient analysis programs. We thus present various solutions to the problem of finding non-overlapping occurrences of a given signature in an event flow. Solving this problem is important for writing our honeytanks as it will be necessary to correctly follow TCP sessions. We introduce two new concept to ASAX in order to ease the programming of parallel programs: the *frozen* variables and the *oracle* format adapter.

We present a methodology for writing parallel programs that benefit from the optimized version [28, 27] of ASAX. This methodology is applied in Chapter 8 and allow us to write a stateful honeytank and thus increase the quality of the simulation with nearly no performance impact compared to the stateless honeytank.

Finally, we solve a simple yet realistic problem using both our framework and Scapy[3], a Python[66] library for manipulating network packets. We compare both approaches and evaluate their performances. On this example, our framework outperforms Scapy by a factor 65.

The remainder of this chapter is organized as follows. In Section 7.1, we explain how to write correct programs for ASAX and avoid common pitfalls. Section 7.2 explains how to modify our programs to take advantage of an optimized version of ASAX. Section 7.3 compares our framework with Scapy. Finally, Section 7.4 presents the conclusion of this chapter.

7.1 Methodology for writing rules

Since the approach used by ASAX and its RUSSEL language can seem unfamiliar to most users, it can be difficult at first to get rid of old programming habits and write correct and efficient analysis rules. A methodology for writing RUSSEL rules to solve common problems has been proposed in [32]. In this section, we present various approaches to solve a problem encountered while developing honeytanks: finding all non-overlapping occurrences of a signature in an event flow.

The following simple example explains this problem. We want to find all occurrences of the signature **A**, **A** in an event flow. That is, we want to find all occurrences of the situation where an event **A** is followed by another event **A**. There can be other events between the two **A**, but there can be no other **A** in-between. Let the sequence of events [**A**, **B**, **A**, **A**] be the event flow. We should find three occurrences : the first event with the third one, the first with the fourth one and the third event with the fourth one

However, it is common to add a constraint to this search pattern by searching only the **non-overlapping** occurrences of a signature. This is useful when the signature represents a sequence of related events like a session or a transaction. In this situation, a new occurrence of the signature can appear only if events forming the previous occurrence of the signature have already appeared in the event flow. To correctly count the number of occurrences in the event flow, we should consider that each event belongs to at most one occurrence of the searched signature. In the given example, it means that we should only find one occurrence of the signature: the first event with the third one. Indeed, since the third event already belongs to an occurrence of the signature, we cannot consider it as the beginning of a new occurrence.

In this example, searching for signature occurrences consists of checking the presence of some events in the event flow. A more complicated example would be to search for a signature of ***correlated*** events, i.e., events that have some attributes in common. Those attributes act like an identifier for the signature occurrence. For example, let us assume that the events have an attribute named `user` which represents the name of the user who produced this event. We would like to find all occurrences of the signature **A**, **A** where the value of the `user` attribute of the second **A** is the same as the value of the `user` attribute of the first one. In this case, while all

events of a signature occurrence have not been found in the event flow, a new occurrence of the signature **with the same identifier** (here, the same user) cannot be found in the event flow.

In the context of honeytanks, this search pattern is necessary to correctly track TCP sessions. A TCP session is identified by a 4-uple composed of the source and destination IP addresses and the source and destination TCP ports. Since this 4-uple is an identifier, a "correct" TCP segment belongs by definition to at most one TCP session. Thus, to correctly track TCP sessions, we must verify that the events are correctly correlated through the same 4-uple identifier and we must be able to determine if a given TCP segment belongs to an existing TCP session or if it is not part of an opened TCP session.

Let us consider the following example. The event stream is made of the events sequence [`SYN`, `SYN+ACK`, `ACK`, `SYN`], where the three first events form a valid TCP 3-way handshake and the last event is the same as the first one. Since the 3-way handshake is valid, this TCP session is considered as opened and the last event has consequently the same 4-uple identifier as an existing TCP session. Thus, the analysis process must consider this last `SYN` segment as being part of an opened TCP session and not as the beginning of a new one. Once we know it is part of an existing session, we can conclude that the host which sends this segment is misbehaving as it is prohibited to send a `SYN` segment inside a session after the completion of the 3-way handshake.

We can now give the invariant of the RUSSEL program which finds all non-overlapping occurrences of a given signature. For each of the n (with $n > 1$) events E_1, ..., E_n composing the signature, we define a RUSSEL rule named `find_i` ($1 \leq i \leq n$). The task of the rule `find_i` is to find the event E_i with the correct identifier in the event flow. At each ASAX cycle, at most one instance of this rule exists for each signature occurrence partially found at this moment. For a given signature occurrence, when a rule instance of `find_i` ($1 < i \leq n$) exists, there is no rule instance of `find_j` ($j \neq i, j < i \leq n$) searching for events of the same signature occurrence active at the same time.

We also define a RUSSEL rule named `find_1` that is in charge of finding the beginning of a new signature occurrence. It must not only find occurrences of the event E_1 but it also ensures that this event is not part of an existing signature occurrence. It means that the identifier of E_1 must not be already used by an existing signature occurrence, i.e., that there is no rule instance searching for events with the same identifier. At each ASAX cycle, there is exactly one instance of the rule `find_1`, no matter the number of signature occurrences present in the event flow.

At a given ASAX cycle, if the event E_1 is found and if it is part of a new signature occurrence, then at the next ASAX cycle, an instance of the rule `find_2` searching for the event E_2 part of this new signature occurrence

will exist. Similarly, at a given ASAX cycle, if the event E_i $(1 < i < n)$ is found and if it is part of an existing signature occurrence, then at the next ASAX cycle, an instance of the rule `find_i+1` searching for the event E_{i+1} part of the same signature occurrence will exist. Finally, if the event E_n of an existing signature occurrence is found during a given ASAX cycle, no occurrence of the rules `find_i` $(1 < i \leq n)$ searching for events of the same signature occurrence will exist at the next ASAX cycle.

As we can see, the difficulty of solving this problem lies in the fact that the rule `find_1` must know if the current event will be processed by another rule instance, i.e. a rule instance searching for events with the same identifier as the current event. The remainder of this section exposes various techniques allowing a rule instance to detect if another rule instance already exists.

7.1.1 Detecting the presence of another rule instance

The intuitive, but incorrect, approach would be as follows. When the rule instances `find_i` $(1 < i \leq n)$ find that the current event is part of the signature occurrence they follow, they set a global boolean variable to true. If the rule `find_1` finds that the current event might be the beginning of a new signature occurrence, it tests the value of this global variable. If it is true, it means that the current event is already part of an existing signature occurrence and that another rule instance will handle it. This solution is only valid if the rule `find_1` is executed after all other rule instances. Otherwise a value might be assigned to the global variable after it has been tested by the `find_1` rule. As no assumption can be made on the execution order of the rule instances, we cannot use this approach as it is. We propose several corrections to this approach.

Waiting for the execution of the last instance

A solution consists of waiting for the execution of the last rule instance to decide if a rule instance already handled the current event. Figure 7.1 shows the pseudo-code of this solution.

Each time a new signature occurrence is found and a rule is triggered to find the remainder of this occurrence, the global variable `nbInstances` is incremented. This variable is decremented when the last event of the signature occurrence is found. Moreover, each instance of `find_i` increments the `nbExecuted_Instances` variable when it is executed. Then, if it handles the current event, it sets the variable `event_handled` to 1.

At the end of its execution, the instance compares the value of the variable `nbExecuted_Instances` with the value of the variable `nbInstances`. If they are equal, it means that this instance is the last to be executed. It can then check the value of the `event_handled` variable. If it is equals to 1, it

means that another rule instance already handled the current event. Otherwise, the rule `find_1` must be triggered to handle the current event. Then, the variables `nbExecuted_Instances` and `event_handled` can be reinitialized to zero.

The main disadvantage of this method is that the roles of the rules are no more clearly separated. Each rule instance must test if it is the last instance and check if the current event is the first event of the signature. Moreover, as we will see in the section 7.2, the technique used cannot benefit from the optimized version of ASAX.

Delay the decision by one event

We have seen that it is difficult for rule instances to communicate with each other because no assumption can be made on their execution order. The previous solution consists in waiting for the execution of the last rule instance to decide if a rule instance already handled the current event. We propose another solution that consists in waiting for the next ASAX cycle to find if a rule instance active during the previous cycle handled the previous event.

The general idea is the following : when `find_1`, the rule instance searching for the first event of a new signature occurrence, finds such an event, it does not immediately consider that it is the beginning of a new signature occurrence. Indeed, it is possible that this event is in fact already part of an existing signature occurrence and that another rule instance will handle it. The rule instance `find_1` must wait for the execution of all other rule instances to take a decision about the current event. We propose to delay this decision to the next ASAX cycle. When the rule instance `find_1` finds a possible start of a new signature occurrence, it triggers for the next event a rule instance `delayed_find_1` and passes the current event as parameter. Meanwhile, the other rule instances set a global boolean variable to true if they handled the current event as being part of an existing signature occurrence. At the next ASAX cycle, the rule instance `delayed_find_1` can check the value of the global boolean variable. If it has the true value, it means that the previous event was part of an existing signature occurrence and that a rule instance active during the previous ASAX cycle already handled it. Otherwise, it means that the previous event was the beginning of a new signature occurrence. As the rule instance `delayed_find_1` has the previous event passed as parameter, it can analyze and handle it and trigger for the current event the rule that will search for the next event of the signature. In all cases, `delayed_find_1` resets the value of the global boolean variable to false.

However, this solution is correct if and only if the rule instance `delayed_find_1` is executed before all other rule instances. Indeed, the global boolean variable must be reset to false before another rule instance possibly set it to

```
global nbInstances, nbExecuted_Instances, event_handled : integer;

init_action;
begin
    nbInstances := 0; nbExcuted_Instances := 0; event_handled := 0;
    trigger off for_next find_1
end;

rule find_1;
begin
    if <this event is the first event of the signature> -->
      begin
          trigger off for_next find_2(ID);
          nbInstances := nbInstances + 1
      end;
    nbInstances = 0 --> trigger off for_next find_1
    fi
end;

rule find_i (with 1 < i < n) (ID);
begin
    nbExecuted_Instances := nbExecuted_Instances + 1;
    if <The ID of the current event is = ID> -->
       begin
           event_handled := 1;
           if <The event is the expected one> -->
              trigger off for_next find_i+1(ID);

              true -->
              begin
                  nbInstances := nbInstances - 1;
                  nbExecuted_Instances := nbExecuted_Instances - 1;
                  <Report errors>
              end
           fi
       end;
       true --> trigger off for_next find_i(ID)
    fi;

    if nbExecuted_Instances = nbInstances -->
      begin
         if event_handled = 0 -->
            trigger off for_current find_1
         fi;
         event_handled := 0;
         nbExecuted_Instances := 0
      end
    fi
end.
```

Figure 7.1: RUSSEL pseudo-code for finding non-overlapping occurrences of a signature

true again. But, as we already explained, the execution order of the rule instances cannot be guaranteed. The solution is to use two global variables and use them in alternate order. If we number the events, the first global variable is set to true when an even event is part of an already existing signature occurrence and the second global variable is used for the odd events. All rules have an additional parameter indicating if the current event is odd or even. We could also have used the format adapter to add this information to each event.

Figure 7.2 shows the pseudo-code of the solution. RUSSEL does not have a boolean type, so we define two global integer variables `flag0` and `flag1`. Their invariant is that `flag0` has the value 1 if the previous event was an even event and if it was part of an already existing signature occurrence and thus handled by a rule instance during the previous ASAX cycle. It has the value 0 otherwise. The `flag1` global variable has the same invariant excepted that it concerns odd events. The `init_action` rule initializes those two variables to zero and trigger for the next event an instance of the rule `find_1`. Its actual parameter has the value 1 because the next event is the first one and thus is an odd event.

The `find_1` rule tests if the current event is the first event of the searched signature. If it is the case, it triggers for the next event an instance of the rule `delayed_find_1` with the current event and its rank (odd or even) as parameter. Then, the rule re-triggers itself. The rule `delayed_find_1` tests the value of the global variable `flag0` or `flag1` according to the rank of the previous event that has been passed as parameter. If its value is 1, it means that the previous event was part of an existing signature occurrence and that a rule instance handled it during the previous record. The only thing the rule has to do is reset the global variable to zero. If it is not the case, it means that the previous event was the first event of a new signature occurrence. The rule then triggers off the current event an instance of the rule `find_2` which will test if the current event is the next event in the signature. The parameters for this new instance are the rank of the current event (which is the opposite of the rank of the previous event) and the identifier of the signature (e.g. the 4-uple identifier of a TCP session).

Finally, several rules `find_i` with $1 < i \leq n$ are defined for the other events forming the remainder of the signature. If the current event does not match the identifier of the signature, the rule re-triggers itself. Otherwise, it means that the current event belongs to the signature occurrence that this rule is following. Thus, the rule set to true the global variable `flag0` or `flag1` according to the rank of the current event. Then, if the current event is the one that the rule expects, it triggers for the next event a rule which will search for the next event of the signature. Otherwise, it means that the current event is unexpected and the rule reports an error.

This solution suffers from two problems. First, in the pseudo-code, we pass the current event as a parameter of the rule `delayed_find`. However, it

```
global internal flag0, flag1 : integer ;

init_action;
begin
    flag0 := 0; flag1 := 0;
    trigger off for_next find_1(1)
end;

rule find_1(p : integer);
begin
    if <this event is the first event of the signature> -->
      trigger off for_next delayed_find_1(p, event);
    fi;

    trigger off for_next find_1(1-p)
end;

rule delayed_find_1(p : integer; previous_event);
if
    p = 0 -->
    if
      flag0 = 1 --> flag0 := 0;
      true --> trigger off for_current find_2(1-p, <previous event ID>)
    fi;

    true -->
    if
      flag1 = 1 --> flag1 := 0;
      true --> trigger off for_current find_2(1-p, <previous event ID>)
    fi
fi;

rule find_i (with 1 < i <= n) (p : integer; ID)
if
    <The ID of the current event is != ID> -->
    trigger off for_next find_i(1 - p, ID);

    true -->
    begin
        if p = 0 --> flag0 := 1 ; true --> flag1 := 1 fi;
        if
            <The event is the one we search> -->
            trigger off for_next find_i+1(1-p, ID);

            true --> <Report errors>
        fi
    end
fi.
```

Figure 7.2: RUSSEL pseudo-code for finding non-overlapping occurrences of a signature

is not possible to pass an event as a parameter in RUSSEL. To simulate this, all the relevant attributes of the current event must be passed one by one as parameters of the rule. This is a fastidious and error-prone task. Second, the technique used is not trivial. The programmer must be very cautious to use correctly the two global variables in alternance and to pass the rank of the current event as parameter to the rules. An error in this code is very difficult to find. To overcome those two problems we propose two solutions explained in the remainder of this section : the frozen variables and the oracle.

Frozen variables

To avoid the use of two global variables and the constraint to pass the rank of the current event as a parameter to the rules, we introduce the concept of *frozen* variables. A frozen variable is made of two parts : a read-only and a write-only area. During an ASAX cycle, all read access to a frozen variable return the content of the read-only area. Besides, all write access to a frozen variable change the value contained in the write-only area. At the end of an ASAX cycle, the value contained in the write-only area is copied into the read-only area and the value of the write-only area is reinitialized to zero. When the programmer uses a frozen variable, he has the guarantee that the value obtained by reading a frozen variable is constant during a whole ASAX cycle. Any assignment done to a frozen variable is only effective at the beginning of the next ASAX cycle. A frozen variable can be seen like a variable for which assignments have a delayed effect.

When using frozen variables, the RUSSEL code for finding non-overlapping occurrences of signature can be dramatically simplified. The programmer must neither handle two global variables nor pass the rank of the current event as a parameter. Moreover, as the write-only area is reset to a default value of zero after each cycle, it means that if there was no assignment to a frozen variable during a cycle, its (read-only) value is automatically reset to zero for the next cycle. Figure 7.3 shows the simplified pseudo-code using frozen variables.

The resulting code is much simpler to write and to understand. However, delaying the process of an event to the next ASAX cycle is not very convenient. The programmer must pass all the relevant attributes of the event one-by-one as parameters. Moreover, the handling of the event is delayed by one cycle. It means that the event will be actually processed when the next event will be available to the ASAX analyzer. If the arrival rate of events is low, the process of the event can be quite delayed. To overcome this problem, we introduce the notion of the oracle.

```
global internal frozen flag : integer ;

init_action;
begin
    flag := 0;
    trigger off for_next find_1
end;

rule find_1;
begin
    if <this event is the first event of the signature> -->
      trigger off for_next delayed_find_1(event);
    fi;

    trigger off for_next find_1
end;

rule delayed_find_1(previous_event);
if
    flag = 0 --> trigger off for_current find_2(<previous event ID>)
fi;

rule find_i (with 1 < i <= n) (ID)
if
    <The ID of the current event is != ID> -->
    trigger off for_next find_i(ID);

    true -->
    begin
        flag := 1;
        if
            <The event is the one we search> -->
            trigger off for_next find_i+1(ID);

            true --> <Report errors>
        fi
    end
fi.
```

Figure 7.3: RUSSEL pseudo-code for finding non-overlapping occurrences of a signature using frozen variables

The oracle : seeing in the future

If we assume that we have an oracle that can accurately predict which event will come on the next cycle, the design of the rules can be further simplified and delaying the detection of a new signature occurrence is no more needed. The general idea is as follows: when we create a rule instance and trigger it for the next event, we also take a look at the future event. If the next event is such that it will be handled by the rule instance we just created, we then know that this future event belongs to an existing signature occurrence. Consequently, we can send a message (through a frozen variable) to the rule instances which will be active during the next cycle, notifying them that the event they analyze is not the beginning of a new signature occurrence.

Figure 7.4 shows the simplified pseudo-code using an oracle for finding non-overlapping occurrences of a signature. The invariant of the `flag` variable is that its read-only area has the value 1 if, during the previous cycle, a rule instance detected that the event available at the current cycle is part of an already existing signature occurrence and that a rule instance will handle it. Otherwise, it means that the current event is the beginning of a new signature occurrence.

The rule `find_1` is greatly simplified. All it has to do is to check the value of the variable `flag`. If it has the value 0, the rule triggers a rule which will search for the remainder of the signature. Then if the next event will be handled by the rule instance it has just triggered, it sets the `flag` variable to 1. The rules `find_i` must be slightly modified. Each time a rule instance is triggered for the next event, the triggering rule must verify if the next event will be handled by the rule instance it has just triggered. If it is the case, it assign the value 1 to the `flag` variable. Since it is a frozen variable, it is reset to zero at the beginning of a cycle if it has not been written during the previous cycle.

To implement an oracle able to foresee the future, we must modify the format adapter. It will not directly convert the events in NADF anymore. Instead, the conversion will be delayed until a second event is available. So, the format adapter will permanently hold two events in memory. When it converts the first one into an NADF record, he can also give information about the next record it is holding back. The second event will only be converted when a third event is available in order to give information about this future event.

With this technique, the RUSSEL rule trying to find the beginning of a new occurrence does not have to delay its analysis anymore or pass the attributes of the event as parameters to another rule. Instead, the delay is done inside the format adapter which appears as able to predict the future event from the rule instances's point of view. The drawback is that the format adapter must wait to have two events in memory to convert the first one in NADF and give it to the analyzer. If the arrival rate of events is low,

```
global internal frozen flag : integer ;

init_action;
begin
    flag := 0;
    trigger off for_next find_1
end;

rule find_1;
begin
   if <this event is the first event of the signature> -->
      if flag = 0 -->
         begin
            trigger off for_next find2(<current event ID>);
            if <next event ID == current event ID> --> flag := 1 fi
         end
      fi
   fi;

   trigger off for_next find_1
end;

rule find_i (with 1 < i <= n) (ID)
begin
    if <The ID of the current event is != ID> -->
      begin
          trigger off for_next find_i(ID);
          if
              <The ID of the next event is = ID> --> flag := 1
          fi
      end;

      true -->
      if <The event is the one we search> -->
         begin
             trigger off for_next find_i+1(ID);
             if <The ID of the next event is = ID> --> flag := 1 fi
         end;

         true --> <Report errors>
      fi
    fi
end.
```

Figure 7.4: RUSSEL pseudo-code for finding non-overlapping occurrences of a signature using an oracle

the conversion of the second event might be quite delayed. To overcome this problem, the oracle can generate *fake* events. The events produced by the event source are called real events. The events artificially created by the format adapter are called fake events and contain an attribute identifying them as such.

Algorithm 2 An oracle format adapter using fake events

At initialization:
$first\_event$ ← create a fake event

When the analyzer asks for a new event:
if <event is available from event source> **then**
  second_event ← <get event from event source>
else
  second_event ← <create a fake event>
end if

nadf ← <convert $first\_event$ and give information about *second_event* >
$first\_event$ ← *second_event*
return *nadf*

The algorithm 2 shows the algorithm of an oracle format adapter using fake events. Fake events are used when the format adapter has only one real event in memory and cannot convert it immediately because the event source has not yet another event. In this case, the format adapter nevertheless converts the real event and announces that the next event will be a fake one. Then, it will convert the fake record into NADF. If there are still no events available from the event source, it will announce that the next event is a fake one, otherwise it will give information about the incoming real event.

7.1.2 Using timeouts to improve performance

As we have seen, the execution time of an ASAX cycle is a function of the number of rule instances to execute. While searching for occurrences of signatures, the number of instances is equal to the number of occurrences existing in parallel in the event flow. When the beginning of an occurrence has been found, a rule instance is created to find the next event of the signature. Even if this next event never occurs in the event flow, this rule instance will nevertheless live until the end of analysis and be executed for each event. If there are many such unfinished occurrences in the event flow, the execution of those *useless* instances can impact the performance.

One solution is to use a timeout mechanism. When a rule instance is

created, it receives a time stamp as a parameter. This time stamp is the time at which the rule should discard itself if the searched event has not been found at this moment. To compute this time stamp, the triggering rule retrieves the current time through the call of the C-function `gettimeofday()` and adds the timeout value to it. When a rule instance is not interested in the current record it does not re-trigger itself immediately. It retrieves the current time and compares it with its expiration time stamp passed as a parameter.

This solution allows to kill useless rules instances and thus improves the execution time of the analysis. However, the timeout value must be chosen carefully. If it is too high, the useless instances will live longer and the performance will degrade. If it is too low, the instances will be killed too early and the analysis will not correctly find all signature occurrences.

7.2 Writing rules for the optimized version of ASAX

The previous section showed various techniques to write correct rules for the problem of finding non-overlapping occurrences of a signature. In this section we introduce the optimized version of ASAX and explain how to transform the rules presented in the previous section to benefit from those optimizations.

Most RUSSEL rules follow the same pattern : they check if they are interested in the current event. If it is the case, they perform some actions. Otherwise, they re-trigger themselves. Most of the time, only one rule instance is interested in the current event. It means that amongst all rules instances of a given ASAX cycle, only one instance will perform some actions while the other will simply re-trigger themselves. In the non-optimized version of ASAX, all rule instances are systematically executed regardless to the fact that they are interested or not in the current event.

The optimized ASAX aims at detecting at each cycle which rules instances will re-trigger themselves without producing any side-effect. Those rule instances are directly moved in the `for_next` bag without being executed. It means that, in the best case, only the rule instance interested in the current event is executed. The duration of an ASAX cycle does no longer depend on the number of instances and is constant. Moreover, identical conditional instructions that are present in several rules are merged and executed only once for all the instances. However, to fully benefit from those optimizations, some constraints must be satisfied when writing RUSSEL rules. In the remainder of this section, we present the general principle used by the optimizations and explain how to write optimizable rules. However explaining the inner working of the optimizations is out of the scope of this thesis. We invite the reader to refer to [28] for a detailed explanation of the optimizations implementation.

7.2.1 Introducing the optimized ASAX

The optimizations try to determine at each cycle which rule instances must be executed and which rules can be safely moved to the `for_next` bag without being executed. In order to achieve this, each rule is abstracted by a binary decision tree where an internal node represents a conditional instruction of the rule and a leaf node represents the decision to either execute the rule, re-trigger it or discard it. By evaluating the conditions in the BDT according to the current environment, the optimizations take a decision (execute, re-trigger or discard) which is valid for many instances of the rule. In the best case, a re-trigger decision is taken which is valid for all the rule instances but the one which must handle the current event. The analyzer actually executes only this instance, the others being all at once re-triggered. The global execution time is thus speeded up. We first present how the BDT is constructed. Then, we explain how it is used.

To abstract a rule into a BDT, the following algorithm is used to perform a post-order traversal of the abstract syntax tree of the rule.

- If the abstract node represents a conditional instruction, the actions depends on the type of conditional instruction encountered. There are three types of them. First, conditional instructions for which the result of the evaluation is the same for all rule instances. Those conditions only compare *constant elements*, i.e. elements which have a constant value during a cycle : literals, event attributes, global frozen variables and call to an external C function. The latter, when used in a conditional instructions, are assumed to not produce any side-effect and constant behavior during a whole cycle. It is the responsibility of the programmer to ensure that C functions called in conditional instructions satisfy those constraints.

  Second, conditional instructions which compare the value of rule instance parameters with *constant elements*. The result of the evaluation of such conditions is the same for all rule instances which share the same values for the tested parameters.

  Third, all other conditional instructions. The evaluation of this last type of condition can potentially give a different result for each rule instance. This is the case for example of a condition which tests the value of a local variable.

  If the conditional instruction is of the third type, an "execute" leaf node is created and returned. Indeed, the result of the evaluation of this kind of conditions varies from one instance to another. To know this result for each instance, we have no choice but execute one by one all rule instances. Otherwise, an internal node containing the condition is created and returned, with its left and right children being the result

of a recursive call performed respectively on the left and right children of the abstract node.

- If the abstract node represents an instruction which does not produce any side-effects (e.g. an assignment to a local variable), it is skipped and the result of the recursive call on the child node is returned.

- If the abstract node represents an instruction producing side-effects (e.g. assignment to a variable, triggering another rule, call to an external C procedure, ...), an "execute" decision is returned.

- If the abstract node represents a self-trigger instruction, this fact is memorized and the result of the recursive call on the child node is returned. A rule instance is considered to re-trigger itself if and only if it uses as they are the symbols of its formal parameters as effective parameters for the re-triggered instance.

- If the abstract node is empty, it means that we have reached the end of a branch of the abstract syntax tree. If we have encountered a self-trigger during the traversal of this branch, we return a "self-trigger" decision, otherwise we return a "discard" decision.

Once a BDT has been constructed for each rule, it is browsed at each cycle to determine which instances of the rule must be executed, self-re-triggered or discarded. At the beginning of the traversal, a set containing all instances of the rule is associated to the root of the BDT and the following algorithm is used to evaluate this first node:

If the node is an internal node with a condition on constant elements, the condition is evaluated according to the current environment. If the evaluation returns a true (respectively, false) value, then the set of instances associated to the current node is associated to the left (respectively, right) child, and a recursive call is done on this child.

If the node is an internal node with a condition of equality (respectively, non-equality) between a constant element and a rule parameter, the constant element is evaluated according to the current environment. Then, a hash table is used to retrieve all rule instances for which the given parameter has the same value as the constant element of the condition. The instances who are both in the set associated to the current node and the set returned by the hash table are removed from the former set and associated to the left (respectively, right) child of the node. The instances remaining in the set associated to the current node are associated to the right (respectively, left) child of the node. Then, a recursive call is done on both children, so the traversal of both branches is done. If the node is an internal node with a condition of inequality between a constant element and a rule parameter, then the condition is evaluated for each instance associated with the current

node according to the value of its parameters. The instances for which the evaluation returns a true (respectively, false) value are associated to the left (respectively, right) child of the node. Then, a recursive call is done on both children.

If the node is a leaf node with a discard decision, the rule instances associated to the node are destroyed. If it is a re-trigger decision, the instances are moved to the `for_next` bag and if it is an execute condition, the instances are executed by the analyzer.

7.2.2 Writing optimizable rules

Not all kind of rules can benefit from the optimizations. They are useful only if they can find an execution path in the rule leading to a self-re-trigger action. Moreover, that path must be taken for most events present in the event flow. Typically, rules which benefit the most from the optimizations follow this pattern: the rule tests if it is interested in the current event. If it is the case, it performs some actions. Otherwise, it just re-triggers itself. In order to be fully optimizable, some pitfalls must be avoided when writing rules following this pattern.

- The rule must not perform side-effects actions before testing the current event. If it is the case, the optimizations will conclude that the rule must be systematically execute and the abstracted BDT of the rule will be a single 'execute' node.
- When a rule is not interested in the current event and re-triggers itself, it must not perform side-effects actions in the branch leading to the self-re-trigger action. If it is the case, the abstracted BDT of the rule will be a conditional node with its left and right children being 'execute' nodes. So, no matter what the current event is, the rule will be executed.
- When a rule tests the current event, it must not compare it with local variables. As explained before, a conditional instruction testing local variables produces an 'execute' node in the BDT.
- When a rule tests the current event and compares it to a rule parameter, it has to avoid using an inequality test. In this case, the inequality condition will be evaluated for each instance of the rule. The instances which are not interested in the current event will then be re-triggered all at once. If they are many instances, evaluating the condition for each of them can be time-consuming and the benefit of re-triggering many instances at once can be lost.
- When a rule tests the current event and compares it with an (non-)equality test to a rule parameter, a performance issue can appear.

The instances for which the tested parameter has the same value as the constant element of the condition are retrieved through a hash table. Then, those instances are compared to the instances associated with the internal node of the BDT. This comparison is a $O(n \times m)$ operation, where n is the number of rule instances associated to the internal node and m is the number of rule instances returned by the hash table. If the values of the parameter used in the condition are not randomly distributed amongst rule instance, m will be large thus degrading the performances.

Optimizing the timeout mechanism

The timeout mechanism presented in section 7.1.2 is not optimizable. Indeed, when a rule instance is not interested in the current record it does not re-trigger itself immediately. It retrieves the current time through the call of the C-function `gettimeofday()` and compares it with its expiration time passed as parameter. This approach suffers from two problems.

First, as the `gettimeofday()` function returns the current time, the returned value will not be the same for all rule instances. However, the optimizations require that an external C function used in a conditional instruction returns the same value for all instances during a cycle. Second, the test used is an inequality test between a rule parameter and a constant element. It means that the condition will be evaluated for each rule instance associated with this node. In this case, they are instances which are not interested in the current event, i.e. typically all but one instance. This test will thus be very inefficient.

The solution to this problem requires to modify the format adapter. It must give two new pieces of information to the analyzer: all events must be sequentially numbered and must contain an attribute which is the average execution time so far of an ASAX cycle. When a rule with a timeout mechanism is triggered, we compute as usual the time stamp when it will be in timeout. We also compute the event number when this timeout should be reached. Since we know the average duration of a cycle and as it is rather stable over time, the amount of events to wait before the timeout is reached is estimated by dividing the period of time to wait by the duration of the average cycle. By adding this value to the current event number we obtain the estimated event number when the rule instance should be in timeout.

The pattern of the rule is as follows. It first tests if it is interested in the current event and in this case performs some actions. Otherwise, it tests if the current event number equals the estimated event number passed as parameter. This test is a conditional test between an attribute of an event and a rule parameter. Moreover, the values of this parameter are likely to be randomly spread amongst rule instance. Thus, it is fully optimizable. If the current event number is not the estimated event number, the rule simply

re-triggers itself. Otherwise, it means that the rule should be in timeout. Thus, it precisely tests if the timeout has been reached by comparing the current time with its expiration time passed as parameter. If the expiration time has been reached, the rule discards itself. Otherwise, it means that the estimation of the amount of events to wait before testing the timeout was too short. The rule thus computes a new estimation and re-triggers itself.

With this technique, the rule instances do not check anymore at each event in order to see if they are in timeout and can thus be directly re-triggered. It is guaranteed that a rule instance will not discard itself before its timeout. However, it might discard later than expected if the execution time of a cycle is too fluctuating or if the flow of events is irregular. However, the fake record generation introduced earlier can be used to regulate the flow of incoming events.

Optimizing tests on a parameter with unbalanced values across instances

As explained earlier, a (non-)equality test on a parameter might lead to an efficiency issue if there are many rule instances for which the value of the tested parameter satisfy the test.

If this kind of test is done in the execution branch taken when the rule instance is interested by the current event, a simple solution is to force the non-optimization of this test. The value of the parameter must be copied into a local variable and the test must check the value of this local variable instead of the value of the parameter. This way, this test will not be optimized and will be abstracted by an 'execute' decision. As this test occurs in the branch taken by the rule instance when it is interested in the current event, it will anyway perform sooner or later side-effects actions which would also have produced an 'execute' decision. As this rule instance would anyway have been executed, this technique does not impact the execution performance.

This efficiency issue can also appear when the test is done to determine if the rule instance is interested in the current event. For example, rules which track TCP sessions usually have as parameters the 4-uple identifier of the tracked TCP session. To determine if they are interested in the current event, they check if it has the same identifier as the one passed as parameter. If the event flow contains many parallel TCP sessions opened to the same server, the "TCP destination port" parameter will have the same value for many rule instances, thus leading to an efficiency issue. However, the solution consisting in copying the parameters to local variables and testing them cannot be reused in this situation. Indeed, the parameters are tested by the rule instances in order to determine if they are interested in the current event. When the test is done, there is usually only one rule instance which is actually executed, while the other retrigger themselves. If we copy those parameters to local variables and then test those variables, it will force

the execution of all rules instances.

We then must change the repartition of the parameters values amongst rule instances to solve this efficiency issue. The solution is to consider the identifying parameters as a whole and to stop testing them one after the other. Instead of receiving the identifying parameters, the rule receives as parameter their concatenation. To determine if the current event must be handled by the rule, it tests if the concatenation of the identifying attributes of the current event matches the unique identifier passed as parameter. By regrouping the different identifiers into a larger, unique parameter, we increase the distribution of the values of this parameter amongst rule instances.

Unfortunately, at the time of writing, this solution is not usable due to an incomplete implementation of the optimizations. Indeed, if the concatenation of the identifying attributes of an event is larger than four bytes, it has to be stored in a string variable (integer variables only hold four bytes). However, the optimization of conditional instructions manipulating string parameters is not fully functional We have thus developed a temporary work-around by using a hash of the identifying parameters and a stub rule. Figure 7.5 shows the modified code using stub rules. In addition to the identifying parameters, the stub rule receives as parameter a hash of those parameters. This hash is stored in a four bytes integer location. To determine if the current event must be handled by the rule, the stub rule tests if the hash of the identifier of the current or the next event equals the hash passed as parameter. If it is not the case, the rule re-triggers itself. Otherwise, it triggers for the current event the complete version of this rule. The complete rule will compare as usual the identifier of the events with its parameter as to ensure that there was no collision. Since this rule instance was triggered for current, those tests are not optimized and the performance issue does not appear. If the complete rule decides to re-trigger itself or to trigger another rule, it must instead trigger the stub version of the rule. By hashing a set of identifying parameters, we increase the distribution of values across the instances thus limiting the risk of having many instances with the same parameter value.

7.3 On the efficiency of parallel programming with ASAX: comparison with Scapy

Scapy [3] is an interactive network packet manipulation tool. It can be used as an interactive command-line tool or it can be imported as a module in a Python [66] program. Scapy offers a pleasant API to capture packets, decode and explore their layers, change their contents and inject them on a network. It is a versatile and programmable tool that allows the creation of various types of network applications. Our framework offers the same versatility as Scapy.

```
global internal frozen flag : integer;

init_action;
begin
    flag := 0;
    trigger off for_next find_1
end;

rule find_1;
begin
    if <this event is the first event of the signature> -->
      if flag = 0 -->
      begin
          trigger off for_next stub_find2(<hash of current event ID>,
                                          <current event ID>);
          if <next event ID == current event ID> --> flag := 1 fi
      end
    fi;

    trigger off for_next find_1
end;

rule stub_find_i (with 1 < i <= n) (hash_of_ID, ID)
if <The hash of the current event ID is = hash_of_ID> or
  <The hash of the next event ID is = hash_of_ID> -->
  trigger off for_current find_i(ID);

  true --> trigger off for_next stub_find_i(hash_of_ID, ID)
fi;

rule find_i (with 1 < i <= n) (hash_of_ID, ID)
if <The current event ID = ID> -->
  if <The event is the expected one> -->
     begin
         trigger off for_next stub_find_i+1(hash_of_ID, ID);
         if <The ID of the next event is = ID> --> flag := 1
     end;
     true --> <Report errors>;
  fi;

  <The next event ID = ID> -->
  begin
      trigger off for_next stub_find_i(hash_of_ID, ID);
      flag := 1
  end
fi
```

Figure 7.5: RUSSEL pseudo-code for finding non-overlapping occurrences of a signature using stub rules

In this section, we solve a simple yet realistic problem using a program written with our framework and a program written with Scapy. We present and compare the source code of both program and we evaluate the performance of each solution.

7.3.1 Finding flows in a captured trace

For this comparison, the problem we want to solve is finding all flows present in a network capture file and, for each flow, printing the number of packets it contains. In this context, we define a flow as a set of packets having the same IP and TCP source and destination.

For example, if a TCP session is established between a host `A` and a host `B`, there will be two flows: the flows of packets sent from `A` to `B` and the flow of packets sent from `B` to `A`. In this case, the first flow contains 2 packets corresponding to the `SYN` and `ACK` segments of the 3-way handshake. The second flow contains one packet corresponding to the `SYN+ACK` segment of the 3-way handshake.

The Scapy solution

Listing 7.1 presents the source code of the program solving this problem using Scapy. Lines 21 to 33 parse the command-line arguments and retrieve the name of the capture file to analyze. At the line 26 a new object of the type `PcapReader` is instantiated on the capture file. This object offers an interface to retrieve packets from a capture file. Then, the `track` function is called with the `PcapReader` object as a parameter.

The `track` function is defined at lines 5 to 18. First, it creates a new, empty, dictionary called `flow_dict`. Then, the function iterates on all the packets present in the network trace. At each iteration, the following actions are executed. In lines 8 to 10, the function tests if the packet is a TCP segment and, if it this the case, extracts its IP and TCP source and destination fields. This 4-uple identifies a flow. Then, the function test if this flow identifier is already present in the diction-nary. If it is not the case it creates the entry and associate to it the value 1 as this is the first packet encountered for this flow. If there already exists an entry in the diction-nary for this flow, its associated value is incremented by 1.

Once the function has iterated over all packets, it prints the contents of the dictionary. This is done by line 17 and 18. For each entry in the dictionary, the function prints the 4-uple identifier of the flow as well as the number of packets it contains.

The ASAX solution

Listing 7.2 presents the source code of the program solving this problem using our framework. Unlike Scapy, the RUSSEL source code does not

Listing 7.1: Count packets exchanged between hosts – Python version

```
1  import sys
2  import scapy
3
4
5  def track(pcap_handler):
6      flow_dict = {}
7      for packet in pcap_handler:
8          if packet.haslayer(scapy.TCP):
9              packet = packet.getlayer(scapy.IP)
10             flow_id = (packet.src, packet.dst, packet.sport,
                   packet.dport)
11
12             if flow_id in flow_dict:
13                 flow_dict[flow_id] = flow_dict[flow_id] + 1
14             else:
15                 flow_dict[flow_id] = 1
16
17     for key, value in flow_dict.iteritems():
18         print key[0], ':', key[2], '->', key[1], ':', key
               [3], ':', value
19
20
21 def main():
22     if len(sys.argv) != 2:
23         sys.exit("Usage: %s file.pcap" %(sys.argv[0]))
24
25     try:
26         pcap = scapy.PcapReader(sys.argv[1])
27         track(pcap)
28     except Exception, err:
29         sys.exit(err)
30
31
32 if __name__ == "__main__":
33     main()
```

Listing 7.2: Count packets exchanged between hosts – RUSSEL version

```
1  global frozen known_flow : integer;
2
3  init_action;
4    trigger off for_next check_packet;
5
6  rule check_packet;
7  begin
8    trigger off for_next check_packet;
9
10   if
11     known_flow = 0 -->
12     trigger off for_current track_flow_opt(0,rawToInt(tcp_session_hash),
           ip_source,ip_dest,tcp_source,tcp_dest)
13   fi
14 end;
15
16 rule track_flow_opt(count,hash:integer;ip_s,ip_d,tcp_s,tcp_d:string);
17 if
18   rawToInt(tcp_session_hash) = hash or
19   rawToInt(next_tcp_session_hash) = hash -->
20   trigger off for_current track_flow(count, hash, ip_s, ip_d, tcp_s,
         tcp_d);
21
22   present last_record -->
23   println(LIBC_inet_ntoa(ip_s), ' : ', rawToInt(tcp_s), '  -->  ',
         LIBC_inet_ntoa(ip_d), ' : ', rawToInt(tcp_d), '  :  ', count);
24
25   true -->
26   trigger off for_next track_flow_opt(count, hash, ip_s, ip_d, tcp_s,
         tcp_d)
27 fi;
28
29 rule track_flow(count,hash:integer;ip_s,ip_d,tcp_s,tcp_d:string);
30 var new_count : integer;
31 begin
32   new_count := count;
33
34   if
35     ip_source = ip_s and ip_dest = ip_d and
36     tcp_source = tcp_s and tcp_dest = tcp_d -->
37     new_count := count + 1
38   fi;
39
40   if
41     next_ip_source = ip_s and next_ip_dest = ip_d and
42     next_tcp_source = tcp_s and next_tcp_dest = tcp_d -->
43     known_flow := 1
44   fi;
45
46   trigger off for_next track_flow_opt(new_count, hash, ip_s, ip_d, tcp_s
         , tcp_d)
47 end.
```

need to open a given network capture file as the file to open is given as a command-line argument and is automatically opened by the format adapter.

The first line defines a global frozen variable named `known_flow`. The invariant of this variable is that it contains the value 1 if the current packet belongs to a previously seen flow. It contains the value 0 if the current packet is not part of an existing flow. We will see later how this invariant is maintained. The `init_action` rule defined in lines 3 and 4 bootstraps the analysis process by creating a new instance of the `check_packet` rule and scheduling its execution for the next – here, the first – packet.

The rule `check_packet` is active for each record. Indeed, its first action is to retrigger itself. Then, it tests if the frozen variable `known_flow` is equal to 0, i.e. if the current packet does not belong to a previously seen flow. If it is the case, a new instance of the rule `track_flow_opt` is created and its execution is scheduled for the current record.

The `track_flow_opt` rule is a stub rule of the `track_flow` rule. The use of stub rules has been explained in Section 7.2.2 of this chapter. The rule `track_flow_opt` is in charge of tracking a flow, i.e. finding all packets belonging to a given flow. Thus, during the execution of the program, there will be one instance of this rule per flow present in the analyzed trace. Moreover, this rule is also responsible for maintaining the invariant of the `known_flow` frozen variable.

The `count` parameter of the `track_flow_opt` stores the number of packets encountered so far in the flow. The `hash` parameter is the result of a hash function on the 4-uple identifier of the flow and the last 4 parameters are this 4-uple identifier. The use of a hash has been explained in Section 7.2.2 of this chapter. If the current or the next packet belong to the monitored flow, the rule triggers for the current packet the execution of an instance of the rule `track_flow`. If the current record notifies that the end of the trace has been reached, then the rule prints the 4-uple identifier of the monitored flow as well as the number of packets it contains. Otherwise, it retriggers itself.

The rule `track_flow` has the same parameters as its stub. If the current packet is part of the monitored flow, then the rule updates accordingly the parameter that stores the number of packets encountered so far in the flow. If the next packet is part of the monitored flow, it set the value 1 to the `known_flow` frozen variable. This way the instance of the rule `check_packet` that will be executed for the next record will known that this record is already part of an existing flow and thus that there is no need to consider it as a new flow.

The approach taken to solve this problem with our framework is totally different from the approach used with Scapy. In this RUSSEL program, the user does not manipulate a dictionary and he does not have to iterate over the data structures. All the information are stored as parameters of the various rule instances. The ASAX analyzer handles the execution of the

various rule instances.

This program is a typical example of an application that can benefit from the optimized version of ASAX. During the execution of this program, one instance of the rule `track_flow_opt` is created per flow. Those instances live until the end of the analyzed network capture file has been reached. Thus, there can be potentially a high number of instances of this rule. With the optimized version of ASAX, only one instance of this rule will be executed for the current record, i.e. the instance monitoring the flow to which the current packet belongs to. The other instances of this rule are retriggered for the next record without being executed.

7.3.2 Performance assessment

We compare the performance of those two programs by comparing their execution time on a given trace. The trace[1] used comes from the Evaluation Data Set of the 1998 DARPA Intrusion Detection Evaluation [25]. It is the file `sample_data01.tcpdump`. The trace is 2,510,362 bytes long and contains 14,523 packets. It contains 590 flows.

The Python program is run[2] by a standard version of Python and by a version using the Psyco [49] just-in-time compiler in order to improve the performance of the execution. The RUSSEL program is run by the standard and optimized version of ASAX. The computer used for this test has an AMD Athlon processor clocked at 1.4 GHz and 640 Mbytes of RAM.

The programs were executed 10 times each and their execution time was computed with the `time` command. Table 7.1 shows the results of those executions. On average, the execution time of the Python program is 23 seconds. When executed with the Psyco JIT compiler, it is 21 seconds on average. The execution time of the RUSSEL program with the standard version of ASAX is 2.2 seconds on average and is 0.3 second on average when executed with the optimized version of ASAX. We verified after each run that the output of all programs were identical. With our framework, the processing of this trace is 65 time faster than the processing performed with Python.

7.4 Conclusion

In this chapter, we presented various programming techniques using RUSSEL to solve the problem of finding non-overlapping occurrences of a signature in an event flow. This problem is encountered in developing honeytanks with ASAX. While this problem can be solved with standard constructs of

[1] The trace can be downloaded at `http://www.ll.mit.edu/IST/ideval/data/1998/1998_data_index.html`

[2] We used Python 2.5.1 and Scapy 1.2.0.2

Run #	Scapy			Scapy + Psyco		
	Real	User	Sys.	Real	User	Sys.
1	22.94	22.85	0.09	21.10	20.87	0.22
2	22.99	22.86	0.13	20.89	20.69	0.20
3	23.17	23.04	0.13	21.11	20.90	0.14
4	23.21	23.06	0.13	21.07	20.91	0.16
5	23.18	23.05	0.14	21.85	21.66	0.18
6	23.06	22.94	0.12	20.68	20.52	0.15
7	22.98	22.76	0.17	20.99	20.82	0.17
8	22.94	22.81	0.12	20.89	20.75	0.13
9	22.94	22.78	0.14	21.23	21.07	0.16
10	23.11	22.98	0.13	21.10	20.94	0.14
Avg.	**23.05**	**22.91**	**0.13**	**21.09**	**20.91**	**0.17**
Run #	ASAX			ASAX opt.		
	Real	User	Sys.	Real	User	Sys.
1	2.23	2.19	0.03	0.34	0.30	0.04
2	2.25	2.20	0.04	0.34	0.32	0.02
3	2.23	2.19	0.04	0.35	0.32	0.03
4	2.23	2.22	0.01	0.36	0.32	0.04
5	2.23	2.18	0.04	0.34	0.30	0.03
6	2.25	2.22	0.03	0.36	0.31	0.04
7	2.25	2.23	0.02	0.34	0.30	0.03
8	2.24	2.22	0.02	0.35	0.31	0.04
9	2.24	2.22	0.02	0.34	0.30	0.04
10	2.23	2.20	0.03	0.34	0.29	0.05
Avg.	**2.24**	**2.21**	**0.03**	**0.35**	**0.31**	**0.04**

Table 7.1: Execution time (in seconds) of equivalents RUSSEL and Python programs

RUSSEL we have introduced some modifications to the ASAX analyzer and to the format adapter which simplify considerably the solution: the *frozen* variables and the *oracle* format adapter. Then, we presented the optimized version of ASAX and showed its general principles. We presented a methodology for writing parallel RUSSEL programs benefiting from those optimizations. The stateful honeytank presented in Chapter 8 heavily relies on this methodology. Finally, we compared our approach with Scapy, a Python library for manipulating network packets and showed that on our example our framework outperforms Scapy in terms of execution time.

Chapter 8

Stateful honeytanks

> Ninety per cent of most magic merely consists of knowing one extra fact.
>
> Terry Pratchett
> *Night Watch*

This chapter presents the design and the implementation of a stateful honeytank. The first implementation of a honeytank presented in the chapter 5 was designed with a performance objective in mind thus leading to the decision to use a stateless design. By exploiting some particularities of the TCP protocol, the stateless honeytank is able to simulate the presence of several hosts and behaves correctly faced to common TCP traffic. However, even if the simulation is able to handle a TCP 3-way handshake and some exchange of data, there are many other aspects of a TCP session it is not able to handle correctly. At the cost of maintaining some state, we showed in chapter 6 that it is possible to reply correctly to fingerprinting tools. We now propose to design and implement a honeytank that maintains all the necessary state to handle a TCP session in a realistic way. Finally, we explain how stateful application simulators can be integrated into a stateful honeytank.

This chapter is organized as follows. Section 8.1 presents the objectives that a stateful honeytank must achieve. Section 8.2 presents the design of a stateful honeytank. It explains what states must be maintained in order to achieve the objectives and presents the architecture of an implementation of a stateful honeytank in our framework. Section 8.3 applies the programming methodology presented in Chapter 7 to build an optimized stateful honeytank. Finally, Section 8.4 explains how to write stateful application simulators by translating stateful Nepenthes vulnerability modules into our framework.

8.1 Objectives of a stateful honeytank

Low-interaction honeypots simulate a subset of the functionality of a real host. Their degree of realism depends on the simulated functionality. We presented in Chapter 5 the stateless honeytank which is efficient but has a low level of realism. However, we believe that our approach and the ASAX system supporting it allows the degree of realism of a honeytank to be as high as required. As a proof of concept, we present a stateful honeytank simulating realistically several aspects related to the handling of a TCP session. In terms of functionality, the objectives of a stateful honeytank are the following.

- **Tracking TCP sessions.** Other honeytank implementations are not able to unambiguously decide if a given packet is part of an existing TCP session. In some cases, this prevent the honeytank from correctly handling TCP 3-way handshakes or the TCP session clearings. The first objective of the stateful honeytank is thus to accurately track TCP sessions in order to identify to which session a given segment belongs.

- **Handling TCP session initiation and clearing phases.** With its ability to track TCP session, the stateful honeytank knows in which state is each TCP session. It will thus correctly handle the 3-way handshake and TCP clearing phases in every situation.

- **Detection of out of order TCP segments.** The sequence number present in TCP segments is used to detect lost, duplicated or desequenced segments. The stateful honeytank will detect retransmitted and desequenced segments in the data exchange phase of a session. However, we will not implement their resequencing.

- **Retransmission of `SYN+ACK` segments.** As TCP is a reliable protocol, segments which are not acknowledged in time must be retransmitted. We did not implement this part of the specification. However, to prove that a retransmission mechanism is feasible, the stateful honeytank will retransmit `SYN+ACK` segments which are not acknowledged in time during the 3-way handshake phase.

- **Supporting NMap OS fingerprinting scans.** Finally, we do not want to lose features already present in the other honeytank implementations. Our last objective is thus to still reply in a realistic way to fingerprinting tools.

Many important aspects of TCP are not taken into account in this implementation. For example, the receive and send window size are not managed and the TCP options are not handled. However, the stateless honeytank

is already sufficient to interact correctly with several hosts and to collect data. As the stateful honeytank is more accurate in its simulation than the stateless honeytank, this implementation is also good enough to exchange some data with other hosts.

8.2 Design and implementation of a stateful honeytank

To fulfill the objectives presented in the previous section, the honeytank must maintain a list of opened TCP sessions and maintain information about each session and each emulated host. To achieve this, we propose a design based on two kinds of RUSSEL rules. First, a rule is responsible for maintaining all state related to an emulated host. This rule also emulates all aspects of a host which are not related to simulating an opened TCP session (handling ICMP requests, replying to NMAP tests, ...). Second, several rules are responsible for handling all facets of a TCP session from the 3-way handshake and the data exchange phases to the TCP session clearing. Moreover, those two kinds of rules must exchange information with each other. For example, the rule simulating a host must know if the current packet will be processed by a rule simulating an opened session or if it must process the packet by itself.

For each kind of rule, we present the information that must be maintained and then present the design and the implementation allowing to store this information.

8.2.1 Simulating a host

To simulate the behavior of a host (excluding all aspects related to simulating a TCP session), a stateful honeytank must maintain the following information for each simulated host:

- **The IP address.** To decide if the host is interested in the current packet, it must know its own IP address.
- **The state of TCP and UDP ports.** When the host receives an UDP or TCP segment, its reaction depends on the opened or closed state of the destination port.
- **The services running.** The host must know which service will handle the data carried by TCP or UDP segments sent to opened ports.
- **The type of TCP/IP stack.** To reply correctly to an NMap fingerprinting scan, the honeytank hast to know the type of the TCP/IP stack of the simulated host and how to reply to the NMap scans.

- **The last Initial Sequence Number.** As explained in Chapter 6, the host must maintain the last ISN chosen to reply correctly to NMap scans.
- **Activity timestamps.** By maintaining a timestamp of the last packet received, a host can discard itself if it has not been contacted for a long time.
- **The number of active TCP sessions.** The number of active sessions can also be used to discard unused hosts.

To maintain this information, we propose the following design. For each type of simulated TCP/IP stack, we define a rule named `emulate_host_X` where `X` is the type of the TCP/IP stack. This rule simulates the behavior of a host having a given `X` stack. In the network simulated by a honeytank, multiple hosts can have the same type of stack. Thus, for a given type of stack, one instance of the corresponding `emulate_host_X` rule is created per simulated host bound to this type of stack.

Information maintained per rule. With this design, the type of the TCP/IP stack as well as how to reply to the NMap tests is embedded in the RUSSEL code of each `emulate_host_X` rule. Moreover, we decide to hard-code the state of TCP and UDP ports in the `emulate_host_X` rules. While this decreases the flexibility of the honeytank, it eases its implementation. Similarly, the association between the opened ports and the service replying to the incoming data is embedded in the code of each rule.

More precisely, when the rule receives an UDP datagram, it can directly trigger a rule instance which simulates an UDP service and handles the data present in the UDP datagram. As for TCP segments, conceptually, the association between a TCP port and a service is embedded in the code of the host simulation rule. Indeed, when a new TCP session is initiated, this rule triggers a rule that tracks the new TCP session. This association could thus be passed as a parameter to the session simulation rule. When this rule receives a TCP segment containing data, it could trigger a rule simulating a TCP service according to the association passed as a parameter. In practice, this is not the case as the session simulation rule always executes the same TCP service for a given port. So, strictly speaking, the association between a TCP port and a service is hardcoded in the rules simulating a session.

Information maintained per instance. The IP address of the simulated host is specific to each instance of the rule `emulate_host_X`. It is passed as a parameter when the instance is created and remains constant. The activity timestamp is also maintained as a parameter of the rule. Each time an instance receives a packet, it updates this parameter with the timestamp

of the received packet. Similarly, the last Initial Sequence Number is stored as a rule parameter and is updated each time a new TCP session is created.

The number of active TCP sessions of the host is also maintained as a parameter. The `emulate_host_X` increases this parameter when it receives at TCP segment initiating a new session. However, the simulation of a host and the simulation of opened TCP sessions are decoupled. It is thus not straightforward for the host simulation rule to know when a session is over. We propose the following solution. We define a global frozen variable named `TCP_session_terminated`. When it is not null, it contains the IP address of the host for which a TCP session has been terminated during the previous ASAX cycle. The rule `emulate_host_X` checks the value of this variable, if it contains the IP address of the simulated host, it decreases its parameter containing the number of active sessions.

Maintaining the list of opened sessions. The host simulation rule has to know if the current packet is part of an existing TCP session. If it is the case, the rule leaves the processing of this packet to the rule instance simulating the session. Otherwise, it handles the packet by itself. However, the attentive reader will have observed that the list of opened sessions is not maintained by the rule simulating a host. Indeed this information is not strictly necessary. We define a global variable named `TCP_session_already_exists`. This variable is guaranteed to be set to 1 if the current packet is part of an existing TCP session and set to 0 otherwise. The rules simulating a session are responsible for updating this variable. We will discuss this in the subsection dealing with the session simulation rules.

Dynamic or static allocation of emulated hosts

The user deploying a stateful honeytank should be given the possibility to allocate emulated hosts statically or dynamically. The static allocation is the easiest to program. Indeed, the rule `emulate_host_X` must be instantiated once per simulated host at the beginning of the program (typically in the `init_action` rule). The instances live up to the end of the execution. However, this solution is only practical for a small number of hosts. Indeed, with the non-optimized version of ASAX, the ASAX cycle execution time (T_{ASAX}) depends on the number of rule instances. Thus, if we allocate statically a large number of hosts, the performance will be bad.

If the user chooses a dynamic allocation, the rules are instantiated only when needed, i.e. when the analyzer receives a packet destined to a host having not yet an instance simulating it. When an instance has been inactive for a given period of time, it is discarded. This solution is the most rewarding if we have to simulate a large number of hosts as it limits the number of rule instances and provides better performance than the static allocation solution. However, its programming is complex. We need a mechanism to

know if a rule instance simulating the destination host of the current packet already exists. Moreover, we also need timeout mechanism to discard useless rules.

The techniques used to launch the stateful honeytank, either with a static or dynamic allocation of host, are presented in Section 8.2.3 of this chapter.

Implementation

Figure 8.1 presents the pseudo-code of the `emulate_host_X`. First, it tests if the current packet is intended for the simulated host by inspecting its destination IP address. If it is the case, the rule checks if a TCP session associated with the emulated host has been terminated during the previous cycle and computes the number of active session accordingly. Then, it handles the current packet according to the protocol it contains. If it is an ICMP packet and if it contains an ICMP echo request, the rule sends the adequate ICMP echo reply immediately. If the IP packet contains an UDP message, then the destination port is tested. If it is closed, this packet must be handled as a `PU` Nmap test. Otherwise, a rule handling UDP protocols is triggered for the current event.

Listing 8.1: Simplified TCP/IP stack emulation rule for the stateful honeytank

```
1  rule emulate_host_X(
2       simulated_IP_address, last_ISN,
3       last_timestamp, nb_sessions : integer);
4
5  var new_ISN, new_nb_sessions : integer;
6  if
7     <The current IP destination  == simulated_IP_address> -->
8     begin
9        if
10          TCP_session_terminated = simulated_IP_address -->
11          new_nb_sessions := nb_sessions - 1;
12          true --> new_nb_sessions := nb_sessions
13       fi;
14       new_ISN := last_ISN;
15
16       if
17          <Packet is ICMP Echo Request> -->
18          <Forge and send ICMP Echo Reply>
19
20          <Packet is UDP> -->
21          if
22             <destination port is a closed port> -->
23             <Reply to the NMap PU test>
24             true -->
25             trigger off for_current <rule simulating UDP service>
26          fi;
```

```
27
28          <Packet is TCP> -->
29          if
30             <destination port is an opened port> -->
31             if
32                TCP_session_already_exists = 0 -->
33                if
34                   <SYN> -->
35                   begin
36                      new_ISN := <Choose a correct ISN>;
37                      <reply with a SYN+ACK>;
38                      new_nb_sessions := new_nb_sessions + 1;
39                      trigger off for_next <rule tracking TCP session>
40                   end;
41
42                   true --> <Reply to the NMap test>
43                fi
44             fi;
45             /* Destination port is closed */
46             true --> <Reply to the NMap test>
47          fi;
48       trigger off for_next emulate_host_X(simulated_IP_address, new_ISN,
49       packet_timestamp, new_nb_sessions)
50       end
51    fi;
52
53    true -->   /* IP destination is not the simulated IP */
54    if
55       nb_sessions = 0 AND
56       <elapsed time since last activity timestamp > timeout> --> skip;
57       true -->
58       if
59          TCP_session_terminated = simulated_IP_address -->
60          trigger off for_next emulate_host_X(simulated_IP_address,
61          new_ISN, packet_timestamp, new_nb_sessions - 1)
62          true --> trigger off for_next emulate_host_X(
63          simulated_IP_address, new_ISN, packet_timestamp,
64          new_nb_sessions)
65       fi
66    fi
67 fi;
```

The case of TCP segments is more complex. If the TCP segment is sent to a closed port, the rule tests if it is one of the special test segments sent by NMap and it replies to it according to the type of the simulated TCP/IP stack. If it is sent to an opened port, the rule cannot handle it directly. First of all, it verifies that the segment is not part of an existing TCP session. To achieve this, the rule checks the content of the `TCP_session_already_exists` global variable. This variable is guaranteed to contain 1 if the current packet is part of an existing TCP session and 0 otherwise. If the current packet is part of an existing TCP session, the rule has nothing to do as the packet will be processed by a rule simulating

the session. Otherwise, the rule `emulate_host_X` must process the packet by itself. It tests its TCP flags. If it is not a `SYN` segment, it checks if it is one of the special message sent by Nmap and replies accordingly. Otherwise, this segment is the beginning of a 3-way handshake. The rule chooses an initial sequence number according to the type of simulated TCP/IP stack, replies to it and updates the number of opened TCP session associated with this host. Then, it triggers a rule that will follow the remainder of this session. Finally, the rule retriggers itself with updated parameters.

If the current packet is not intended for the simulated host, the rule checks if there is still active TCP sessions on this host. If it is not the case and if the simulated host has not been contacted for a long time, the rule discards itself. This test is only performed when the honeytank is launched with a dynamic allocation of emulated hosts. In a static configuration, the rule instances simulating hosts live until the end of the execution. If the rule is not yet in timeout, it tests if a TCP session associated with the emulated host has been terminated during the previous cycle. If it is the case, the rule retriggers itself with an updated value for the parameter storing the number of active sessions of this host. Otherwise, the rule retriggers itself without modifying its parameters.

8.2.2 Handling TCP sessions

To handle TCP sessions, a stateful honeytank must maintain the following information for each session:

- **Identifier of the session.** For each session, the 4-uple identifier of this session must be known.

- **Status of the session.** The honeytank must know in which phase is each session: the 3-way handshake, the data exchange phase or the clearing phase.

- **Timestamp, retransmission timeout and data of the last `SYN+ACK`.** To correctly retransmit `SYN+ACK` segments during the 3-way handshake phase, the honeytank must know the timestamp of the last `SYN+ACK` sent as well as the last value of the retransmission timeout. The latter is the amount of time after which the `SYN+ACK` segment must be retransmitted if the final `ACK` of the 3-way handshake has not been received. Moreover, it stores the representation of the `SYN+ACK` packet to be able to retransmit it as it is.

- **Expected acknowledgment and sequence number.** In order to detect retransmitted or desequenced segments, the honeytank maintains the expected sequence number of the next packet it should receive. Similarly, in order to detect that one of the packet it sent has

been lost, the expected acknowledgment number is maintained. However, we implemented this feature only for the 3-way handshake phase. Consequently, only rules dealing with the 3-way handshake will maintain the expected acknowledgment number.

- **Activity timestamps.** By maintaining a timestamp of the last packet received, lost packets can be detected and retransmitted. However, this feature is not implemented outside the 3-way handshake. Instead, we use it to discard rule simulating a session which has not been contacted for a long time.

To maintain this information, we propose the following design. To handle TCP sessions, we define several rules corresponding to the different phases of a TCP session:

1. The rule `TWHS_wait_ack` corresponds to the state in the 3-way handshake where a `SYN` segment has been received and has been replied to with a `SYN+ACK` segment. This rule waits for the final `ACK` segment of the 3-way handshake. It is triggered by the host simulation rule `emulate_host_X` when a `SYN` segment is received for a non-existing session.

2. The rule `handle_session` corresponds to the state of the data transfer phase, once the 3-way handshake phase been completed. This rule waits for TCP segments until a `RST` or `FIN` segment is received. It is first triggered by the `TWHS_wait_ack` rule once the 3-way handshake has been completed. Then, it retriggers itself.

3. The rule `CLEAR_wait_ack` corresponds to the state in the session clearing phase where a `FIN` segment has been received and has been replied to with a `FIN+ACK` segment. This rule waits for the final `ACK` of the session. It is triggered by the `handle_session` rule once it has received a `FIN` segment.

Each of these rules represents a different phase of the TCP session and, for a given session, only one instance of those rules exists. Thus, the mere existence of an instance of those rules implicitly states the phase in which a session is. The other information to maintain, such as the session identifier or the expected sequence or acknowledgment number, are stored as rule parameters.

Retransmitting lost `SYN+ACK` segments

One of the objectives of the stateful honeytank is to retransmit `SYN+ACK` segments that have not been acknowledged in time in the 3-way handshake phase. To achieve this, the rule `TWHS_wait_ack` stores information *via* its

parameters: the timestamp of the last `SYN+ACK` segment sent for this session, the retransmission timeout value and the representation of the `SYN+ACK` segment in order to retransmit it.

Each time it is executed, the rule tests if the elapsed time since the last `SYN+ACK` segment sent for this session is greater than the retransmission timeout passed as parameter. If it is the case, it retransmits the segment and retriggers itself with an update timestamp and a double retransmission timeout. When the retransmission timeout reaches 30 seconds, the rule discards itself.

Detecting out of sequence segments

Another objective of the honeytank is to detect out of order segments. This is done by the rule `handle_session`. It maintains the value of the expected sequence number of the next segment of the emulated session through a rule parameter. When the rule receives a segment, it checks if its sequence number matches the expected sequence number passed as parameter. If it is the case, the current segment was received in sequence. If the sequence number is smaller than expected, it means that the current segment is a retransmission and can be discarded. Otherwise it means that a previous segment placed before the current one has not yet been received. We should place the current segment in a buffer and wait for the previous segments to arrive. However, as explained in section 8.1, we do not respect this part of the TCP protocol. When such a segment is received, it is ignored. This forces the attacker to resend those unacknowledged segments.

Notifying the host simulation rule

We have seen earlier that rules simulating a session must communicate with the rule simulating a host to notify that they will handle the current event. This is done by applying the Oracle technique presented in section 7.1.1. When the rules simulating a session are executed, they check if the next packet is part of the session they simulate. If it is the case, they set the variable `TCP_session_already_exists` to 1. This variable is tested during the next cycle by the `emulate_host_X` instances to determine if it has to process the current packet by itself.

The rules must also notify the `emulate_host_X` rule when a session is over. To achieve this, they set the variable `TCP_session_terminated` to the IP address of the host for which a session has been terminated. This variable is tested by the `emulate_host_X` instances to determine if one of their session has been terminated.

Implementation

The construction of the rules is rather classic. They first check if the current packet is a TCP segment part of the session they simulate. This is done by comparing the 4-uple identifier with the identifier passed as parameter. If it is the case, they verify that the segment is coherent with regard to the state of the state machine they represent. Figure 8.2 shows the pseudo code of the rule `handle_session`.

It first checks if the current packet is intended for the simulated session by inspecting its 4-uple identifier. If it is the case, the rule checks the type of TCP segment received. If the segment is a `RST`, the rule notifies the `emulate_host_X` rule by setting the `TCP_session_terminated` to the IP address of the destination host. If it is a `FIN` segment, the rule replies with a `FIN+ACK` segment and creates an instance of the `CLEAR_wait_ack` which will follow the remainder of the session. It also tests if the next segment is intended for the simulated session. If it is the case, it notifies the host emulation rule by setting the `TCP_session_already_exists` variable to 1.

If the received segment is an `ACK`, its sequence number is compared to the expected sequence number. If it is lower, it means that the received segment is a retransmission and can be ignored. If it is higher, the segment is out of order and we terminate the session as we do not support this case. Otherwise, if the segment contains a payload, it triggers a rule for handling it. Then the rule retriggers itself and possibly notifies the host emulation rule that it will handle the next packet.

Finally, if the rule must not handle the current packet, it tests if it has reached its timeout. If it is not the case, it retriggers itself and, if necessary, notifies the host emulation rule that it will handle the next packet.

Listing 8.2: Simplified code of the `handle_session` rule

```
1
2  rule handle_tcp_session(
3       ip_s, ip_d, tcp_s, tcp_d, /* 4-uple TCP identifier */
4       expected_seq,             /* Expected sequence number */
5                                 /* of the next TCP segment */
6       last_timestamp : integer);
7  if
8     <current IP source  == ip_s> AND <current IP dest.  == ip_d> AND
9     <current TCP source == tcp_s> AND <current TCP dest. == tcp_d> -->
10    if
11       <Packet is a TCP RST segment> --> TCP_session_ended := ip_d;
12
13       <Packet is a TCP FIN segment> -->
14       begin
15          <Send a FIN+ACK>
16          trigger off for_next CLEAR_wait_ack(ip_s, ip_d, tcp_s, tcp_d,
17          tcp_sequence + stringlen(tcp_payload) + 1);
18          if
19             <next packet has same 4-uple ident. as our session> -->
```

```
20              TCP_session_already_exists := 1
21           fi
22        end;
23
24        <Packet is a TCP ack segment> -->
25        if
26           /* Is the TCP sequence number correct ? */
27           tcp_sequence = expected_seq -->
28           begin
29              /* Reply to payload if present */
30              if
31                tcp_payload != '' -->
32                trigger off for_current TCP_service_simulation
33              fi;
34
35              <self retrigger with updated expected_seq parameter>;
36
37              if
38                 <next packet has same 4-uple ident. as our session> -->
39                 TCP_session_already_exists := 1
40              fi
41           end;
42           /* Ignore retransmitted or out-of-sequence segments */
43           true -->
44           begin
45              <self retrigger>;
46              if
47                 <next packet has the same 4-uple ident. as our session>
48                 --> TCP_session_already_exists := 1
49              fi
50           end
51        fi;
52
53     /* The current packet is not part of our session */
54     true -->
55     if
56        <elapsed time since last activity timestamp > timeout> --> skip;
57        true -->
58        begin
59              <self retrigger>;
60              if
61                 <next packet has same 4-uple ident. as our session> -->
62                 TCP_session_already_exists := 1
63              fi
64        end
65     fi
66  fi;
```

8.2.3 Launching the stateful honeytank

In Section 8.2.1, we explained that the allocation of emulated hosts could be either static or dynamic. This influences the way the stateful honeytank

must be launched. In a static configuration, the instances simulating hosts must be created inside the `init_action` rule. In a dynamic configuration, we need to create instances of the emulate_host_X rule when needed.

In order to achieve this goal, a mechanism informing us if a rule instance simulating the destination host of the current packet already exists is needed. Such mechanisms have been presented in Chapter 7. We use the technique of the Oracle presented in section 7.1.1. We define a global frozen variable `host_already_exists` and we modify the rule `emulate_host_X` the following way. In addition to verifying if the current packet is destined to the simulated host, the rule also checks if the next packet is intended for it. If it is the case, it sets the variable `IP_already_exists` to 1. We then define a rule named `allocate_host`. This rule is executed at each packet and tests the value of the variable `IP_already_exists`. If it is set to 0, it means that no rule instance will handle the current packet. It thus triggers an instance of the emulate_host_X rule to handle it.

8.3 Optimized implementation

To achieve its goals, a stateful honeytank maintains many information through rule parameters and uses a rule instance per emulated host as well as a rule instance per TCP session. With the non-optimized version of ASAX, we have seen that T_{ASAX}, the cycle processing time, is proportional to the number of rule instances. When simulating large amounts of hosts in an environment where many TCP sessions are opened, the number of rule instances can become quite big and seriously impact the performance. A dynamic allocation of emulated hosts and a timeout mechanism for discarding unused hosts and session can moderate the problem.

However, we also developed a stateful honeytank tailored for the optimized version of ASAX. Under some conditions, it allows the ASAX cycle time to remain constant, no matter what the number of rule instances is. The methodology for writing optimizable rules has already been presented in section 7.2. The stateful honeytank can truly benefit from the optimized version of ASAX. As all rule instances wait for a specific packet, for each received packet only a limited number of rule instances must be executed: the rule instances simulating the destination host of the current and next packets as well as the instances simulating the sessions of which the current and the next packet are part. All other rule instances can retrigger themselves. However, some rules must be written with care to avoid efficiency issues.

Optimizing the timeout mechanism First, the `TWHS_wait_ack` rule must check if its retransmission timeout has expired and retransmit a `SYN+ACK` if it is the case. As explained in Chapter 7, computing the elapsed time

since the last activity and comparing it with an inequality test with the retransmission timeout parameter is not optimizable. While developing the stateful honeytank, we thus apply the technique presented in Section 7.2.2. The format adapter is used to number all packets. When an instance of this rule is triggered, it receives as an additional parameter the estimated packet number when the retransmission timeout is expected to expire. Indeed, the format adapter produces fake events if no real packets are available. The stream of events analyzed is thus continuous. Moreover, when the honeytank is optimized, the cycle processing time is almost constant. The number of the record when a timeout will be reached can be simplified to adding to the current packet number the result of the division of the timeout value per the average T_{ASAX}.

Unbalanced parameter values Second, it is possible that many instances of rules handling a TCP session have one component of the TCP 4-uple identifier in common. This is likely to occur when an attacker scans the network simulated by the honeytank in search of vulnerable services. For example, he can systematically open sessions to the port 80 of emulated hosts. The 4-uple identifier is passed as parameter to the various rule simulating a session. We have seen in Chapter 7 that when many instances share the same parameter value an efficiency issue can occur. We thus apply the technique described in section 7.2.2. Instead of testing the four components of the 4-uple identifier one by one, we use a stub rule which tests a hash of it and triggers the real rule if the hashes matches. By hashing the components of the 4-uple identifier, we increase the distribution of values across the instances, thus limiting the risk of having many instances with the same parameter value.

8.3.1 Introduction to performance evaluation

With those techniques, the stateful honeytank can fully benefit from the optimized version of ASAX. It means that for each received packet, four rule instances will be executed at the most while the others are retriggered all at once. Thus, T_{ASAX} will be constant. This has an interesting consequence. As the execution time of a cycle is constant, the response time of a stateful honeytank can be computed in advance.

Let us denote the ASAX cycle processing time by T_{ASAX} and the number of packets in the packet queue of the format adapter by f. When a host sends a packet to the honeytank, it is placed at the tail of the queue. This packet is processed only when all other f packets in front of it in the queue have been processed. Since each of those f packets will be processed in a constant T_{ASAX} time, we can conclude that the response time of an optimized stateful honeytank for a given packet is $T_{ASAX} \times f$.

If the elapsed time between the arrival of two network packets is lower than T_{ASAX}, it means that packets arrive faster than they can be processed. Thus, they start to accumulate in the network queue of the format adapter and the value of f increases. Consequently, the response time of the honeytank also increases. Instead of letting packets accumulate and trying to reply to them at all costs, we decide to stop creating new hosts and new sessions when f hits a user-defined limit. This will help to diminish the size of the network packet queue. The performances of the stateful honeytank and the impact of this limitation are further discussed in Chapter 10.

8.4 Stateful simulation of services

8.4.1 Introduction

We explained so far how our framework can be used to build a stateful honeytank, i.e. a honeytank that maintains state in order to simulate more accurately the TCP/IP layer of the emulated hosts. Now, we apply the same techniques to build stateful application repliers.

In Chapter 3, we presented the Nepenthes honeypot. This honeypot is shipped with several stateful application repliers. We explain in this section how to translate those stateful service simulators in RUSSEL and integrate them into the stateful honeytank. Nevertheless, the techniques presented here can be generalized to build any type of stateful repliers.

8.4.2 Nepenthes

Nepenthes [1] is a honeypot framework that aims at collecting *self replicating malware* by simulating only the vulnerable parts of services. The architecture of Nepenthes consists of a core module and of several other modules implementing specific functionality. Amongst those modules are the vulnerability modules that are in charge of simulating the vulnerabilities of services. We invite the reader to refer to our detailed presentation of the Nepenthes architecture in Chapter 3 for additional details.

Architecture of the vulnerability modules

Nepenthes is shipped with several vulnerability modules that are all based on the same skeleton presented in Listing 8.3 We now explain this structure.

Each vulnerability module is made of two C++ classes called the `Vuln` and `Dialogue` classes. The `Vuln` class defines the configuration of the module. For example, the TCP ports on which the module wants to listen are defined in this class. This class is instantiated once by the Nepenthes core at start-up. After its instantiation, a method of the instance is called by the core to retrieve the configuration of the module.

Listing 8.3: Simplified skeleton of a Nepenthes vulnerability module

```
1  namespace nepenthes {
2    class Vuln : public Module {
3      public:
4          Vuln(...);
5          bool Init();
6            [...]
7    };
8
9    class Dialogue : public Dialogue {
10     public:
11         Dialogue(...);
12         ConsumeLevel incomingData(Message *msg);
13           [...]
14
15     protected:
16         Buffer *m_Buffer;
17         int m_State;
18           [...]
19     };
20 }
```

The `Dialogue` class implements the state machine of the vulnerable service simulation. Each time a TCP connection is established on a port bound to the module, a new instance of this class is created and associated to the connection.

When some data is available on the connection, the Nepenthes core reads it into a buffer. Then, it calls the `incomingData()` method of the `Dialogue` instance associated to the connection. The data buffer is passed as a parameter to the method. The method analyzes the buffer and replies according to the state machine simulating the vulnerable service.

The Dialogue class

The architecture of the `Dialogue` class is the following.

- An `incomingData()` method. This method is called by the Nepenthes core. The core reads data from a TCP socket into a buffer and passes it as the `msg` parameter of this method. The `msg` buffer usually contains the payload of only one TCP segment. Since a complete valid request for the simulated protocol might be spread over several TCP segments, the `incomingData()` method must sometimes concatenate the buffers received from several calls before having a complete request to process. Once the `incomingData()` method has obtained a complete request, it analyzes it and replies to it according to the simulated protocol.

- Two instance variables are defined: `m_buffer` and `m_state`. The

`m_state` variable stores the current state of the state machine of the simulated service. The `m_buffer` variable is used to concatenate the data received from the Nepenthes core in order to form a complete request.

- The constructor of this class instantiates those instance variables. It can also send a "welcome banner" if the simulated protocol requires it.
- For some protocols, the class also defines some constants.

The incomingData() method

Typically, the `incomingData()` method performs the following actions:

1. The contents of the `msg` buffer passed as a parameter to the method is appended to the `m_buffer` buffer.

   The `msg` buffer contains data read on a socket by the Nepenthes core. Usually, it contains the payload of only one segment. Thus, the buffer might not contain a complete request.

2. Then, according the value of the `m_state` variable, the method expects to find a given type of request in `m_buffer`.

3. If the data in `m_buffer` is too short to contain a request of the expected type, the method assumes that the whole request has not yet been read from the socket by the Nepenthes core. The `incomingData()` cannot process the contents of the `m_buffer` right now. Thus, the method simply returns, assuming that it will be called later with additional data.

4. If the data in `m_buffer` is long enough to be a request of the expected type, the `m_buffer` is parsed. If it is not a request of the expected type, the simulation is ended and the TCP connection is terminated. Otherwise, the request is handled and a reply is sent if needed. Then, the variable `m_state` is updated to hold the current state of the state machine of the simulated service. Finally, the bytes forming the processed request are removed from `m_buffer` and the method returns. If a final state has been reached, the connection is terminated.

However, this ideal behavior is only performed by a few modules. Most modules shipped with Nepenthes do not check extensively the contents of `m_buffer`. A lot of modules do not use `m_buffer` at all. They assume that the request to process is totally contained in the buffer passed as a parameter to the `incomingData()` method. Similarly, most modules do not parse the request and simply test if the length of the data is long enough to contain the expected request.

Listing 8.4: Simplified skeleton of a RUSSEL translation of a Nepenthes vulnerability module

```
1  global external frozen valid_tcp_segment : integer;
2
3  global constants ...
4
5
6  init_action;
7  begin
8    <initialization>
9  end
10
11 rule incoming_data_opt(hash,ip_s,ip_d,tcp_s,tcp_d :integer;buffer:string
       ;state:integer);
12 begin
13    if
14        rawToInt(tcp_session_hash) = hash and valid_tcp_segment = 1 -->
15      trigger off for_current incoming_data(hash, ip_s, ip_d, tcp_s,
            tcp_d, buffer, state);
16
17      true -->
18        trigger off for_next incoming_data_opt(hash, ip_s, ip_d, tcp_s,
              tcp_d, buffer, state)
19    fi
20 end;
21
22
23 rule incoming_data(hash,ip_s,ip_d,tcp_s,tcp_d :integer;buffer:string;
       state:integer);
24 var new_buffer : string;
25     new_state : integer;
26 begin
27   if
28     rawToInt(ip_source) = ip_s and rawToInt(ip_dest) = ip_d and
           rawToInt(tcp_source) = tcp_s and rawToInt(tcp_dest) = tcp_d -->
29         begin
30           new_buffer := '';
31           new_state := state;
32
33       /* Append payload to the input buffer */
34       new_buffer := concat(buffer, tcp_payload);
35
36           [ Process the contents of new_buffer ...]
37           /* If needed, send a reply and update the state */
38           new_state := ...;
39
40           trigger off for_next incoming_data_opt(hash, ip_s, ip_d, tcp_s
                 , tcp_d, new_buffer, new_state)
41     end;
42
43     true --> trigger off for_next incoming_data_opt(hash, ip_s, ip_d,
           tcp_s, tcp_d, buffer, state)
44   fi
45 end.
```

8.4.3 Translating a Nepenthes vulnerability module into RUSSEL

Translating a Nepenthes vulnerability module into RUSSEL is rather straightforward. A vulnerability module is translated into a RUSSEL module. The simplified skeleton of a Nepenthes vulnerability module is presented in Listing 8.3 while Listing 8.4 shows its translation into RUSSEL.

Constants and variables classes of the `Vuln` and `Dialogue` classes are translated into global RUSSEL variables. They are initialized by the `init_action` rule of the module.

The `incomingData()` method of the `Dialogue` class is translated into two RUSSEL rules: `incoming_data_opt` and `incoming_data`. The first rule is a "stub" of the latter. This stub rule is specially written to be optimized. It tests if the current packet must be handled by the `incoming_data` rule. If it is the case, this rule is instantiated for the current record by the stub rule. This is an application of the methodology for writing optimizable rules presented in Chapter 7. We invite the reader to refer to this chapter for additional explanations on the use of stub rules.

The `incoming_data` rule and its stub receive several parameters. The first one are the usual parameters specifying the 4-uple identifier of the associated TCP session. The two last parameters are the translation of the `m_state` and `m_buffer` instance variables of the `Dialogue` class of the translated Nepenthes module.

In Nepenthes, the core calls the `incomingData()` method and passes as a parameter a buffer containing data read from a TCP socket. The mechanism used by the RUSSEL translation is somewhat different. The translated rule reads by itself the contents of the payload of the current packet. However, before reading the payload, the rule must ensure that the current segment is valid, i.e. that it is not a retransmitted or out-of-sequence segment. To achieve this, the `incoming_data_opt` tests the value of the global frozen `valid_tcp_segment` variable. This variable is set to 1 when the current segment is valid and its payload can be processed. This is an invariant that must be maintained by the stateful honeytank. We will explain in the following paragraphs how to do it.

Then, the `incoming_data` rule can analyze the contents of the input buffer and can decide to send a reply, update the maintained state, ...

8.4.4 Adding the translated modules into the stateful honeytank

The stateful honeytank must undergo three modifications in order to integrate the translated modules.

First, the translated modules must be imported into the honeytank. This is trivially done by adding the name of the modules to the list of imported

modules declared with the `uses` instruction.

Second, the honeytank must create a new instance of the `incoming_data` rule when a TCP connection is established on a port associated to the simulated service. In the stateful honeytank, the rule `TWHS_wait_ack` waits for the final `ACK` segment of the TCP 3-way handshake. After receiving this segment, the TCP connection is considered as established. Thus, the `TWHS_wait_ack` must be modified in the following way. When a session is established, the rule tests if the TCP destination port number is one of the port associated with the translated vulnerability module. If it is the case, it triggers for the current record a new instance of the `incoming_data` rule.

Finally, we must modify the stateful honeytank to respect the invariant stating that the global frozen `valid_tcp_segment` variable contains the value 1 when the current segment is neither a retransmitted nor an out-of-sequence segment. To achieve this, we must modify the rule `handle_tcp_session` of the honeytank. This rule is in charge of tracking a TCP connection once it has been established. The format adapter exports some information about the next record. The `handle_tcp_session` is modified to test if the next record is part of the tracked TCP connection. If it is the case, it tests the sequence number of the upcoming segment. If the sequence number shows that the next segment is neither a retransmitted nor an out-of-sequence segment, the `handle_tcp_session` sets the global frozen `valid_tcp_segment` variable to 1. This is an application of the methodology for writing RUSSEL rules presented in Chapter 7.

8.5 Conclusion

We have presented in this chapter the design and the implementation of a stateful honeytank. We have presented the state it must maintain in order to offer a better simulation of TCP/IP and of application-level protocols. We have applied our programming methodology presented in Chapter 7 in order to build a honeytank that benefit from the optimized version of ASAX. Moreover, we explained how stateful Nepenthes vulnerability modules can be translated in RUSSEL and integrated into our framework.

The functionality and the efficiency of stateless and stateful honeytanks is studied and compared in Chapter 10. We show that, thanks to the optimized version of ASAX, maintaining state has little impact on the performance of a stateful honeytank compared to a stateless one. Moreover, Chapter 11 presents a qualitative analysis of the traffic collected by stateless and stateful honeytanks. It shows that stateful honeytanks running stateful service simulators can improve the quality of the collected traffic for some protocols.

Chapter 9

Generating and analyzing incoming traffic

> Time is like a drug. Too much of it kills you.
>
> Terry Pratchett
> *Small Gods*

In this chapter we present the design and the development of a traffic generator based on ASAX. The prototype honeytanks presented in the previous chapters have different levels of functionality and performance. We have seen that the stateful honeytank has a greater level of realism than the stateless honeytank with respect to the TCP session handling. Similarly, we already have the insight that a stateless honeytank will have a constant packet processing time no matter what the number of simulated hosts is whereas the performance of the stateful honeytank depends on the number of active hosts and sessions if it is executed by the non-optimized version of ASAX.

To highlight those different functionality levels and to have a better view on the performance of the honeytanks, we have developed a traffic generator to assess their functionality and performance. It generates traffic at various rates according to customizable test scenarios and computes the response time of each step of the scenario. We first present the requirements that the traffic generator must meet and explain why we choose ASAX to implement it. Then, we explain what modifications need to be done in ASAX in order to implement a traffic generator fulfilling those requirements. We also present the test scenarios which are executed by the generator. We present three different implementations, including a specially tailored one for the optimized version of ASAX. Finally, we present a theoretical model of the behavior of the traffic generator and the stateful honeytank when they are used one against each other.

9.1 Traffic generator requirements

Testing the performance of network elements can be complex as many aspects of the communication can be evaluated. We focus on test scenarios generating traffic that stress-test honeytanks and highlight their limitations. To achieve this, the traffic generator must fulfill the following requirements.

- The tests performed by the traffic generator must not be limited to simple stateless request/reply patterns. They have to be able to generate packets according to a test scenario that can be encoded as a finite state machine. For example, they must be able to simulate a host contacting a honeytank or a real host and performing a TCP 3-way handshake.
- For each step of the test scenario, the consistency of the reply sent by the tested honeytank is checked. Its response time is also measured.
- As honeytanks simulate the presence of multiple hosts, the generator must be able to initiate tests towards multiple simulated hosts.
- Similarly, honeytanks receive traffic originating from various attackers. The generator must reproduce this situation by forging traffic seemingly originating from various IP addresses.
- The generator has to be able to initiate tests in parallel toward several simulated hosts. If the tests were to be performed sequentially, they would not correctly evaluate the behavior of stateful honeytanks.
- The rate at which the tests are initiated must be customizable. Indeed, the honeytanks must be tested with various incoming traffic rate in order to find if their performances gracefully decays or, on the contrary, if it suddenly drops as we reach their limits.

To fulfill those requirements, we design and implement with ASAX a framework that permits to easily create various tests scenarios. The finite state machine test scenario can easily be translated in a set of RUSSEL rules and the binding with the `libnet` library allows us to forge and inject arbitrary packets onto a network.

9.2 Performing periodic actions with ASAX

The general design of a traffic generator using ASAX is the following. A RUSSEL rule is in charge of initiating tests at a given rate. The tests are initiated unconditionally and may overlap, i.e., if the period of time to wait before initiating a new test is elapsed, a new test is generated even if the previous is not completely terminated. Initiating a test consists in forging

and sending an adequate network packet toward the tested host. Then, a rule instance must be triggered. It waits for the honeytank answer and it follows the finite state machine describing the test scenario. Different source and destination IP are possibly chosen at each iteration so that the traffic seems to originate from different hosts and is directed toward different hosts. Different source and destination ports may also be chosen.

To initiate tests at a given rate, we must be able to trigger the execution of rules instance at a precise time. However, this is not possible as rule instances are executed only when an event is given to the analyzer by the format adapter. The execution cycle of ASAX is thus rhythmed by the arrival rate of events, where each event can be seen as a *clock tick*. In the context of the traffic generator, the received events are the network packets sent by the tested honeytank. As we have little or no control on those events, we cannot use them as *clock ticks* to initiate tests at a given rate. We have experimented several solutions to this problem.

Periodic generation of events with the format adapter. The first solution consists in modifying the format adapter to create fake events at a given rate. Then, the traffic generator uses a RUSSEL rule waiting for the occurrence of such a fake event. When a fake event is found, a new test is instantiated. We used the `setitimer()` system call to generate signals at a given rate. A signal handler was registered in the format adapter. When the signal handler receives a signal, it sets a flag indicating that a fake event must be generated the next time the analyzer asks for a new event. In fact, this approach has several problems.

The format adapter does not push events toward the analyzer. It is the analyzer that asks for a new event to the format adapter when it has finished processing an event. It means that several signals can be generated during an ASAX cycle if its duration is longer than the signal generation interval. It is important to take all received signals into account and to create a fake event for each signal received. Indeed, the RUSSEL program only instantiates a new test when the current event is a fake event. If some signals are ignored, the test will not be initiated regularly. To take into account the potentially multiple signals generated during an ASAX cycle, the format adapter maintains a queue of fake events. When the signal handler receives a signal, it creates a fake event with the current time as attribute and places it at the end of the queue. However, there is a risk of *starvation* for the real network events. If the format adapter gives the priority to the fake events, it will fetch a network packet and convert it into NADF only if the fake event queue is empty. If the interval between each signal is shorter than the duration of an ASAX cycle, the fake event queue will never be empty and the real network events will never be processed. But giving the priority to the network events is neither a good idea. Similarly,

if the incoming rate of network events is higher than the processing rate of ASAX, the fake events will never be translated and thus new tests will not be instantiated by the RUSSEL program.

To overcome this problem, we maintain two queues. The first one contains the fake events generated at each received signal. The second contains the network events received by the libpcap. When the analyzer asks for a new event, the format adapter compares the timestamp attribute of the first event of each queue and converts the older one into NADF. With this technique, the real and fake events are interleaved according to their timestamps and all events are guaranteed to be sooner or later processed. Unfortunately, this solution does not give satisfactory results. The main problem is that the `setitimer()` function does not generate signals regularly when a generation interval lower than one millisecond is used. Moreover, when a burst of network packets is received, their timestamps are close one to each other. Thus, several network events will be converted before a fake event is converted. If the processing time of those real events is high, the fake event is converted lately and this impacts the regularity of the tests initiation.

Periodic execution of RUSSEL rules. Then, we implemented a second solution. Instead of relying on the format adapter to generate fake events regularly, the RUSSEL program tracks itself the elapsed time and decides if it is time to initiate a new test. A rule, executed for each event, is in charge of computing the time elapsed since the last test initiated. If the elapsed time is greater or equal to the requested inter-test initiation time, then a new test is initiated. However, the precision of this technique depends on the regularity on which events are processed. If events are scarce or if they occur irregularly, the rule computing the elapsed time – and thus the initiation of tests – will not be executed regularly. Similarly, the same problem occurs if the processing time of an ASAX cycle is not stable. To overcome the irregular occurrence of events, we reuse the *oracle* format adapter designed for the honeytanks presented in section 7.1.1. When the analyzer asks for a new event, this format adapter generates a fake event if there are no real network events available at this time. This way, there is always an event available for the analyzer and the stream of events is as dense as possible.

We now have a single format adapter suitable for both the honeytanks and the traffic generator. Moreover, we can reuse the programming techniques presented in section 7.1.1 which rely on the foreseeing abilities of the format adapter to ease the programming of the traffic generator. With this technique, the generation rate of new tests has reached the maximal granularity of the ASAX system. A new test can be instantiated at each ASAX cycle. This is also the limit of this approach. It is impossible to instantiate new tests at interval which are smaller than the processing time of an ASAX cycle. To solve the problem of the variations of the ASAX cycle processing

time, we carefully wrote the RUSSEL rules of the traffic generator to limit the number of rule instances and thus limit the duration of an ASAX cycle. We explain our different implementations in the section 9.3 of this chapter.

9.3 Implementing a Network Traffic Generator with ASAX

```
global next_test_time : integer;
global inter_test_time : integer;

rule test_creator:
if
   <current time> < next_test_time --> trigger off for_next test_creator;
   true -->
   begin
      trigger off for_current start_test;
      next_test_time := next_test_time + inter_test_time;
      trigger off for_next test_creator
   end
fi;

rule start_test;
begin
   <choose the host to contact>
   <forge and send the first packet of the test>
   trigger off for_next wait_reply_1
end;
```

Figure 9.1: Simplified generation rule

A basic implementation of the traffic generator is depicted by figure 9.1. A global variable `next_test_time` which contains the timestamp when a new test should be initiated is defined. The variable `inter_test_time` contains the time to wait between each new test. A rule named `test_creator` is in charge of initiating new tests at the right time. It is executed for each event. If the current system time is lower than the value of the `next_test_time`, it simply re-triggers itself. Otherwise, it triggers for the current event the execution of a rule which will start the test scenario. Then, it computes the timestamp of the next test instantiation by adding the inter-test instantiation time to the `next_test_time` variable. The rule `start_test` chooses which emulated host will be tested, sends the first packet of the test and triggers an instance of the rule `wait_reply_1` which will wait for the reply of the tested host. This rule checks the validity of the reply and computes the elapsed time since the sending of the first packet. If the test scenario contains multiple steps, the rule instance sends the next packet and triggers another rule instance (`wait_reply_2`) which will wait for the reply.

Each `wait_reply_i` ($1 \leq i \leq N$) rule for the N steps of the test scenario is parametrized to identify the reply of the tested host (typically, it is the 4-uple identifier of a TCP session).

With this method, the tests will be initiated on time and regularly as long as the processing time of an ASAX cycle is lower than the inter-test time. If the cycle time becomes greater than the inter-test time, the tests will be initiated tardily. Moreover, if the cycle time fluctuates, the tests will not be generated regularly. This situation is likely to occur: for each test, the generator sends a packet and creates a rule instance which will wait for the reply. If the generation rate is high and the tested honeytank replies slowly, rules instances will accumulate. Moreover, if the honeytank is overloaded and drops some packets some rule instances in the generator will never receive the answer they expect but will nevertheless be executed at each cycle. This will lead to an increase of the cycle processing time. We propose three solutions to this problem: discarding useless instance by using timeouts, imposing a limit on the number of rule instances and using the optimized version of ASAX.

9.3.1 Improve the performance by using timeouts

The first solution we experimented uses timeouts to discard rules which have not received the packet they were waiting for after a certain amount of time. Choosing a good timeout value is not easy. If it is set too high, we do not reach optimal performances. If it is set too low, we miss replies sent by the honeytank. We compute the timeout value dynamically the following way. For each step of the test scenario, a global variable is defined. It holds the timeout value for the expected reply of the honeytank for this step (e.g., the SYN+ACK reply for a previously sent SYN). When the program starts, those variable are initialized to a value chosen by the programmer. Each time a rule instance `wait_reply_i` is executed, it tests if the current event is the expected one. If it is not the case, it computes the elapsed time since its instantiation. If it is lower than its associated timeout, the instance retriggers itself, otherwise it is discarded. If the rule instance receives its awaited packet, it computes a new timeout value and updates the adequate global variable. This way, all other rule instances waiting for the same type of reply will use the updated timeout. The new timeout value is computed as

$$(rtt + NPQ \times T_{ASAX}) \times f$$

where rtt is the round-trip time of the received packet, NPQ is the number of buffered (awaiting) network packets in the network queue of the format adapter, T_{ASAX} is the time used by the ASAX analyzer to process all the rule instances for the last received event, and f is a "security factor" (e.g., 1.5) experimentally determined. This way, the computed timeout value is dynamically adapted to the reply speed of the tested honeypot (rtt) as well

as the amount of time a packet waits in the network event queue before being processed ($NPQ \times T_{ASAX}$).

With this timeout mechanism, the useless rule instances are killed and the global load of the ASAX analyzer is reduced. However, under heavy load, the time used by the ASAX analyzer to process rule instances might still be greater than the requested generation rate. When this situation occurs, it means that the packets are generated faster than the generator is able to handle their replies. If it goes on sending new packets, new rule instances will further decrease the performance of the ASAX analyzer and replies will accumulate. To prevent this situation, we stop the generation of new packets whenever T_{ASAX} is greater or equal to the desired generation rate. This mechanism avoids the generator being overwhelmed by the received traffic it induces by instantiating new tests.

9.3.2 Limiting the number of rule instances

The previous solution using timeouts showed some practical limitations. The choice of the initial value for the timeout as well as the security factor proved to be crucial for the good working of the generator. Unfortunately, the right values for those parameters must be chosen experimentally by trials and errors. We thus developed another solution which does not require fine tuning.

The accuracy of the generator depends on the ASAX cycle processing time, T_{ASAX}. It must remain stable and lower than the inter-test instantiation time. On the non-optimized version of ASAX, T_{ASAX} directly depends on the number of rule instances to execute at each cycle. Thus, to ensure that T_{ASAX} remains stable and does not grow over a given threshold, we propose to put a limit to the number of rule instances. When this limit is reached, the generator stops initiating new tests and tries to discard useless rule instances. Once the number of rule instances drops below the threshold, tests are anew initiated.

Discarding useless rule instances relies on the following principles. First, the events given to the analyzer are numbered sequentially. As only one event is processed during an ASAX cycle, the event number also identifies the ASAX cycle. It can been seen as a timestamp. When a rule `wait_reply_i` is triggered, it receives as parameter the timestamp cycle number when it has been instantiated. Thus, rule instances which have lower values for these parameters are older than the other instances. Second, we assume that the computer running the generator and the computer running the tested honeytank are directly connected to the same physical network. In this configuration we can assume that packets are received in the same order as they were sent. Moreover, we know that the honeytank replies to incoming traffic with a FIFO policy. We can thus conclude that when the honeytank replies to a given packet, all older packets have either been already processed

or have been lost. Thus, when a rule instance `wait_reply_i` in the generator receives the packet it has been waiting for, it means that older rule instances will never receive their awaited packet and can be thus discarded, i.e. all instances with a lower timestamp parameter can be killed. The advantage of this method is that the user has only one parameter to tune: the maximum number of rule instances.

9.3.3 Traffic generator using the optimized version of ASAX

Historically, the optimized version of ASAX was not yet available when we developed the traffic generator. Thus, we had to use techniques limiting the number of rule instances in order to keep the ASAX cycle processing time to a stable and low value. However, when the optimized version of ASAX became usable, we developed another version of the traffic generator in order to benefit from the optimizations. As long as some constraints are respected, the optimizations allows the ASAX cycle time to remain constant no matter what the number of rule instances is. Those constraints have already been explained in section 7.2.1 and they have been applied to the stateful honeytanks in Chapter 8.

The traffic generator can also benefit from the optimizations. Indeed, many instances of the rules `wait_reply_i` exist during the execution of the generator. Moreover, the test scenarios we developed guarantee that all test instantiations are different, i.e. the IP addresses or the TCP ports change at each iteration. Thus, we are guaranteed that only one rule instance `wait_reply_i` will be interested in the current event while all other instances will retrigger themselves. This situation is ideal as the optimizations can detect the only rule instance which needs to be executed and retrigger at once all other rule instances.

The generator is a good example where inefficiencies can appear if it is not programmed correctly. In particular, we have seen that we had to avoid the conjunction of two situations: first, an optimized condition performs an equality test on a parameter. Second, many rule instances have as effective parameter the value which satisfies to this test. We have presented in section 7.2.2 a technique using a hash function and stub rules to avoid this situation. In a few words, it allows the values of the parameters to be randomly distributed amongst all rule instances and avoid the case when many rule instances share the same parameter value. This technique was used to write a fully optimized version of the stateful honeytank. We could also use it to program the generator but it is possible to avoid it and use a more elegant approach.

In the case of the honeytank, no assumption can be made on the received traffic as it is connected to the Internet. However, the case of the generator is different. The generator initiates traffic toward a honeytank and analyzes the replies. It means that it will only receive traffic from emulated hosts it

has first contacted. Some hosts will not reply and some hosts will reply with incorrect data but an emulated host which has not been contacted by the generator will not send data by itself. This applies also to TCP sessions. For example, when the traffic generator initiates TCP sessions using always the same TCP destination port number, e.g. 80, it will never receive replies with a TCP source port different than 80. We can use this knowledge to write fully optimizable rules for the traffic generator. Instead of triggering `wait_reply_i` with the 4-uple identifier of the TCP session, we can omit the constant part of this identifier (in this example, the TCP destination port) as we know it will always have the same value. We thus avoid having many instances with the same parameter value and do not need to rely on a hash function.

Chapter 10

Assessing traffic handlers

> The mystery of life isn't a problem to solve, but a reality to experience.
>
> Frank Herbert
> *Dune*

This chapter presents the assessment of our prototype honeytanks. In the previous chapters, we have exposed various techniques for building traffic handlers and we have applied them to create prototype honeytanks as well as traffic generators. The honeytank implementations offer different levels of realism by maintaining more or less state. In this chapter, we present a methodology to assess the functionality of a honeytank. We also propose a mathematical model to predict the performance of honeytanks and traffic handlers, for a class of practical situations. We check experimentally that the model works in practice. We compare our honeytanks with Honeyd both for conformance to expected functionality and for efficiency, by means of our traffic generator. We also perform efficiency tests with TCPReplay to have tests that cannot be biased by using our tools only.

This chapter is organized as follows: In Section 10.1, we explain how to set up a network testbed to conduct the assessments. In Section 10.2, we present functional tests that evaluate the capacity of a honeytank to accept incoming traffic and to handle it in a realistic way. Those tests offer a qualitative view of a honeytank in terms of realism of the simulation. In Section 10.3, we compare the standard and optimized versions of ASAX. We show that, in the context of a stateful honeytank, the time needed to execute an ASAX cycle is quite stable, independently of the number of active rule instances, when we use the optimized version of ASAX. We thus provide, in Section 10.4, a finer analysis of the ASAX cycle time to support the assumption that the time needed to handle an incoming packet is almost constant (for the same kind of packet). Based on this assumption,

we present, in Section 10.5, a theoretical model to predict the performance of a traffic handler. We estimate the behavior of a single traffic handler. Then, we study how two traffic handlers (e.g., a traffic generator and a honeytank) interact with each other. Finally, we validate our model with a simple simulator of a honeytank and a generator exchanging data with each other. In Section 10.6, we observe the evolution of honeytanks performance when subjected to different rates of incoming traffic. This offers a quantitative view of the performance of a honeytank. We show the conformance of those practical tests to the theoretical model. In Section 10.7 and 10.8, we perform specific efficiency tests to highlight the differences between our honeytanks and Honeyd. The tests use our traffic generator as well as TCPReplay [65]. Our honeytanks outperform Honeyd in most situations.

10.1 Network testbed requirements

To perform the tests, we set up an environment simulating two class `A`[1] networks. Figure 10.1 depicts the network configuration. We use a private network of two computers, named `H` and `G`, linked together through a hub, a switch or a cross-cable. Computer `H` is used to run the tested honeytanks. It will handle packets sent to the 11.0.0.0/8 network. It forges packets seemingly originating from hosts inside the 12.0.0.0/8 network. It uses computer `H` as a gateway to the 11.0.0.0/8 network. The actual IP addresses of the network interfaces of G and H might be chosen freely but must not be inside the 11.0.0.0/8 and 12.0.0.0/8 networks. If it were the case, packets sent to those addresses would be handled by both the operating system of `G` or `H` and by the ASAX system running on those computers.

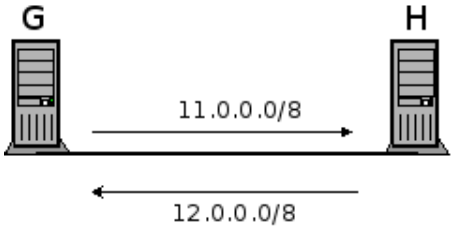

Figure 10.1: Network configuration for assessing honeytanks

The actual configuration used to perform the tests presented in this chapter is the following. Computer `G` has an AMD Athlon CPU clocked at 1400 MHz, 640 megabytes of RAM and uses a 100 Mb/s ADMtek NC100 Ethernet adapter in full-duplex mode. Its IP address is 192.168.0.1 . Computer `H` has a Intel Pentium III CPU clocked at 665 MHz, 256 megabytes of RAM and uses a 100 Mb/s ADMtek NC100 Ethernet adapter in full-duplex mode. Its

[1] 2^{24} IP addresses

IP address is 192.168.0.2 . Those computers are connected together trough a cross-cable. They are both running Ubuntu Linux 7.04 .

10.2 Functional testing

To maximize the amount of malicious traffic collected, a honeytank should behave like a real host. In particular, it must implement enough of the TCP protocol to handle correctly the opening of a TCP session and exchange some data with the initiator of the session. Moreover, a honeytank should not be too easily detectable, i.e., it should react correctly to the most common situations. In this section, we propose several test scenarios that highlight the ability of a honeytank to simulate correctly a host at the TCP level. However, those tests are not meant to be a full TCP conformance test suite and reproduce only the most common situations. Moreover, they do not evaluate the application-level simulation of the honeytank.

We defined eight test scenarios and implemented them with the traffic generator presented in Chapter 9. Those scenarios assess the handling of TCP sessions openings and closings as well as the handling of segments sent to unopened sessions. Moreover, a ninth test is performed using NMap and evaluates the behavior of the tested system against an NMap OS fingerprinting scan. We use those tests to evaluate the stateless honeytank presented in Chapter 5, the extended honeytank presented in Chapter 6 and the stateful honeytank presented in Chapter 8. We also assess the performance of the low-interaction honeypot Honeyd[47].

10.2.1 Configuration of the traffic generator

The test scenarios are independent from each other and are executed one at a time. For each test, the traffic generator initiates a transaction each 1000 μseconds during 10 seconds. Those transactions are generated unconditionally, i.e., the generator does not wait to receive the answer to a previous transaction before initiating a new one. The source IP of the forged segments is always 11.0.0.1. The source TCP port starts at 1 and is incremented at every iteration. The destination of those segments is always the 12.0.0.1 IP.

The generator initiates traffic toward only one simulated host as those tests assess the functionality of the honeytanks and not their ability to simulate numerous hosts. Moreover, to collect all the answers of the tested honeytanks, the traffic generator is configured to have an unlimited number of active rules and there is no limit on f_G, the length of its incoming packet queue. The generator stops its execution after one minute of network inactivity. We use the optimized version of ASAX to run those tests and the implementation of the test scenarios uses the techniques presented in Section 9.3.3 to benefit from the optimizations.

10.2.2 Configuration of the tested systems

The tested honeytanks are configured the following way. The stateless honeytank replies to all traffic it receives. As the computer on which it is running is configured to receive all traffic destined to the 12.0.0.0/8 network, the stateless honeytank thus emulates all the hosts of this network. The extended and the stateful honeytanks simulate only the host 12.0.0.1. The TCP ports 25, 80 and 111 are opened and respectively bound to a SMTP, HTTP and RPCInfo simulator. All other TCP ports are closed. The host simulated by the extended and stateful honeytanks is created at the initialization of the honeytanks and has an unlimited lifetime. We use a static configuration with only one simulated host to evaluate the intrinsic ability of a honeytank to pass these functional tests without interferences with the creation and destruction of dynamic hosts. Finally, Honeyd is configured to simulate the 12.0.0.1 host for which the TCP port 80 is opened and all other ports are closed. The shell-based HTTP simulator given in the `Honeyd` archive is bound to the port 80.

10.2.3 Assessing the handling of TCP sessions openings and closings

To collect malicious traffic, honeytanks must successfully accept incoming TCP sessions. The following tests assess various aspects of opening and closing TCP sessions.

Test 1: Attempt to open a TCP session on a closed port

This test consists in trying to open TCP sessions by sending `SYN` segments to the TCP port 55 of the simulated host. This port is closed on all tested honeytanks. A correct implementation should reply with a `RST` segment for each connection attempt. Table 10.1 shows the result of this test. During 10 seconds, the generator sent 10,000 `SYN` segments. All tested systems replied correctly as the generator received a `RST` segment for each `SYN` segment issued.

	Generator	
	SYN sent	RST rcvd
Stateless	10000	10000
Extended	10000	10000
Stateful	10000	10000
Honeyd	10000	10000

Table 10.1: (Test 1) Opening attempt of a TCP session on a closed port

Test 2: opening a TCP session on an opened port

In this test, the generator opens TCP sessions by sending `SYN` segments to the TCP port 80 of the simulated host. This port is opened on all tested honeytanks. If the host replies correctly, the generator completes the TCP the 3-way handshake by sending an `ACK` segment. Once a session is opened, the generator does not send any other packet belonging to this session. A correct implementation should reply with a `SYN+ACK` segment with the correct sequence and acknowledgment numbers for each connexion opening. Table 10.2 shows the result of this test.

The stateless and stateful honeytanks behave as expected. Each `SYN` sent is replied with a valid `SYN+ACK` segment with the correct acknowledgment and sequence numbers. The `ACK` segments completing the 3-way handshakes have been accepted by the tested host. However, the extended honeytank that we presented in Chapter 6 does not behave as expected. It replied correctly to `SYN` segments, but it replied with a `RST` segment for some `ACK` segments finalizing the 3-way handshake.

This problem occurs when the incoming traffic rate is high and causes two TCP sessions openings on the same host to interleave. When the extended honeytank receives the `SYN` segment of the first TCP session, it chooses an initial sequence number, remembers it, and replies with a `SYN+ACK`. The honeytank must yet receive an `ACK` segment to complete the TCP session opening. If at this moment the honeytank receives a `SYN` segment from a second TCP session opening, it will choose another initial sequence number, remember it – thus overwriting the previous value saved – and send a `SYN+ACK` reply. When the honeytank receives the `ACK` segment finalizing the 3-way handshake of the first TCP session, it detects that the acknowledgment number of the segment does not match the last initial sequence number and thus it rejects the TCP session opening by issuing a `RST` segment.

Honeyd also behaves strangely. It received 10.000 `SYN` segments but replied only to 2,041 of them. However, even after the generator sent `ACK` segments finalizing 3-way handshakes, Honeyd continued to send `SYN+ACK` segments, sending 5,337 of them in total. More surprisingly, we found that Honeyd was sending FIN segments to the generator. After manually inspecting the traffic exchanged between the generator and Honeyd, we found the explanation to those behaviors. First, even if the generator replies quickly with an `ACK` to the `SYN+ACK` segments it receives, Honeyd seems either to lose some of these segments or to process them lately. As a result, Honeyd considers that the generator has never received its `SYN+ACK` segments and starts to retransmit them. Second, once a TCP session has been established on the port 80 of the emulated host, Honeyd sometimes sends an `ACK` segment containing an HTTP reply with a 404 error message. As the generator never issued a HTTP GET request, Honeyd is not supposed to send such messages. Since the generator does not expect such messages, it does not

acknowledge them. Honeyd retransmits them several times and finally sends a `FIN` segment to close the session. We wrote a simple Python program that opens a TCP connection using standard system calls to a host emulated by Honeyd. Once the connection is established, the Python program closes it and thus the underlying operating system issues a `FIN` segment. We have observed that, after the `FIN` segment has been issued, Honeyd still sends segments containing an HTTP reply after the connection closing even if no HTTP request were sent. This behavior is thus independent from our traffic generator. Moreover, we observed this behavior of Honeyd with the shell-based HTTP replier as well as with the shell-based echo replier. To the contrary, internal services of Honeyd written in Python behave as expected. We have contacted the author about this issue. To perform a fair comparison with Honeyd, we thus use its Python internal services for the efficiency assessments presented at the end of this chapter.

	Generator			
	SYN sent	SYN+ACK rcvd	RST rcvd	FIN rcvd
Stateless	10000	10000	0	0
Extended	10000	10000	84	0
Stateful	10000	10000	0	0
Honeyd	10000	2041 / 5337	0	7248

Table 10.2: (Test 2) Opening of a TCP session on an opened port

Test 3: opening and closing a TCP session on an opened port

For this test, the generator opens TCP sessions on the TCP port 80 of the simulated host. Once the 3-way handshake completes correctly, it closes the session by sending a `FIN+ACK` segment. Then, it waits for the `FIN` reply and sends the closing `ACK`. Table 10.3 summarizes the results of this test.

The stateless honeytank behaves as expected, handling correctly the 3-way handshake and terminating correctly the session by replying to the `FIN` sent by the generator and accepting the final `ACK` segment. The stateful honeytank also behaves as expected.

Again, the extended honeytank does not behave correctly. In this test, the generator sends a `FIN+ACK` segment directly after sending the final `ACK` of the 3-way handshake and expects a `FIN+ACK` reply from the honeytank. However, Table 10.3 shows this is not the case. As explained for the previous test, when several sessions are opened simultaneously on the same host, the extended honeytank might reject the final `ACK` of the 3-way handshake and reply with a `RST`. Here, over the 10,000 sessions initiated by the generator, five 3-way handshakes did not complete correctly as the extended honeytank sent a `RST` segment upon receiving the `ACK` finalizing the handshakes. Thus, only 9,995 sessions were correctly managed, going from the 3-way handshake

to the exchange of `FIN` segments for the clearing phase. However, we can see that the extended honeytank did send 10,000 `FIN` segments during the test. This is explained by the fact that this honeytank does not maintain enough state to handle correctly all aspects of a TCP session and replies unconditionally to `FIN` segments it receives. Thus, when the extended honeytank receives the `ACK` segment finalizing the 3-way handshake followed by a `FIN` segment starting the clearing phase, it replies to the `FIN` segment even if it has sent a `RST` segment in reply to the previous `ACK` received.

Finally, the behavior of Honeyd described for the previous test still holds for this test. It often considers that the `ACK` sent by the generator to finalize the 3-way handshake is in timeout, thus Honeyd retransmits several `SYN+ACK` segments. Here it correctly replied to 1,753 `SYN` segments but sent 4,269 `SYN+ACK` segments in total. Moreover, Honeyd sends sometimes segments containing HTTP replies even if no HTTP request has been sent by the generator. It retransmits them several times, and finally sends a `FIN` segment to close the sessions. In this test, Honeyd replied correctly to 1,115 `FIN` segments sent by the generator but in total sent 4,708 `FIN`. Since the shell-based HTTP server seems bugged, those results must be interpreted with care. A performance test using Python internal services of Honeyd is presented at the end of this chapter.

	Generator				
	SYN sent	SYN+ACK rcvd	FIN+ACK sent	FIN rcvd	RST rcvd
Stateless	10000	10000	10000	10000	0
Extended	10000	10000	10000	9995 / 10000	5
Stateful	10000	10000	10000	10000	0
Honeyd	10000	1753 / 4269	1753	1115 / 4708	0

Table 10.3: (Test 3) Opening and closing of a TCP session on a opened port

Test 4: retransmitting SYN+ACK segments in the TCP 3-way handshake

This test assesses the capacity of a honeytank to retransmit `SYN+ACK` segments during the 3-way handshake. The generator sends a `SYN` segment to the port 80 of the simulated host to initiate a TCP session. When the generator receives the `SYN+ACK` reply from the honeytank, it does not complete the 3-way handshake and never sends the final `ACK`. The honeytank should consider that its `SYN+ACK` has been lost and should retransmit it several times. It is important to note that the generator does not process the received packets, i.e., rule instances waiting for the `SYN+ACK` replies are not created. Instead, a single rule instance is in charge for counting the total amount of `SYN+ACK` segments received.

Table 10.4 summarizes the results of this test. As expected, since the stateless and extended honeytank do not implement a retransmission fea-

ture, they just reply to the received `SYN`. The stateful honeytank and Honeyd however correctly retransmit unacknowledged `SYN+ACK` segments. The number of `SYN+ACK` segments sent by the stateful honeytank is about twice the number of segments sent by Honeyd. In the stateful honeytank, each time a segment is retransmitted the amount of time to wait before retransmitting the segment anew is doubled. When the timeout reaches 30 seconds, the honeytank gives up the retransmission of this segment. We believe that Honeyd stops retransmitting segments when the retransmission timeout reaches about 15 seconds.

	Generator	
	SYN sent	SYN+ACK rcvd
Stateless	10000	10000
Extended	10000	10000
Stateful	10000	65892
Honeyd	10000	40000

Table 10.4: (Test 4) SYN+ACK retransmission during 3-way handshake

10.2.4 Assessing the handling of segments sent to unopened sessions

The traffic received by honeytanks does not always consists of nice and clean TCP sessions. Part of the traffic consists of TCP segments sent to non-existent TCP session. The next tests evaluate the ability of honeytanks to reply correctly to this kind of traffic.

Test 5: handling SYN+ACK segments sent to an unopened session

In this test, the generator sends `SYN+ACK` segments to the TCP port 80 of the simulated host. This situation often occurs when an IP address used by the honeytank has been spoofed by an attacker and used to initiate a session with a host on the Internet. The host replies with a `SYN+ACK` segment sent to the spoofed IP address. A correct implementation should reply with a `RST` segment when receiving such a `SYN+ACK` segment. We can see in Table 10.5 that all tested systems behave correctly.

Test 6: handling RST segments sent to an unopened session

In this test, the generator sends `RST` segments to the TCP port 80 of the simulated host. A correct implementation should ignore those segments. The results of this test presented in Table 10.6 show that all the tested system correctly pass this test.

	Generator	
	SYN+ACK sent	RST rcvd
Stateless	10000	10000
Extended	10000	10000
Stateful	10000	10000
Honeyd	10000	10000

Table 10.5: (Test 5) SYN+ACK received outside a valid TCP session

	Generator	
	RST sent	Replies rcvd
Stateless	10000	0
Extended	10000	0
Stateful	10000	0
Honeyd	10000	0

Table 10.6: (Test 6) RST received outside a valid TCP session

Test 7: handling FIN segments sent to an unopened session

In this test, the generator tries to close an unopened TCP session by sending a `FIN` segment to the TCP port 80 of the simulated host. A correct implementation should reply with a `RST` segment for each attempt. Table 10.7 summarizes the results. The stateful honeytank reacts correctly by replying with a `RST` segment. Honeyd prints a "Killing unknown connection" message when it receives those segments and ignores them. The stateless and extended honeytanks however do not maintain enough state to pass this test. Indeed, they are not able to tell whether a given segment belongs to an existing session or not. They reply unconditionally to the `FIN` segments received.

	Generator		
	FIN sent	FIN+ACK rcvd	RST rcvd
Stateless	10000	10000	0
Extended	10000	10000	0
Stateful	10000	0	10000
Honeyd	10000	0	0

Table 10.7: (Test 7) FIN received outside a valid TCP session

Test 8: handling payload sent to an unopened session

In this test, the generator sends an `ACK` segment with a payload containing an HTTP GET request to the TCP port 80 of the simulated host. A cor-

rect implementation should reply with a `RST` segment for each connection attempt. Table 10.8 shows the results of this test.

As expected, the stateful honeytank and Honeyd correctly pass this test. However, the stateless and extended honeytank are not able to determine if a given segment is part of an existing session or not. We explained in Chapter 5 and 6 how those honeytanks can use the acknowledgment number of the received segments to guess if it is part of an established session. To highlight the limitation of this approach, we carefully choose the acknowledgment number of the segments sent by the generator to ensure that they are accepted by the stateless and extended honeytanks. We can see that the stateless and extended honeytanks accept the `ACK` segment, process the HTTP payload they contain and reply with a HTTP reply containing 207 bytes (actually a 200 OK message).

	Generator		
	ACK with payload sent	RST received	Total payload received
Stateless	10000	0	2070000 bytes
Extended	10000	0	2070000 bytes
Stateful	10000	10000	0 bytes
Honeyd	10000	10000	0 bytes

Table 10.8: (Test 8) ACK with HTTP payload received outside a valid TCP session

10.2.5 Behavior against an NMap OS fingerprinting scan

Finally, we test the ability of the honeytanks to reply correctly to an NMap OS fingerprinting scan. For this test, we launch an NMap OS fingerprinting scan against the emulated host by the honeytanks. It is ran from the `G` computer and is configured to probe the TCP port 55 and 80 of targeted host. The speed of the scan is left to the default configuration of NMap. We use two configurations. First, the extended and stateful honeytank as well as Honeyd are configured to simulate the TCP/IP stack named "Microsoft Windows XP Professional SP1". Then, they are configured to simulate the TCP/IP stack named "Microsoft Windows 98 or 98SE". When NMap scans the host simulated by the stateless honeytank, it reports an unknown type of stack. This is normal as the stateless honeytank has no support for simulating various types of stacks. When NMap scans the host simulated by the extended and stateful honeytank and the host simulates by Honeyd, it finds the expected type of simulated stack in both configurations of the honeypots.

10.3 Comparison between the standard and optimized versions of ASAX

From now on, and to the end of this chapter, we concentrate on efficiency issues. In Section 10.5, we propose a mathematical model to predict the performance of a traffic handler. Our main hypothesis is that, in a fixed given context (i.e., a specific traffic handler), the execution time T_{ASAX} of an ASAX cycle, for an incoming packet, is constant (i.e., it only depends on the content of the packet but not on the total number of active rules). We first provide some evidence that this hypothesis is reasonable if we use the optimized version of ASAX [27]. In this section, we perform efficiency tests with both the optimized and the original version of ASAX to show that the duration of an ASAX cycle is essentially proportional to the number of active rules for the original version but, in contrast, essentially bounded by a constant for the optimized version. In the next section, we further analyze the results of these tests to argue that the main relevant parameter to estimate the response time of a traffic handler is its T_{ASAX} measured for ASAX cycles corresponding to incoming packets (i.e., the so-called "real" records).

10.3.1 Testing procedure

We assess two versions of the stateful honeytank. The first one is specially crafted for the optimized version of ASAX and uses the optimization techniques presented in Section 7.2. We note this version as HT_{opt}. The second version, noted HT_{no-opt}, uses standard programming techniques. Both versions are used in their unlimited configuration, i.e., there is no limit on the size of the network packet queue. Moreover, they are configured to close opened TCP sessions after 10 seconds of inactivity.

Both versions of the stateful honeytank are executed by the optimized version of ASAX, noted $ASAX_{opt}$ as well as by the non-optimized version noted $ASAX_{no-opt}$. Thus, there are four configurations tested in total.

To assess those four configurations, we use the traffic generator executed by $ASAX_{opt}$ to open a given number of TCP sessions at a given rhythm towards hosts simulated by the honeytanks. Each time a new session is initiated, the generator increments the IP destination, the IP source and the TCP port source. The TCP port destination is always 80. We consider four types of tests. A *long* test consists of opening 10,000 TCP sessions whereas a *short* test consists of opening 1,000 sessions. A *fast* test opens a TCP sessions every 1,000 microseconds while a *slow* test uses an inter-session time of 10,000 microseconds.

For each test, we measure the evolution of T_{ASAX} over time as well as the evolution of the response time of the honeytank, i.e., the elapsed time between the emission of a `SYN` segment by the traffic generator and the

reception of the SYN+ACK reply of the honeytank.

In total, the four configurations of the honeytank tested with the four combinations of the short/long and slow/fast tests represent sixteen tests. However, we only present here a subset of the most relevant results.

10.3.2 Non-optimized stateful honeytank executed by the non-optimized ASAX

For this test, we assess the behavior of HT_{no-opt} executed by $ASAX_{no-opt}$ with a *short* and *slow* test. Figure 10.2 shows the evolution of T_{ASAX} and the response time of the honeytank over time.

We can see that T_{ASAX} grows linearly during 10 seconds. Its peak value is about 2,500 microseconds. At this point the generator has initiated its last session and the first session has reached its timeout value of 10 seconds. After this point, the generator remains silent and the honeytank progressively discards inactive sessions. Since a session is initiated ever 10,000 microseconds while T_{ASAX} always remains under 2,500 microseconds, packets are processed rapidly enough and do not accumulate into the queue. A rule simulating a session is thus created each 10,000 microseconds.

This figure clearly shows that T_{ASAX} grows linearly in the number of rule instances. Since the response time mostly depends on T_{ASAX}, it also grows linearly in the number of rule instances. As explained earlier, the peaks in the response time are due to the stalling of the libpcap library and to the scheduling policy of the operating system. We can observe that the response time of the honeytank equals to about three times its T_{ASAX} on average. Since network packets are scarce, the format adapter generates many fake records. On average, a real network packet arrives in the middle of the processing of a fake record. The real packet cannot be processed after this fake record because the format adapter that simulates foreseeing abilities has already announced that the next record is a fake one. Thus, the real network record will only be processed after this second fake record. On average, processing a real packet takes from two to three cycles, hence the response time approaching three times the T_{ASAX}.

We then redo this experiment with a *long* and *fast* test in order to stress the honeytank. Figure 10.3 shows the evolution of T_{ASAX}, the response time and the network queue over time. The T_{ASAX} grows rapidly and reaches over 25,000 microseconds. The honeytank is unable to cope with the packet arrival rhythm and packets starts to accumulate in the queue. About two minutes after the start of the test, all SYN packets have been processed. The honeytank does not have to create new rule instances and T_{ASAX} stabilizes itself. The ACK packets are being processed and the T_{ASAX} decreases as the honeytank discards unused sessions.

This figure shows that the response time of the honeytank grows faster than linearly in the number of rule instances. Indeed, the response time

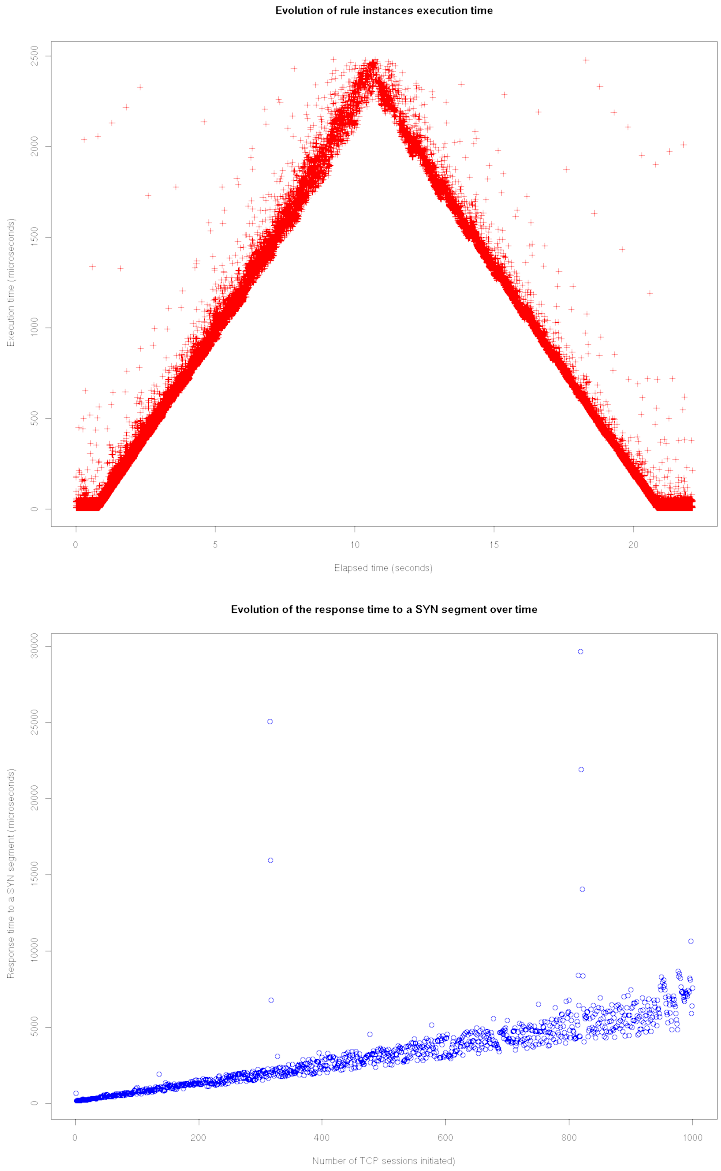

Figure 10.2: Non-optimized stateful honeytank executed by the non-optimized version of ASAX – Evolution of TASAX and response time for a *short* and *slow* test

depends on both the T_{ASAX} of the honeytank and the time a packet waits in the network queue. In this test, the queue accumulates up to 9,000 packets and the response time reaches over 150 seconds.

10.3.3 Optimized stateful honeytank executed by the optimized ASAX

For this test, we assess the behavior of HT_{opt} executed by $ASAX_{opt}$ with a *short* and *slow* test. Figure 10.4 shows the evolution of T_{ASAX} and the response time of the honeytank over time.

Unlike the previous test, T_{ASAX} remains rather stable over time. Its value is on average 21 microseconds and never goes over 400 microseconds. Similarly, its response time is stable and is about 375 microseconds. This figure shows that, with the optimized version of ASAX, T_{ASAX} is bounded and is quite low.

We then redo this experiment with a *long* and *fast* test in order to stress the honeytank. Figure 10.5 shows its result. Both the response time and T_{ASAX} suffers from several peaks. However, we explained earlier that calls to the `libpcap` library stall the execution of ASAX on some occasions. This produces the peak observed in this figure. Globally, T_{ASAX} remains under 400 microseconds. Its average value is 120 microseconds.

10.3.4 Non-optimized stateful honeytank executed by the optimized ASAX

This test assesses the behavior of HT_{no-opt} executed by $ASAX_{opt}$ with a *short* and *slow* test. We do not show graphs for this test as the results obtained are similar to the configuration where $ASAX_{no-opt}$ is used. The response time and T_{ASAX} both grow linearly in the number of rule instances.

It means that, in order to benefit from the optimizations, RUSSEL rules must be specially written for the optimized version of ASAX. However, the performance of $ASAX_{opt}$ executing non-optimized rules is not worse than the performance of $ASAX_{no-opt}$. Consequently, the optimized version of ASAX can be safely used in all the cases.

10.3.5 Optimized stateful honeytank executed by the non-optimized ASAX

Finally, we assess the behavior of HT_{opt} executed by $ASAX_{no-opt}$ with a *short* and *slow* test. The test results are similar to the configuration where HT_{no-opt} is used as the response time and T_{ASAX} are linear in the number of rule instance. However, the absolute values of this test are twice faster as the test where HT_{no-opt} is used. It might be due to the fact that the programming techniques used to write optimized rules force the programmer to write more structured code. It means that no matter the version of ASAX used, the programmer can directly write rules tailored for the optimized version of ASAX.

10.4 Analysis of the ASAX cycle time

In the previous section, we have compared several versions of the stateful honeytank run with two different versions of ASAX. Figures 10.4, and 10.5 suggest that the time needed to execute an ASAX cycle is quite stable, independently of the number of active rule instances. In Section 10.5, we make use of this property to sketch a simple mathematical model allowing us to predict the performances of a traffic handler (e.g., a generator or a stateful honeytank) for a class of practical situations. This model is used later on, in Section 10.6, to analyze the results of various efficiency tests conducted with the traffic generator, the stateless and stateful honeytanks, and Honeyd.

The mathematical model we propose, in Section 10.5, makes the assumption that the time needed by a traffic handler to treat a single *incoming* packet is a constant. This constant may vary depending on the type of each packet because the ASAX rules do different things for different kinds of packets. The important point is that, given a specific traffic handler, implemented as a particular RUSSEL module, the time needed to handle a packet is independent of the number of active rules. The time needed to handle a packet, denoted by T_{ASAX}, is the execution time of the ASAX cycle during which this packet constitutes the current record analyzed by all active rules. Independence with respect to the number of active rules means that only a fixed number of active rules, determined by the type of the packed, is executed regardless of the total number of active rules. The number of active rules may be huge but it does not impact the execution time of the ASAX cycle since only the "useful" rules are executed (see [27] for details about how this is achieved).

In this section, we provide a finer analysis of T_{ASAX} to support the assumption that it may be considered as constant, given a specific kind of incoming packet and a specific RUSSEL analysis module. Figure 10.6 gives a different view of the execution times given in Figures 10.4 and 10.5. This test was performed with the generator placed on the fastest machine. We distinguish between the time spent in the format adapter (noted VREAD) and the time needed to process the current record with the active rules. For the later time, we distinguish to cases: the current record is a "real" record, i.e., it contains a packet to be analyzed, or it is a "fake" record, i.e., a record generated due to the absence of an incoming packet. We have measured the execution times for every packet and we count how many packets fall inside pre-defined intervals of times. Interval of times of the form $[2^n + 1..2^{n+1}]$ (microseconds) are chosen, for readability. Each such interval is noted 2^{n+1} on the figure. We see on Figure 10.6 that all times are small compared to the execution times of active rules for "real records". The time spent in the format adapter is smaller than 32 microseconds for most records. It has been computed that this time is equal to 20 microseconds, on the average, for

"real" records. We also see that the execution time of active rules is smaller than 64 microseconds for most "fake" record. It is greater for "real" records (which are much fewer) and equal to 140 microseconds at the average. Thus, we can assume that for "real" records, an ASAX cycle takes $20 + 140 = 160$ microseconds independently of the number of active rule instances.

To further support our assumption that T_{ASAX} is a constant for the incoming packets of a specific traffic handler, we compare in Figures 10.7 and 10.8 the analysis times of "real" records, by the stateful honeytank, executed both with the optimized and the non optimized versions of ASAX. We see that those times are much smaller for the optimized version: they mostly range from 65 to 256 microseconds while the times for the non optimized version range from 65 to 65536, showing that a constant T_{ASAX} is indeed a specificity of the optimized version of ASAX. A non optimized system has a cycle execution time proportional to its number of active rules, not to its number of "useful" active rules.

10.5 Predicting and ensuring the performance of a traffic handler: a simple model

In this section, we present a simple mathematical model to predict the performance of a traffic handler or of two of them interacting together. Since this model is based on a number of strongly simplifying hypotheses, we validate our conclusions with a simple simulator of two interacting traffic handlers: a generator G and a honeytank H. In Section 10.5.1 we explain how we define and compute the response time of a traffic handler, based on the discussion of the previous section. In Section 10.5.2, we propose a model of transactions for traffic handlers. The conclusions of our reasonings are valid as far as this model is applicable. In Sections 10.5.3 and 10.5.4, we provide formulas to compute the response time of a traffic handler. We consider two kinds of traffic handlers: bounded and unbounded ones. Bounded traffic handlers limit the size of their incoming queue to guarantee their response time; thus, they have to give up part of the transactions they are involved in. Section 10.5.5 compares the efficiency results predicted by the mathematical model with measures obtained by simulation. We show that they are very close.

10.5.1 Response time of a traffic handler

We postulate that the main parameter to check for a traffic handler is its *response time*. We define the response time as the difference between the date of the arrival of a packet and the date of the handling of the packet by the traffic handler (e.g., responding with another packet or closing a TCP session). Maintaining a stable and reasonably small response time is needed

to ensure that the traffic generator interacts properly with its environment. In the case of a honeytank, for instance, it is needed to make it look realistic. Thus our goal is twofold: (a) predicting the response time of a traffic handler in a given context and (b) finding ways to maintain its performance in case of heavy traffic.

In all this section, we assume that we use traffic handlers designed for the optimized version of ASAX. We suppose that the duration T_{ASAX} of an ASAX cycle is constant. Therefore, the worst case response time of a traffic handler is $T_{ASAX} \times f$ where f is the length of the incoming queue of the traffic handler at this point of time. So, our main objectives boil down to (a) compute f and (b) find a way to limit f when the traffic is too intense.

10.5.2 Modeling the interactions of a traffic handler with its environment

A simple model of transactions

To model the interactions of a traffic handler with its environment, we need adequate parameters. The incoming packet rate is a first candidate: If the time between the arrival of two successive incoming packets for the traffic handler is less that an ASAX cycle T_{ASAX}, it is impossible to maintain a constant response time while keeping correct all functionality of the traffic handler. Nevertheless, the incoming traffic rate is not a very usable parameter because it may vary depending on the response of the traffic handler. A better parameter is the *number of transactions initiated during a time unit*. We use the word *transaction* as a generic term to designate a "complete interaction" between the traffic handler and its environment. Depending on what we want to achieve with the traffic handler, we may define differently what a transaction is. For instance, a transaction can be a complete TCP session or it can be a 3-way handshake, or yet more simply the discarding of a wrong incoming packet.

We model the system composed of the traffic handler and its environment by assuming two interacting systems G and H. Both G and H may represent the environment or the traffic handler depending on the concrete situation. The only distinction between the two is the fact that G initiates the transactions (of course, assuming that only one system initiates transactions while the other only responds is another big simplification). We characterize a transaction by a finite string of the two letters g and h, denoting a packet received by respectively a generator and a honeytank. For instance, the transaction corresponding to the test where G opens TCP/IP sessions and then close them immediately is modeled by the string *hgh-hgh*. The four occurrences of h correspond to the SYN, ACK, FIN, and ACK sent by the generator while the two occurrences of g correspond to the SYN+ACK and FIN+ACK replied by the honeytank. To simplify our

reasoning further, we assume that the total time needed to complete a single transaction does not depend on the particular order in which the packets are exchanged between G and H, e.g., transactions *ghhhgh*, *hhhhgg*, *hgghhh* are equivalent. We show later with our simulator that this assumption is reasonable when the number of completed transactions is large enough.

Parameters

We characterize a system as described above with the following parameters.

1. Inter-transaction time: T_I. This parameter is the average time between the starting dates of two successive transactions. To simplify things further, we assume that there is a constant amount of time between the starting dates of two transactions.

2. Number of incoming packets by transaction for H: n_H. This parameter is the number of packets received by H during a single transaction. Here again, we assume a constant number, which can be computed as a mean of all actual values.

3. Time needed by H to handle a single incoming packet: t_H. Once more, we assume it is a constant. In the case of our traffic handlers, this is a reasonable assumption provided that we use the optimized version of ASAX and a traffic handler designed for it. Indeed, every incoming packet is processed in a single ASAX cycle, which takes T_{ASAX} time. Notice that T_{ASAX} may be different for G and H depending on their functionality and the machine they are running on. In case we use H to stand for the external environment (the Internet), we assume that $t_H = 0$ because we can see the Internet as an infinite parallel machine able to process an infinite number of packets in parallel.

4. Number of incoming packets by transaction for G: n_G.

5. Time needed by G to handle a single incoming packet: t_G.

From these independent parameters, we define two derived ones. They simplify some formulas we write later on.

1. Total time spent by H to process a single transaction: T_H. Obviously we have $T_H = t_H \times n_H$.

2. Total time spent by G to process a single transaction: T_G. We have $T_G = t_G \times n_G$.

10.5.3 Performance of a single traffic handler

We first analyze the performance of a single traffic handler, independently of the other. For the sake of the discussion, we choose to reason about G but the same reasoning applies to H. It may help to think of H as the whole Internet, here.

Computing the response time

Since we do not take into account the order of packets in the transaction, we assume that G receives "its packets" immediately at the start of the transaction. Since the response time of G is equal to its T_{ASAX} times the length of its queue of incoming packets, f_G, we just need to compute f_G. There are only two cases to consider.

1. $T_I \geq T_G$

   In this case, G is able to complete a transaction before the start of another one. f_G is permanently equal to 0 or to 1. The response time is equal to t_G (T_{ASAX} of G).

2. $T_I < T_G$

   In that case, G is not able to complete a transaction in time. The queue of incoming packets is permanently growing. Assuming that the first transaction starts at time 0, the length of the queue at time t is equal to the number of received packets minus the number of processed packets. The number of received packets is equal to the number of initiated transactions ($\frac{t}{T_I}$) times the number of packets in a transaction (n_G). The number of processed packets is equal to $\frac{t}{t_G}$. Thus, we have:

   $$\begin{aligned} f_G &= \frac{t}{T_I} \times n_G - \frac{t}{t_G} \\ &= \frac{t(T_G - T_I)}{t_G T_I} \end{aligned}$$

   The response time is just

   $$f_G \times t_G = \frac{t(T_G - T_I)}{T_I}.$$

   It grows linearly over time.

Time to recover from a traffic peak

A traffic peak takes place when T_G becomes momentarily greater than T_I. It may happen during a scan, for instance. To model this simply, we assume that a heavy traffic lasts during a period of time T. Then it stops completely.

At that moment, the length of its incoming queue is $\frac{T(T_G-T_I)}{t_G T_I}$. The time needed to recover an empty queue is

$$f_G \times t_G \;=\; \frac{T(T_G - T_I)}{T_I}.$$

In fact, it is just the time needed to respond to the last packet of the peak. The duration of the crisis is

$$T + f_G \times t_G \;=\; \frac{T \times T_G}{T_I}.$$

We now turn to the problem of trying to accelerate a traffic handler in case of a crisis. We take the position that the traffic handler must give up part of the transactions, to be able to handle the others properly, maintaining a bounded response time. We have to distinguish two different cases because a traffic generator G can simply stop to initiate new transactions while a honeytank must at least deal with the first packet of the transaction.

Guaranteeing the response time of a traffic generator

In the case where $T_I < T_G$, there may be a problem with the response time of a traffic generator but it is always possible to guarantee this response time by limiting the size f_G of its incoming queue in such a way that $f_G \times t_G$ is smaller or equal to the desired response time. This can be obtained by first starting new transactions at any rate (choosing any T_I) but ceasing to initiate new ones as soon as f_G reaches its fixed bound. In any case, the number of initiated transactions is approximately equal to the one obtained by choosing $T_I = T_G$. Increasing the maximal size of the incoming queue has a marginal effect on the number of initiated transactions but it badly impacts the response time. Thus, in theory, it is useless to use a traffic generator with an inter-transaction time T_I smaller than T_G but, in practice, it is easier to check the size of the queue than to fix T_I as equal to T_G because we do not always know t_G with certainty.

Let us justify these claims. We denote the number of initiated transactions at time t by ntr. At any time t, the following relation holds

$$ntr \times n_G - \frac{t}{t_G} \;=\; f_G$$

Indeed, our hypothesis that the order of packets in a transaction is irrelevant allows us to assume that G has already received all its packets for the initiated transactions. Thus its queue contains all this packets minus the one already processed. From this equality, we compute

$$ntr \;=\; \frac{f_G \times t_G + t}{T_G}$$

Since f_G is bounded, the term $f_G \times t_G$ becomes negligible when t grows. So we can write

$$ntr \approx \frac{t}{T_G}$$

This shows that by limiting the size of the queue to any fixed bound we get a real inter-transaction time approximately equal to T_G (equal to T_G at the limit).

Guaranteeing the response time of a honeytank

Since a honeytank only responds to incoming traffic, it may not limit the number of initiated transactions. The best it can do is to stop new transactions as soon as possible when a crisis is detected. In the following, we assume that the honeytank is able to react in such a way that it does receive only the first packet of all unwanted transactions. To reach this goal, it may refuse to answer to this first packet or it may send a special packet to stop the transaction (e.g., a RST). A crisis may happen only if $T_I < T_H$, which we assume from now on. Thus we only consider two situations:

1. $T_I < t_H$

   In this case, the incoming rate of the first packet of new transactions is so high that stopping the transactions does not solve the problem: the queue of the honeytank will increase anyway. There is no way to guarantee the response time of the honeytank.

2. $t_H \leq T_I < T_H$

   In this case, the input queue length can be bounded by any number greater than zero. The number of accepted transactions does not depend much on this bound either. It mostly depends on T_H and T_I. Let us denote the number of accepted transactions by *ntr* and the number of stopped transactions by *nst*. The two following equalities hold:

   $$f_H = ntr \times n_H + nst - \frac{t}{t_H}$$

   $$ntr + nst = \frac{t}{T_I}$$

   The first equality states that the size on the incoming queue is equal to the total number of received packets minus the number of processed packets. The second equality just says that any initiated transaction has been either accepted or stopped. Eliminating *nst* from this system, we get the new equality

   $$ntr = \frac{1}{n_H - 1}\left(f_H + \frac{(T_I - t_H)t}{t_H \times T_I}\right)$$

The number of accepted transactions by unit of time is

$$\frac{ntr}{t} = \frac{1}{n_H - 1}\left(\frac{f_H}{t} + \frac{T_I - t_H}{t_H \times T_I}\right)$$

Since f_H is bounded, we get, when t is big enough,

$$\frac{ntr}{t} \approx \frac{T_I - t_H}{t_H \times T_I(n_H - 1)}$$

Let us denote the average time between the starting dates of two accepted transaction by T_A. We thus have

$$T_A \approx \frac{t_H \times T_I(n_H - 1)}{T_I - t_H}$$

If T_I is close to t_H, T_A is close to ∞: almost no transaction is accepted. If T_I is close to T_H, T_A is close to T_I: almost all transactions are accepted.

10.5.4 A closer look at G and H working together

We now analyze the performance of a traffic generator G and a honeytank H fighting against each other. To be concrete, we imagine an experiment where the traffic generator initiates transactions during a period of time T. Then it stops initiating transactions definitively. We compute the duration D needed to complete the whole test, i.e., the difference between the starting time of the first transaction and the completion time of the last one. We argue that

$$D = \frac{T \times \max(T_G, T_H, T_I)}{T_I}$$

The simplest way to justify our claim is to say that we have two traffic handlers collaborating to complete common transactions. Hence, D is the maximum of the times needed by G and H to complete their own part of the work. Using the formula derived previously for a single traffic handler, we get:

$$\begin{aligned} D &= \max\left(\frac{T \times \max(T_I, T_G)}{T_I}, \frac{T \times \max(T_I, T_H)}{T_I}\right) \\ &= \frac{T \times \max(T_G, T_H, T_I)}{T_I} \end{aligned}$$

To better justify this formula, we can distinguish several cases by comparing the values of T_G, T_H, and T_I. The most difficult case arises when $T_I < \min(T_G, T_H)$. Let us assume without loss of generality that $T_H \leq T_G$. Using once again our hypothesis that the ordering of packets in a transaction is irrelevant, we can suppose that, for every transaction, H receives all its packets before responding to G. Thus, for G, it looks like H is initiating

transactions with an inter-transaction time equal to T_H, during a time period equal to $\frac{T \times T_H}{T_I}$. Therefore, we get:

$$\begin{aligned} D &= \frac{\frac{T \times T_H}{T_I} \times T_G}{T_H} \\ &= \frac{T \times T_G}{T_I} \\ &= \frac{T \times \max(T_G, T_H, T_I)}{T_I} \end{aligned}$$

10.5.5 Validating our results by simulation

A simple simulator of G and H

To validate the previous reasoning we use a simple simulator of G and H, which maintains two queues containing pairs of integer numbers between 1 and the number of packets in a single transaction. These numbers are sufficient to decide when and what a (simulator of) traffic handler must reply to each incoming packet. The simulation consists of incrementing a counter t with a time unit equal to the greatest common divisor of t_G, t_H, and T_I. At each point of time, we modify the simulated system by removing and/or adding "packets" (i.e., numbers) to one or to both of the queues according to the state of the queues and to the description of transactions. Figure 10.9 is produced by our simulator; it depicts how the queues of G and H evolve over time during a test consisting of 6 transactions *hghhgh*. It is assumed that $T_I = 150$, $t_G = 100$, and $t_H = 50$ (we always assume that timings are given in microseconds, unless otherwise stated). The curve of f_H is drawn in red (continuous line); the curve of f_G is drawn in blue (discontinuous line). The maximal value of f_G is 3. Both queues become empty after $1,450$ microseconds. Figure 10.10 shows how the queues vary when the transactions are initiated during 10 seconds. Thus $66,666$ transactions are created. The maximal values of f_G and f_H are now $10,553$ and $21,104$, respectively. The queues become empty after $13,333,450$ microseconds.

The ordering of packets in transactions does not matter.

Table 10.9 shows the results of a number of simulations we have done to justify our hypothesis that the particular ordering of packets in a transaction has no influence on the total time needed by traffic handlers to complete a reasonably large number of transactions. The first seven columns of the table provide the parameters of the transactions. We have chosen these parameters to cover most situations that may arise. The reader may assume that the timings are given in microseconds. We initiate transactions during 10 seconds and we measure the total duration of the test both simulated

T_I	t_G	t_H	T_G	T_H	Case	Transaction	D_s	D_{th}
200	50	25	100	100	$T_G, T_H \leq T_I$	*hhhhgg*	10000025	10000000
						hghhgh	10000025	10000000
						hgghhh	10000025	10000000
200	50	100	100	400	$T_G < T_I < T_H$	*hhhhgg*	20000150	20000000
						hghhgh	20000100	20000000
						hgghhh	20000150	20000000
200	200	25	400	100	$T_H < T_I < T_G$	*hhhhgg*	20000125	20000000
						hghhgh	20000075	20000000
						hgghhh	20000125	20000000
150	100	100	200	400	$T_I < T_G < T_H$	*hhhhgg*	26666650	26666666
						hghhgh	26666550	26666666
						hgghhh	26666550	26666666
150	150	50	300	200	$T_I < T_H < T_G$	*hhhhgg*	20000050	20000000
						hghhgh	19999950	20000000
						hgghhh	20000050	20000000
150	100	50	200	200	$T_I < T_G = T_H$	*hhhhgg*	13333450	13333333
						hghhgh	13333450	13333333
						hgghhh	13333450	13333333

Table 10.9: Initiating transactions between G and H during 10 sec: test duration

(D_s) and predicted by the formula

$$D_{th} = \frac{T \times \max(T_G, T_H, T_I)}{T_I}$$

We observe that, in every case, all simulated and theoretical timings are very close, which really supports our reasoning and conclusions. Notice that the duration D of the test can be two times bigger or more that the period T during which transactions are initiated (except in the "good case", when $T_G, T_H \leq T_I$). This means that, as soon as T_G or T_H is greater than T_I, the response time of one of the two traffic handlers quickly becomes much too big to be acceptable: completing the last transaction may take more that 16 seconds.

The response time of G can be guaranteed.

Table 10.10 supports our claim that the response time of G can be guaranteed by stopping to initiate new transactions when the incoming queue of G reaches an arbitrarily fixed bound. We only consider situations where $T_G \geq T_H$ and $T_G > T_I$, because the queue of G remains most of the time empty otherwise. The actual values of t_G and t_H are the same as in Table

Case	f_G^b	f_G^m	ntr_{th}	ntr_s	ntr_{max}	D_s
$T_H < T_I < T_G$	1	2	25000	25001	50000	10000525
	10	11	25005	25008		10003275
	100	101	25050	25074		10029675
	1000	1001	25500	25770		10308075
	10000	10001	30000	32360		12944075
	∞	30902		50000		20000075
$T_I < T_H < T_G$	1	2	33333	33334	66666	10000350
	10	11	33338	33340		10002150
	100	101	33383	33409		10022850
	1000	1001	33833	34026		10207950
	10000	10001	38333	40300		12090150
	∞	41203		66666		19999950
$T_I < T_G = T_H$	1	2	50000	50000	66666	10000350
	10	12	50005	50015		10003250
	100	113	50050	50112		10022650
	1000	1136	50500	51279		10256050
	10000	10764	55000	66666		13333450
	∞	10764		66666		13333450

Table 10.10: Generator with bounded queue: actual number of transactions

10.9. We only make simulations for the transaction *hghhgh*. The column f_G^b gives the bounds we use for f_G. As soon as the bound is reached, the generator stops to initiate new transactions. The columns f_G^m give the maximal values actually reached by f_G. One can see that $f_G^b + 1 \geq f_G^m$ whenever $T_G > T_H$, which proves that stopping initiating new transactions when the fixed bound on f_G is reached is sufficient to ensure that the response time of G is bounded. When $T_G = T_H$, f_G^m may go more significantly beyond f_G^b. This is a limit case however, which is not very likely. The column ntr_{th} provides the number of transactions that should be initiated according to our formula

$$ntr_{th} \;=\; \frac{f_G \times t_G + t}{T_G}$$

The column ntr_s gives the actual numbers of simulations initiated by the simulator while ntr_{max} gives the maximal numbers of simulations that could possibly be initiated. We see that the number of initiated transactions must be significantly reduced to maintain the response time. Our "theoretical" formula is close to the simulation when f_G^b is not too big. In any case, it provides a lower bound to the number of initiated simulations. Finally, D_s provides the duration of the tests. We observe that it remains very close to the period of time during which transactions are initiated, which confirms that our method to guarantee the response time is effective.

Case	f_H^b	f_H^m	ntr_{th}	ntr_s	nst_s	ntr_{max}	D_s
$T_G < T_I < T_H$	1	3	16667	16668	33332	50000	10000550
	10	13	16676	16675	33325		10002600
	100	103	16766	16748	33252		10024500
	1000	1003	17666	17473	32527		10242000
	10000	10003	26666	24722	25278		12416700
	∞	43758		50000	0		20000100
$T_I < T_G < T_H$	1	4	11112	11112	55554	66666	10000450
	10	13	11121	11120	55546		10002750
	100	103	11211	11193	55473		10024650
	1000	1003	12111	11926	54740		10244550
	10000	10003	21111	19260	47406		12444750
	∞	70516		66666	0		26666550
$T_I < T_G = T_H$	1	3	44445	33334	33332	66666	10000200
	10	12	44454	44451	22215		10001150
	100	110	44544	44517	22149		10011050
	1000	1098	45444	45185	21481		10111250
	10000	10973	54444	51852	14814		11111300
	∞	21526		66666	0		13333450

Table 10.11: Honeytank with bounded queue: actual number of transactions

The response time of the stateful honeytank can be guaranteed

Similarly, Table 10.11 provides evidence that the response time of H can be often guaranteed by stopping accepting new transactions when the incoming queue of H reaches an arbitrarily fixed bound. We only consider situations where $T_H \geq T_G$ and $T_H > T_I$, because the queue of H remains most of the time empty otherwise. Again, the actual values of t_G and t_H are as in Table 10.9 and we only make simulations for the transaction *hghhgh*. The column f_H^b gives the bounds we use for f_H. As soon as they are reached, the honeytank refuses to accept new transactions. The columns f_H^m gives the maximal values actually reached by f_H. One can see that $f_G^b + 3 \geq f_H^m$ whenever $T_H > T_G$, which proves that refusing new transactions when the fixed bound on f_H is reached is sufficient to ensure that the response time of H is bounded in many situations. When $T_G = T_H$, f_H^m may go more significantly beyond f_H^b. This is an unlikely limit case, once again. The column ntr_{th} provides the number of transactions that should be initiated according to our formula

$$ntr_{th} = \frac{1}{n_H - 1}\left(f_H + \frac{(T_I - t_H)t}{t_H \times T_I}\right)$$

The column ntr_s gives the actual numbers of simulations initiated by the simulator while nst_s and ntr_{max} gives the number of refused transactions and the maximal numbers of simulations that could possibly be initiated. Here also our "theoretical" formula is close to the simulation when f_H^b is not too big. To the contrary of the previous case, it provides an upper bound to the number of initiated simulations. As before, D_s provides the duration of the tests, which here also remains close to the period of time during which transactions are initiated.

The limit case when $T_G = T_H$

Experiments with our simple simulator confirms the intuition that in most practical situations only one of the two fighting traffic handlers G an H may be facing a crisis. Either $T_I > \max(T_G, TH)$, in which case everything is fine for both G and H, or one of the two traffic handlers maintains an empty incoming queue while overwhelming the other one with its packets. The only limit case takes place when $T_G = T_H > T_I$. In such as a situation, the two incoming queues are permanently growing. This is quite unlikely as shown by the following experiment. We replay the test $T_I < T_G = T_H$ of Table 10.9 while making $t_G$vary slightly. Starting from the original value 100, we successively change it to 99, 97, 95, 94, and 93. These changes have almost no impact on the value of f_H (the red curve of Figure 10.9). They dramatically impact the curve of f_G, however, as shown in Figure 10.11. We see that the curves are flattening very quickly. For $t_G = 97$, the (blue) curve is two times smaller than the original one. For $t_G = 93$, the (magenta) curve is flat: the queue of G is permanently empty.

10.6 Experimental evaluation of performance: conformance to the theoretical model

In this section, we perform and analyze a number of experiments, made with the traffic generator (called G) and the stateful honeytank (called H). The results of these experiments are very close to what is predicted by the mathematical model of the previous section, supporting our claim that this model clearly describes and explains the efficiency issues for traffic handlers based on our approach.

10.6.1 Testing procedure

We have presented in Section 10.1 the configuration used to perform the tests. To assess these systems, we use the traffic generator presented in Chapter 9. It is executed by the optimized version of ASAX and the RUSSEL rules use the programming techniques presented in Section 9.3.3 to fully benefit from the optimizations. During 10 seconds, the generator initiates

TCP sessions towards hosts simulated by the honeypots. When the 3-way handshake of a session is completed, the generator closes it by issuing a `FIN+ACK` segment, waiting for the `FIN+ACK` reply of the hosts and issuing the final `ACK`. The generator starts by contacting the 12.0.0.1 IP on the port 80 with a segment seemingly originating from the port 1 of the 11.0.0.1 host. Each time a new session is initiated, the generator increments the IP destination, the IP source and the TCP port source. The TCP port destination is always 80. The test is repeated with different traffic generation rates. The generator initiates a new session each 1000, 500, 250, 125 and 75 microseconds.

For those tests, we use the traffic generator and the stateful honeytank in their bounded and unbounded configurations, thus totalizing four different combinations. In its unbounded configuration, the generator initiates sessions systematically. In the bounded configuration, sessions are not initiated when the network packet queue of the format adapter holds more than 10 packets. When the length of the queue goes under this threshold, the sessions are initiated again. In its unbounded configuration, the stateful honeytank process all incoming packets. In its bounded configuration, the stateful honeytank ignores `SYN` segments and does not create new hosts when the network packet queue of the format adapter holds more than 10 packets.

The honeytanks and the generator use a rule that counts the various types of packets received. This rule is independent of the simulation and is executed for each packet received. Moreover, the generator computes the average response time of the honeytank to `SYN` segments in the 3-way handshake phase as well as the response time to `FIN` segments in the connection closing phase. Finally, we instrumented the format adapter used by the honeytanks and the generator to export in a text file, for each event, the time spent in the format adapter and the time used to execute all rule instances for this event. The exported information also signals if the event was a real network event or a fake event generated by the format adapter (see Section 7.1.1).

10.6.2 Unbounded generator versus unbounded honeytank

In this section, we analyze the results of the tests with an unbounded generator G and an unbounded stateful honeytank H. Table 10.12 provides the parameters of the tests, using the terminology of Section 10.5: T_I is the inter-transaction time. t_G is the time needed by G to handle a single packet. We estimate t_G as the average value of T_{ASAX} for "real records", i.e., NADF records containing an incoming packet. T_G is the time needed by G to handle all the data segments of a single transaction. In this test, one has $T_G = 2 \times t_G$. t_H and T_H are the corresponding times for the stateful honeytank H ($T_H = 4 \times t_H$). All these times are given in microseconds. f_G^a

T_I	t_G	t_H	T_G	T_H	$Case$	f_G^a	D_G	f_H^a	D_H	D_{th}
75	74	431	148	1724	$T_I < T_G < T_H$	1	111.204	154179	230.581	229.866
125	72	219	144	876	$T_I < T_G < T_H$	1	62.444	84451	69.804	70.800
250	71	206	142	824	$T_G < T_I < T_H$	1	30.546	33275	33.596	32.960
500	77	193	154	772	$T_G < T_I < T_H$	0	14.959	8324	15.497	15.440
1000	74	146	148	584	$T_G < T_H < T_I$	0	9.998	2	9.999	10.000

Table 10.12: Unbounded generator versus unbounded stateful honeytank: average queue length and test duration

is the average size of the queue of incoming packets for G while D_G is the total duration of the test from the generator standpoint. D_G is computed as the difference between the system times when the last and first packets received by G are handled by it. f_H^a and D_H provide the same information for H. Finally, D_{th} is the expected duration of the test, according the mathematical study of Section 10.5, i.e.,

$$D_{th} = \frac{10 \times \max(T_G, T_H, T_I)}{T_I}$$

Test durations are measured in seconds. We can make the following observations.

- The times t_G and t_H are quite stable and independent from T_I as postulated in our mathematical study.[2] t_G is around 75 microsecs, while t_H is around 200 microsecs. Hence G is faster than H.
- Since G is faster and has less work to do than H (i.e., only two packets to analyze for each transaction, instead of four), we can expect that G will never be overwhelmed by incoming packets. Indeed, in all tests, we have $T_G < T_H$ and we see that the input queue of G has an average length of 1.
- The honeytank is able to handle all sessions immediately as long as $T_H < T_I$. This is the case only for $T_I = 1000$. In that case, the duration of the test is exactly equal to the period during which sessions are initiated by the generator.
- As soon as $T_I < T_H$, packets accumulate in the incoming queue of the honeytank. The length of the queue increases more and more when T_I diminishes. We see that the duration of the test for H is precisely predicted by our formula (D_{th}). The duration of the test is shorter for

[2]This is not completely true for t_H but we discovered that the problem stems from the way we handled session timeouts. We have fixed this problem but, due to lack of time, we could not redo all measurements.

T_I	$f_H^a \times t_H$	SYN_G	SYN_H	rtt_1	S+A	ACK	FIN_H	rtt_2	FIN_G	RST
75	66×10^6	133278	133273	$--$	133273	266355	133273	$--$	133082	12680
125	18×10^6	80000	79605	$--$	79605	159208	79604	$--$	79604	13863
250	6×10^6	40000	40000	$--$	40000	80000	40000	$--$	40000	11814
500	1×10^6	20000	20000	$--$	20000	40000	20000	$--$	20000	0
1000	292	10000	10000	834	10000	20000	10000	918	10000	0

Table 10.13: Unbounded generator versus unbounded stateful honeytank: number of exchanged segments, and response time

G because, when G has received the last FIN segment from H, most of the final ACK segments sent in response to those FIN segments are still waiting in the queue of H.

Now, let us have a look at Table 10.13. This table provides information about the actual number of data segments exchanged between G and H as well as information about the response time of H. The table contains the following items. SYN_G is the number of SYN segments sent by G. SYN_H is the number of such segments actually handled by H. rtt_1 is the response time of H to these SYN segments. This response time is computed as the difference between the system time of G when the SYN segment is sent and the system time of G when the corresponding SYN+ACK is received by the `libpcap` module. S+A is the number of SYN+ACK segments returned by H and actually received by G. ACK is the total number of ACK segments sent by G and actually received by H. FIN_H is the number of FIN segments sent by G that are received by H. FIN_G is the number of FIN segments sent in response by H and received by G. rtt_2 is the difference between the system time of G when a FIN segment is sent by G and the system time of G when the corresponding FIN from H is received by G. Finally, RST is the number of RST segments received by G from H. No other data segment has been received by G or H during these tests. We can make the following comments.

- The response time of H is good (less than a millisecond) as long as $T_H < T_I$. It becomes quickly unacceptable otherwise. The sign "$--$" in the columns rtt_1 and rtt_2 means that we got overflows when computing mean values with RUSSEL integers. Theoretical approximations for these values are provided in the second column of the table. These approximations are huge except when $T_I = 1000$. In that case, the approximation is significantly lower than the measured value. This can be explained by the fact that additional factors may impact the response time when packets are treated quickly. When the input queue is large, waiting in the queue of H is so long that all other factors can be ignored.

- We see that G always send the expected number of SYN segments except in one case. This agrees with our prediction since $t_G < T_I$. A few SYNs are not sent when $T_I = 75$. This is easily explained by the fact that t_G and T_I are almost equal in that case. (G initiates at most one session by ASAX cycle. This is just a design decision to avoid the situation where G should be overwhelmed because it starts too many transactions.)

- Very few data segments are "lost" in these tests even when the incoming queue of H is very large. This shows that our decision of "emptying" the buffer of the `libpcap` module as soon as possible is effective. A few packets are nevertheless lost inside the `libpcap` module when the traffic is high.

- RST segments are sent by the honeytank when the traffic is high ($T_I \leq 250$). This can be explained as follows. When H sends a SYN+ACK or FIN segment to G it awaits a ACK segment in reply. If this ACK takes too long to arrive, the corresponding session is closed by H. In fact, the problem is not that it takes too long to arrive but it stays too long inside the incoming queue. The problem arises mostly for FIN segments because SYN+ACK segments are retransmitted while FIN segments are not.

To close this section on the unbounded generator and honeytank, we provide some figures. Figure 10.12 shows how the incoming queues of G and H evolve over time for the test with $T_I = 1000$. It also shows the variations of T_{ASAX} for both G and H. Figure 10.13 provides the same information when $T_I = 250$. We observe some interesting facts.

- For $T_I = 1000$, the incoming queues of both G and H remain close to empty during the whole test, except in a few situations, which are quickly resorbed. We believe that those situations arise because of input/output operations performed by the operating systems: if the process running G or H is interrupted for a relatively long period, many packets may accumulate inside the `libpcap` buffer. When G (or H) is scheduled again it immediately empties the `libpcap` buffer and its incoming queue grows suddenly. However, since t_H is much lower than T_I the queue is emptied quickly.

- For $T_I = 250$, the incoming queue of G behaves similarly as for $T_I = 1000$. To the contrary, the incoming queue of H first grows linearly during 10 seconds and then more slowly. It decreases after about 21 seconds. Notice that this experimental curve looks quite similar to the curves obtained by simulation in Section 10.5.

- In all cases, the pictures describing T_{ASAX} are quite stable along time, in accordance with the experimental study presented in Section 10.3.

T_I	t_G	t_H	T_G	T_H	$Case$	f_G^a	f_H^a	ntr_{th}	ntr_r	ntr_{max}	D_G	D_H
75	89	83	178	332	$T_I < t_H$	0	11620	$--$	7	133333	0.475	11.198
125	86	109	172	436	$t_H \leq T_I \leq T_H$	0	31	3917	3397	80000	10.570	10.790
250	80	146	160	584	$t_H \leq T_I \leq T_H$	0	13	9501	9053	40000	10.269	10.425
500	78	176	156	704	$t_H \leq T_I \leq T_H$	0	10	12276	11861	20000	10.205	10.349
1000	74	140	148	560	$T_H < T_I$	0	0	10000	9849	10000	9.998	9.999

Table 10.14: Unbounded generator versus bounded stateful honeytank: average queue length, number of open sessions, and test duration

10.6.3 Unbounded generator versus bounded honeytank

We perform the same tests as in the previous section but we use a bounded version of the honeytank instead of an unbounded one. According to the mathematical model of Section 10.5 this would allow us to maintain an acceptable response time for the stateful honeytank even in the case of a high traffic, but at the price of refusing to open some sessions. We check whether practical experiments agree with the model.

Table 10.14 provides information about the tests. This table is similar to Table 10.12 but the incoming queue of H is limited to a length equal to 10. This does not mean that we drop all incoming packets when the queue length equals 10 but we refuse to open any new session or to create any new machine. Additionally to the information provided in Table 10.12, we provide the following measurements: ntr_{th} is the number of sessions that should be open according to the mathematical model, i.e.,

$$ntr_{th} = \frac{1}{n_H - 1}\left(f_H + \frac{10 \times (T_I - t_H)}{t_H \times T_I}\right)$$

ntr_r is the number of sessions actually open by H while ntr_{max} is the (theoretical) number of attempts made by G to open sessions. We can make the following observations.

- By limiting the size of the incoming queue of H we ensure that the total duration of the test is now close to 10 seconds in any case. Thus neither G nor H are overwhelmed by the incoming traffic.
- When $t_H < T_I < T_H$, the actual number ntr_r of open sessions is quite close to the theoretical number ntr_{th}. This number decreases when T_I does because H has more and more SYN segments to consider, just to ignore them afterward. Moreover, the average length of the incoming queue remains close to 10, until T_I becomes close to t_H.
- When $T_I < t_H$, the incoming queue of H cannot be limited in any way. Thus very few sessions are open. They are open at the very beginning of the test, which explains why D_G is so small when $T_I = 75$.

T_I	$f_H^a \times t_H$	SYN_G	SYN_H	rtt_1	S+A	ACK	FIN_H	rtt_2	FIN_G	RST
75	964460	133334	133334	1170615	7	14	7	26926	7	0
125	3379	80000	80000	1502	3397	6794	3397	1912	3397	0
250	1898	40000	40000	1885	9053	18106	9053	2027	9053	0
500	1760	20000	20000	2312	11861	23722	11861	2157	11861	0
1000	210	10000	10000	345	9849	19698	9849	429	9849	0

Table 10.15: Unbounded generator versus bounded stateful honeytank: number of exchanged segments, and response time

- Finally, we see that not all sessions are open by H even for $T_I = 1000$. This is the price to pay for using a bounded stateful honeytank: we have to refuse to open some sessions sometimes. If the `libnet` module was always delivering packets at the same speed, this phenomenon would not happen. To solve this problem, one may think of increasing the limit of the queue length but this is not always a good solution since it may unacceptably increase the response time of H sometimes.

We now analyze Table 10.15, which is the counterpart of Table 10.13 for the bounded stateful honeytank. Here also some interesting remarks can be done.

- Although a limited number of sessions are open by H, all sessions that are accepted are correctly open and closed. No packet is lost inside an accepted session. Thus, our policy for limiting the number of sessions is strict but effective.
- An acceptable response time is maintained during the open sessions and it is acceptably predicted by the formula $f_H^a \times t_H$. Although this formula is not completely precise, it correctly gives the order of magnitude of the response time. This is true only when $t_H < T_I < T_H$ however.
- Since the response time of H is now quite low, there are no RST segments sent by H anymore.

As in the previous section, we now provide pictures showing how T_{ASAX} and the incoming queue evolve. Figure 10.14 is the counterpart of Figure 10.12 with the bounded stateful honeytank. We see that all pictures are very similar in both figures. When $T_I > T_H$, limiting the incoming queue of H is not really needed since the queue is almost empty most of the time. Figure 10.15 is the counterpart of Figure 10.13 with the bounded stateful honeytank. The picture of the incoming queue is now completely different for H. Instead of growing up to more than $50,000$ packets, it remains close to 10, except for a limited number of values. No peaks are greater than

T_I	$f_H^a \times t_H$	SYN_G	SYN_H	rtt_1	S+A	ACK	FIN_H	rtt_2	FIN_G	RST
75	43×10^6	133281	133124	−−	133124	266248	133124	−−	133124	14047
125	18×10^6	80000	79591	−−	79591	159180	79590	−−	79590	14055
250	7×10^6	39857	39857	−−	39857	79714	39857	−−	39857	12141
500	1×10^6	20000	20000	−−	20000	40000	20000	−−	20000	0
1000	296	10000	10000	888	10000	20000	10000	962	10000	0

Table 10.16: Bounded generator versus unbounded stateful honeytank: number of exchanged segments, and response time

T_I	$f_H^a \times t_H$	SYN_G	SYN_H	rtt_1	S+A	ACK	FIN_H	rtt_2	FIN_G	RST
75	1×10^6	133333	133333	1161191	6	12	6	1900	6	0
125	4620	80000	80000	1599	2997	5994	2997	1872	2997	0
250	1911	40000	40000	1888	9044	18088	9044	2016	9044	0
500	1770	20000	20000	2129	11879	23758	11879	2158	11879	0
1000	213	10000	10000	351	9841	19682	9841	438	9841	0

Table 10.17: Bounded generator versus bounded stateful honeytank: number of exchanged segments, and response time

150 packets. This is consistent with the response time being close to 2 milliseconds, i.e., approximately $10 \times t_H$.

10.6.4 Bounded generator

Using a bounded version of the generator has a very low impact on the results described in the two previous section. Indeed, the generator is much faster than the stateful honeytank, partly because it is running on a faster computer, and partly because it has less work to do. We have chosen to run G on the fastest machine because our main goal here is to assess the honeytank performance, which can be done fairly only if the generator performance is not impacted by the traffic received from the honeytank. Thus, we present tests made with a bounded generator only for the sake of completeness.

Table 10.16 is the counterpart of Table 10.13 when using a bounded generator and an unbounded stateful honeytank. The generator queue length is limited to 10. There is no significant difference between the data provided by the two tables. Similarly, Table 10.17 reports on a bounded generator and a bounded honeytank. The results are almost the same as for an unbounded generator and a bounded honeytank (see Table 10.15).

10.7 Efficiency of honeytanks : comparison with Honeyd

In this section, we compare the efficiency of our honeytanks with Honeyd [47]. As explained in Section 10.2, we do not use the shell-based external service simulators of Honeyd for this efficiency test. On the one hand, because they seem bugged and, on the other hand, because, with those external services, Honeyd has to create an external process per established TCP connection. This would be much less efficient than using a single process as we do. Instead, we use a trivial echo service written in Python (i.e., a Honeyd internal service). A detailed presentation of the architecture of Honeyd, including the external and internal service simulators, is presented in Chapter 3.

For this comparison, we use our traffic generator to initiate TCP connections at different rates (i.e. with different T_I values) during 10 seconds. Once a connection is established, the generator closes it by issuing an appropriate `FIN` segment, waits for the reply of the tested system and then issues the final `ACK` segment. The tables showing the raw results of those tests are presented in Appendix B.

We first analyze the efficiency of one unbounded and three bounded stateful honeytanks, and of a stateless honeytank. Figure 10.16 shows the results of this test. The upper graph shows the number of correctly established and then closed TCP connections according to different generation rates. The lower graph presents the response time to accepted `SYN` segments. It is important to analyze the two graphs side-by-side. The average response time to `SYN` segments presented in the lower graph must be put in relation with the number of accepted `SYN` segments presented in the upper graph. Indeed, the response time of a honeypot can be improved by ignoring `SYN` segments.

We can observe that, for all honeytanks, all TCP connections that are established are correctly closed later on by the honeytanks. The unbounded stateless and stateful honeytanks are able to correctly accept and close all initiated connections. However, as T_I increases, their response time to SYN segments also increase. In those cases, the sessions are initiated faster than the honeytanks can process them. Thus, packets do accumulate in the network queue of the honeytanks, increasing the response time to those packets. Those behaviors were explained in the theoretical model presented in Section 10.5.

Similarly, we observe that the bounded honeytanks achieve a much lower response time by refusing to establish new connection when their network queue reaches a given threshold. However, the value of the threshold has little impact to the number of connections established. Figure 10.17 shows only the results for the bounded honeytanks and gives a clearer view of their

behavior. The bounded honeytanks open and close about 13,000 connections for an inter-generation time up to 500 μsec. The response time for the bounded honeytanks at 10, 100 and 1000 packets are respectively about 2 millisecond, 16 milliseconds and 150 milliseconds, as predicted by the mathematical model.

Figure 10.18 shows the behavior of Honeyd compared with the bounded honeytanks. We observe that not all connections established by Honeyd are correctly closed. For inter-generation times up to 500 μsec (i.e., 2,000 sessions initiated per second), the bounded honeytanks establish and close more TCP connections than Honeyd while having a much better response time, in the case of honeytanks bounded at 10 and 100 packets. For lower T_I, Honeyd opens more sessions than the bounded honeytanks but nearly none of them are correctly closed.

As explained in Chapter 4, the honeytanks maintain their own queue of incoming network packets. At each ASAX cycle, they extract the packets buffered by the libpcap and append them in their queue. Then, if the queue has reached a given threshold, the RUSSEL code of the bounded honeytanks decide to ignore the `SYN` segments. However, those segments have already been copied in the queue, translated into NADF and sent to the ASAX analyzer. Since no packet is lost, the drawback of this approach is that all `SYN` segments accumulated in the queue are processed even if they are eventually ignored by the RUSSEL rules. This increases the time during which the queue length is over the given threshold and thus lowers the number of established connections. However, the advantage of this approach is that there is nearly no packet loss. All `FIN` segments are processed and thus all established connections are correctly closed. Honeyd, however, does not maintain a packet queue on its own. It relies entirely on the libpcap. This allows Honeyd to open more sessions than the bounded honeytanks for very low values of T_I. Unfortunately, Honeyd looses many packets and thus miss lots of `FIN` segments.

To summarize, up to 2,000 sessions initiated per second, a bounded stateful honeytank is able to establish and close more sessions than Honeyd while achieving a much lower response time. From 4,000 sessions initiated per seconds, Honeyd opens more sessions than the bounded honeytanks but it does not close them correctly and its response time is much greater than the response time of bounded honeytanks.

10.8 Stress tests : testing the honeytanks and Honeyd with TCPReplay

In this section, we use TCPReplay to stress-test our honeytanks and Honeyd (using a Python internal service). We generate a network trace containing 20,000 SYN sent to 20,000 different IP addresses. We use TCPReplay

to replay this trace at various speeds. Using an independent tool such as TCPReplay to compare our honeytanks and Honeyd shows that the results are similar to the tests performed using our traffic generator and thus confirms their validity. To the contrary of the previous section, where TCP connections are fully established and then closed, this test is much simpler as only `SYN` segments are sent to the tested system.

Figure 10.19 shows the behavior of an unbounded stateful honeytank and of a stateful honeytank bounded at 1,000 packets. The upper graph shows the number of `SYN+ACK` segments received according to the rate at which `SYN` segments are sent. We can observe that the unbounded honeytank replies to all `SYN` sent. However, from 4,000 `SYN` sent per second, the queue of the bounded honeytank reaches the limit of 1,000 packets and the honeytank starts to ignore some of them. Up to 4,000 packets per second, the response time of both honeytanks is about 2,5 milliseconds. From this point, packets arrive faster than the honeytanks can process them and they accumulate in the queue.

Figure 10.20 shows the results for a stateful honeytank bounded at 1,000 packets and Honeyd. For both systems, the number of `SYN+ACK` replies starts to decrease when the sending rate reaches 4,000 packets per second. However, our honeytank sends more replies than Honeyd while offering a better response time. When more packets are sent per second, Honeyd sends slightly more replies than the honeytank. However, as explained in the previous section, segments that are not replied by Honeyd have been lost by the libpcap. Instead, since we maintain our own network queue, `SYN` segments that are not replied by the bounded honeytank have still been read, converted in NADF and ultimately ignored by the RUSSEL rule. For this particular test, a better strategy would have been to stop appending packets in the network queue when it has reached its threshold instead of catching all possible packets and then ignore them at the RUSSEL level.

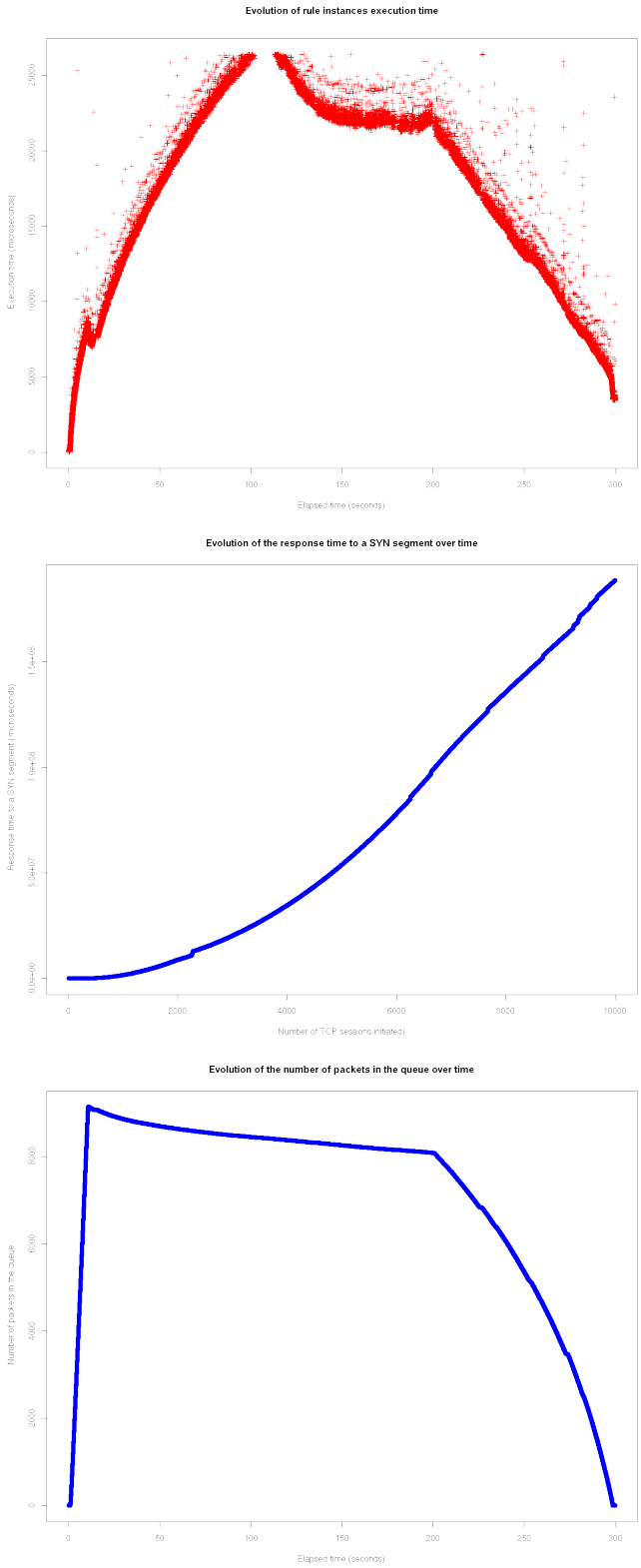

Figure 10.3: Non-optimized stateful honeytank executed by the non-optimized version of ASAX – Evolution of TASAX and response time for a *long* and *fast* test

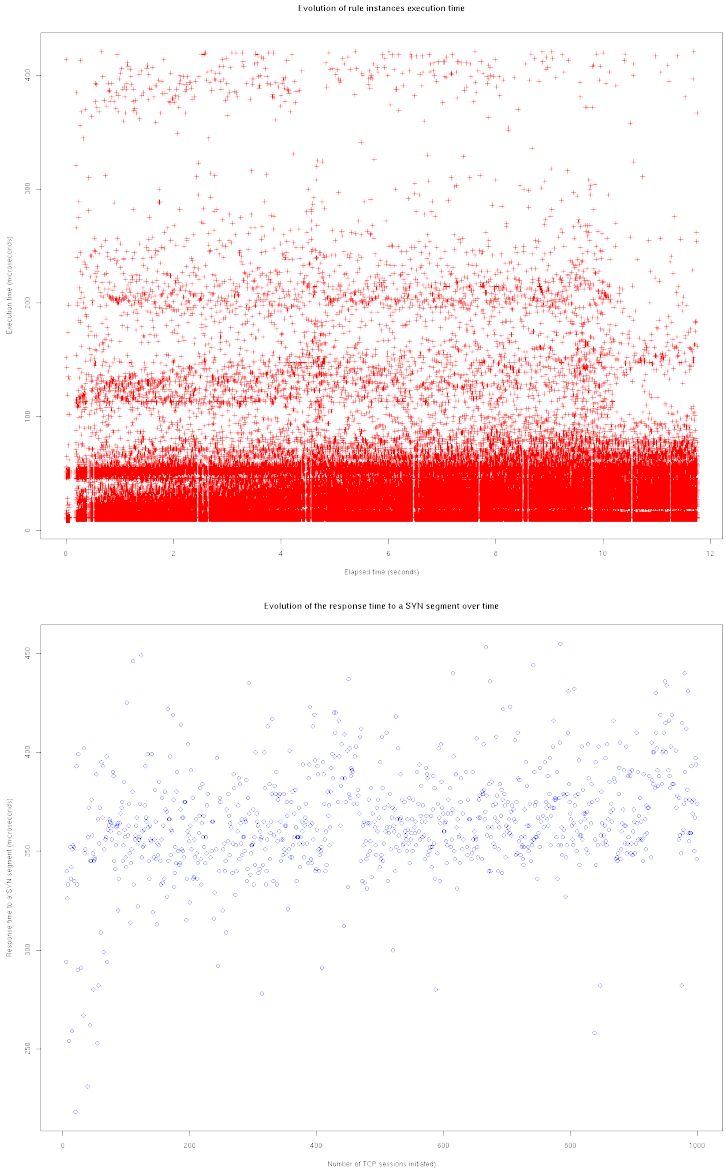

Figure 10.4: Optimized stateful honeytank executed by the optimized version of ASAX – Evolution of TASAX and response time for a *short* and *slow* test

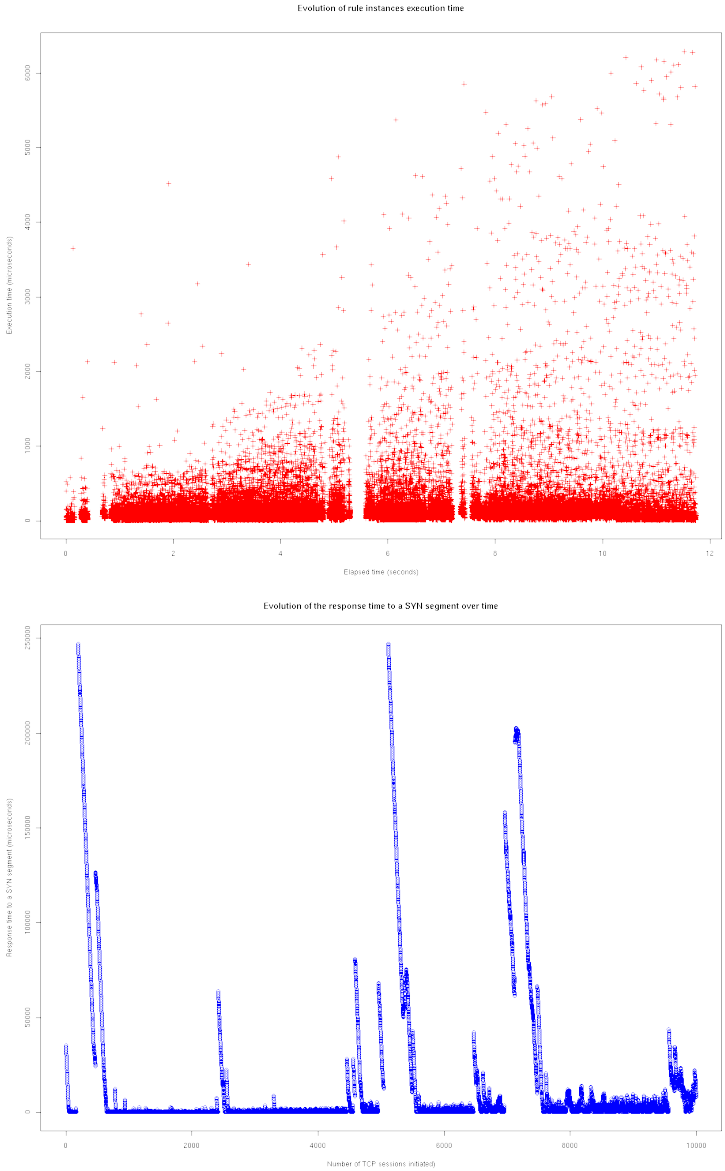

Figure 10.5: Optimized stateful honeytank executed by the optimized version of ASAX – Evolution of TASAX and response time for a *long* and *fast* test

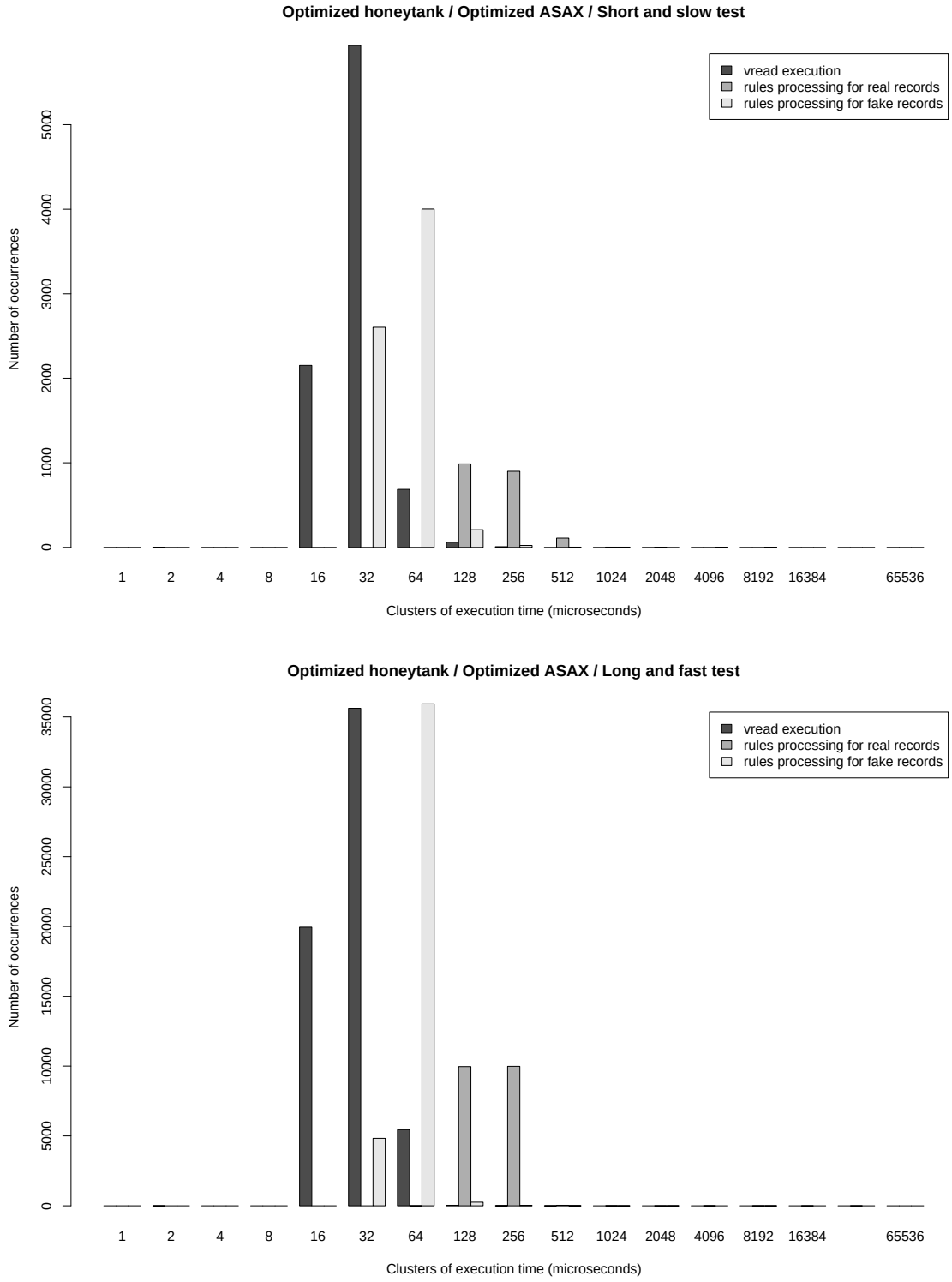

Figure 10.6: Analysis of the ASAX cycle time of the traffic generator and the optimized stateful honeytank

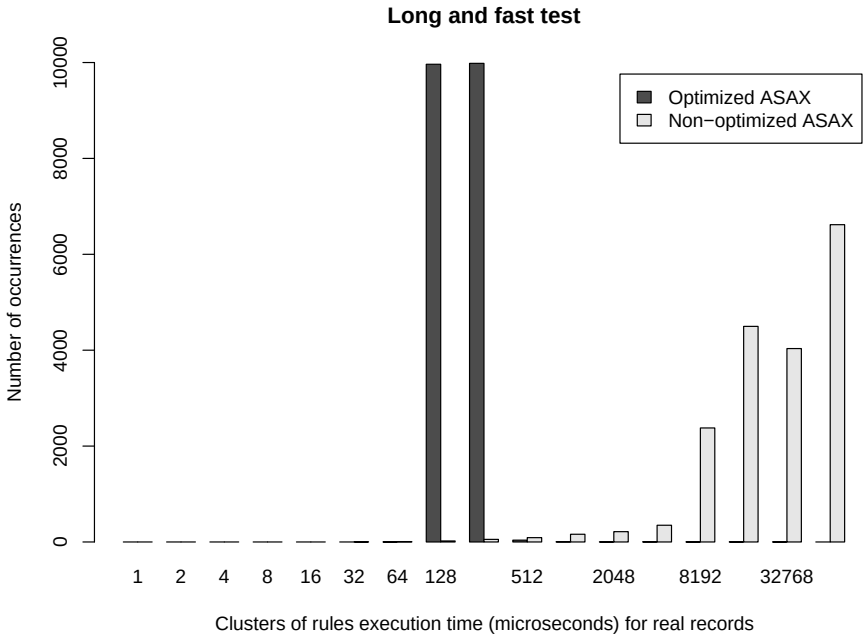

Figure 10.7: Analysis of the ASAX cycle time of the stateful honeytank on a long and fast test

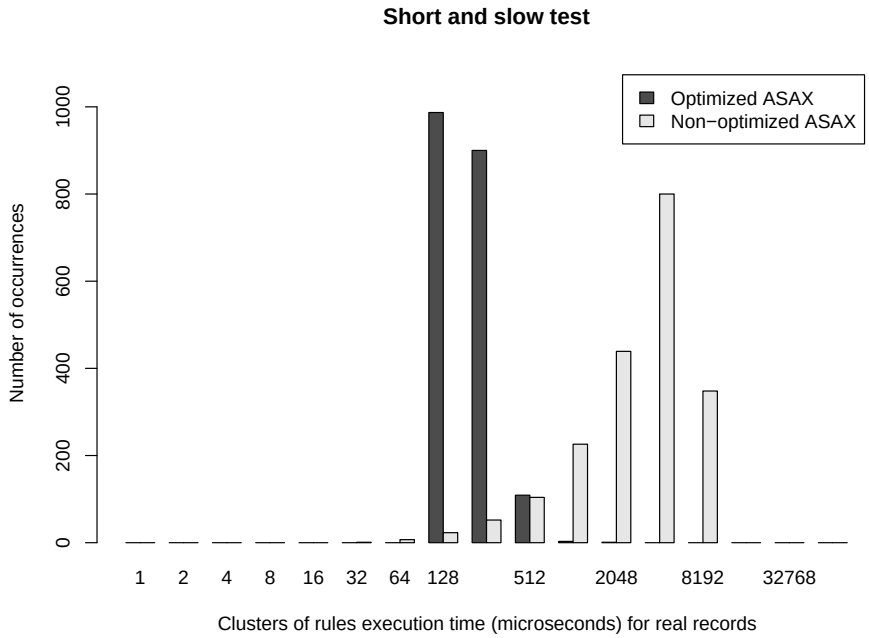

Figure 10.8: Analysis of the ASAX cycle time of the stateful honeytank on a short and slow test

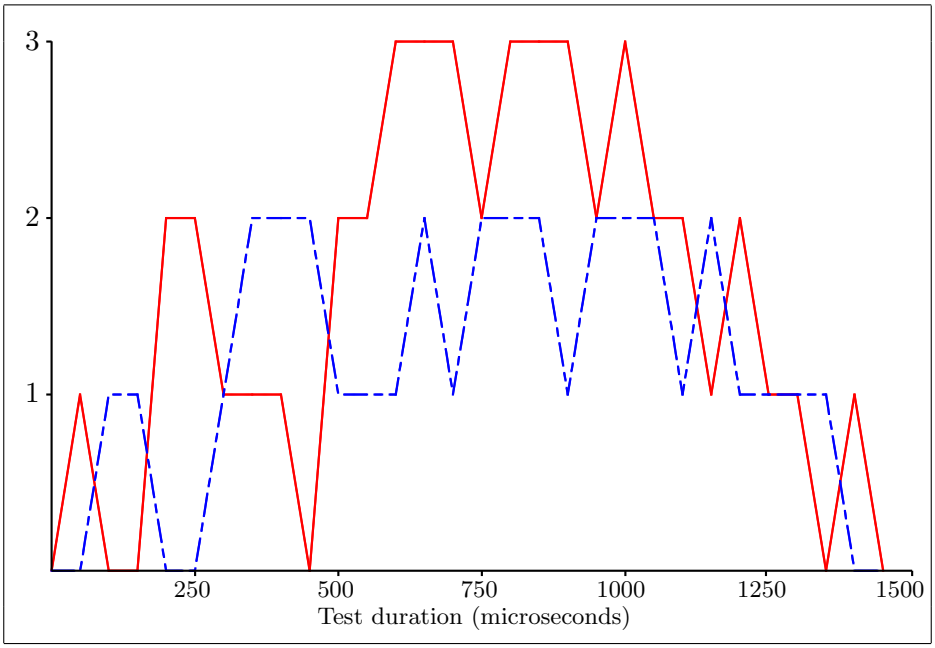

Figure 10.9: Queue lengths for G and H (1 millisecond)

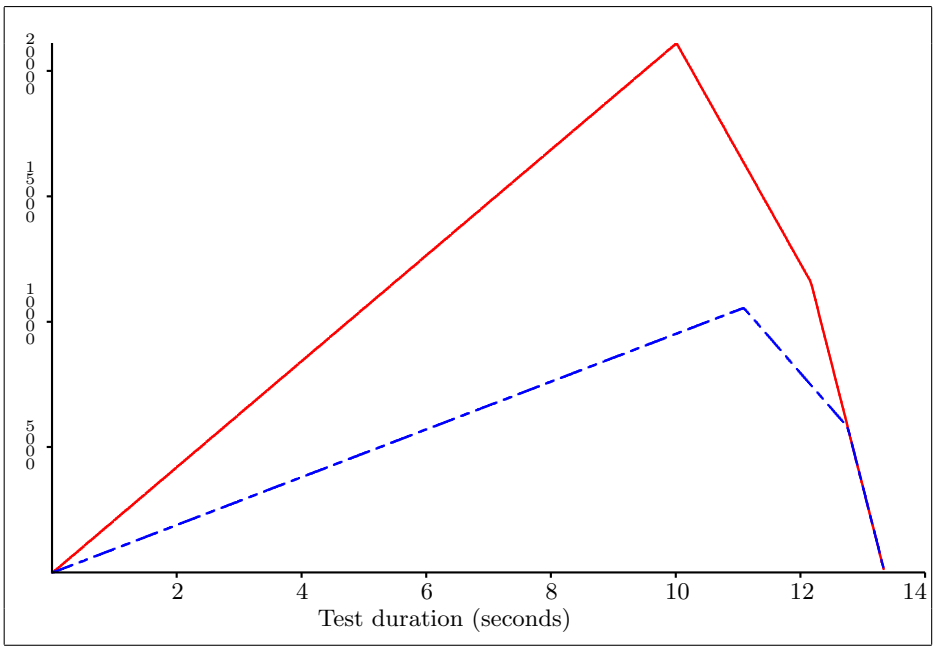

Figure 10.10: Queue lengths for G and H (10 seconds)

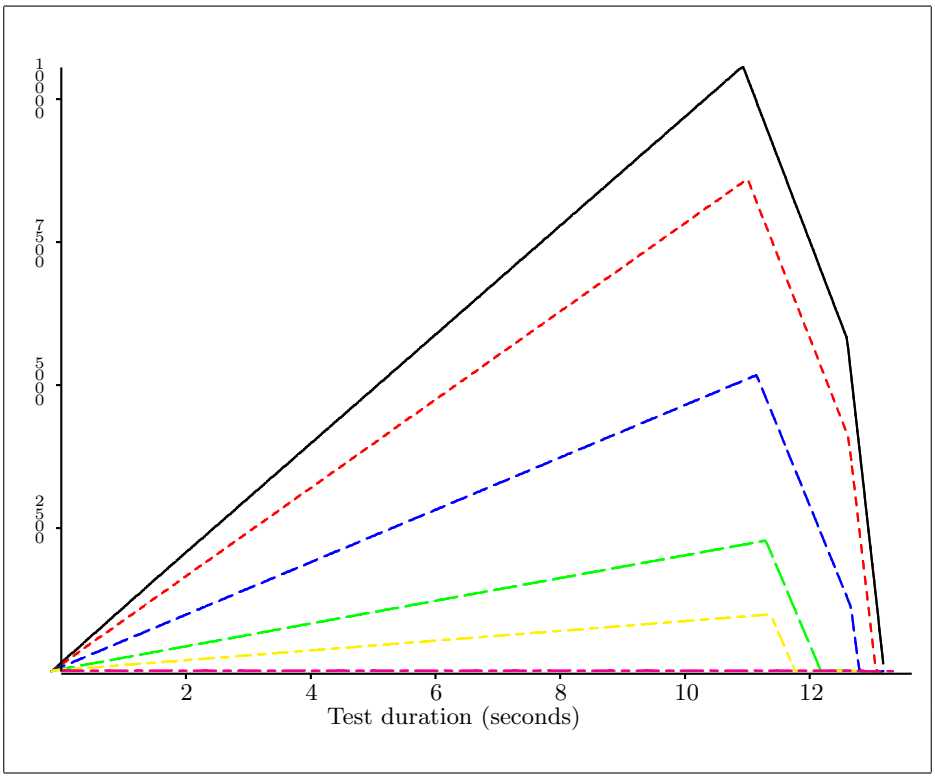

Figure 10.11: T_G becomes slightly smaller than T_H: the queue of G vanishes.

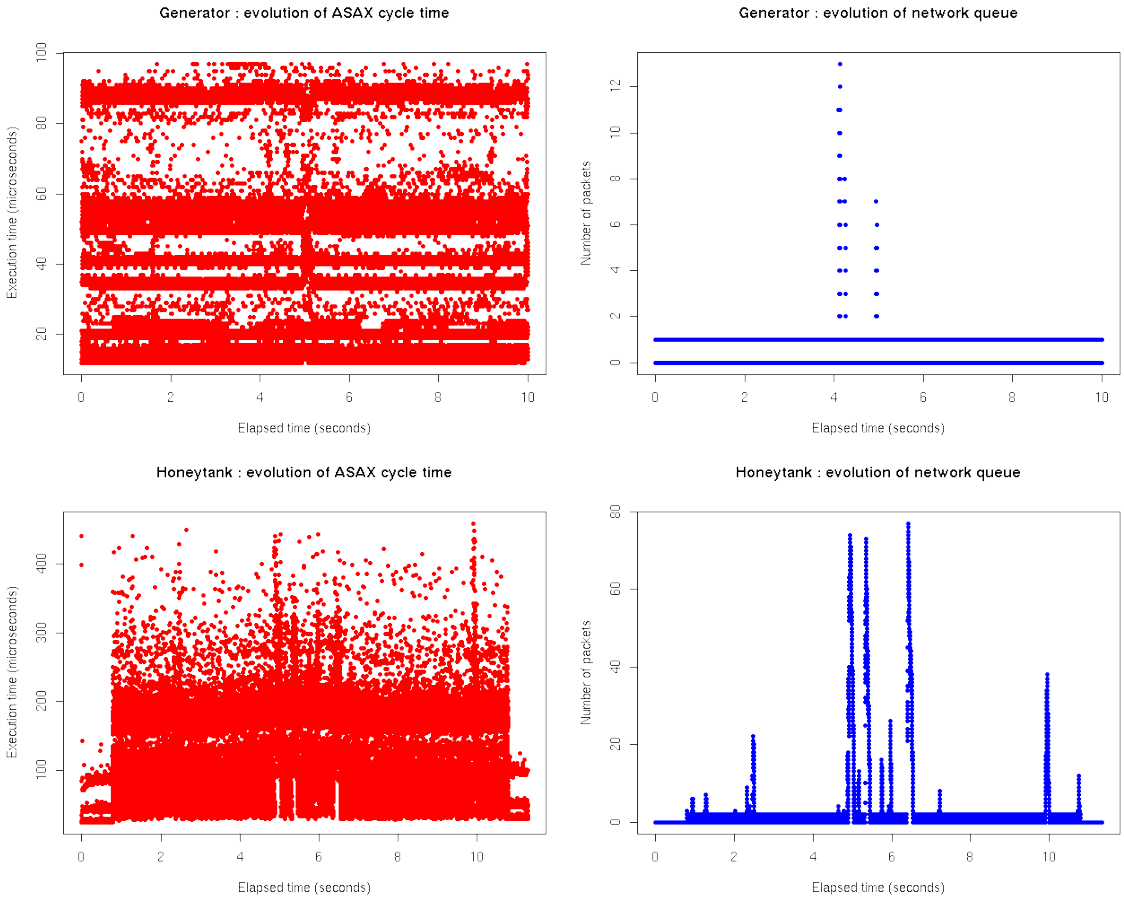

Figure 10.12: Unbounded generator Versus Unbounded stateful honeytank - Generation interval: 1000 μsec

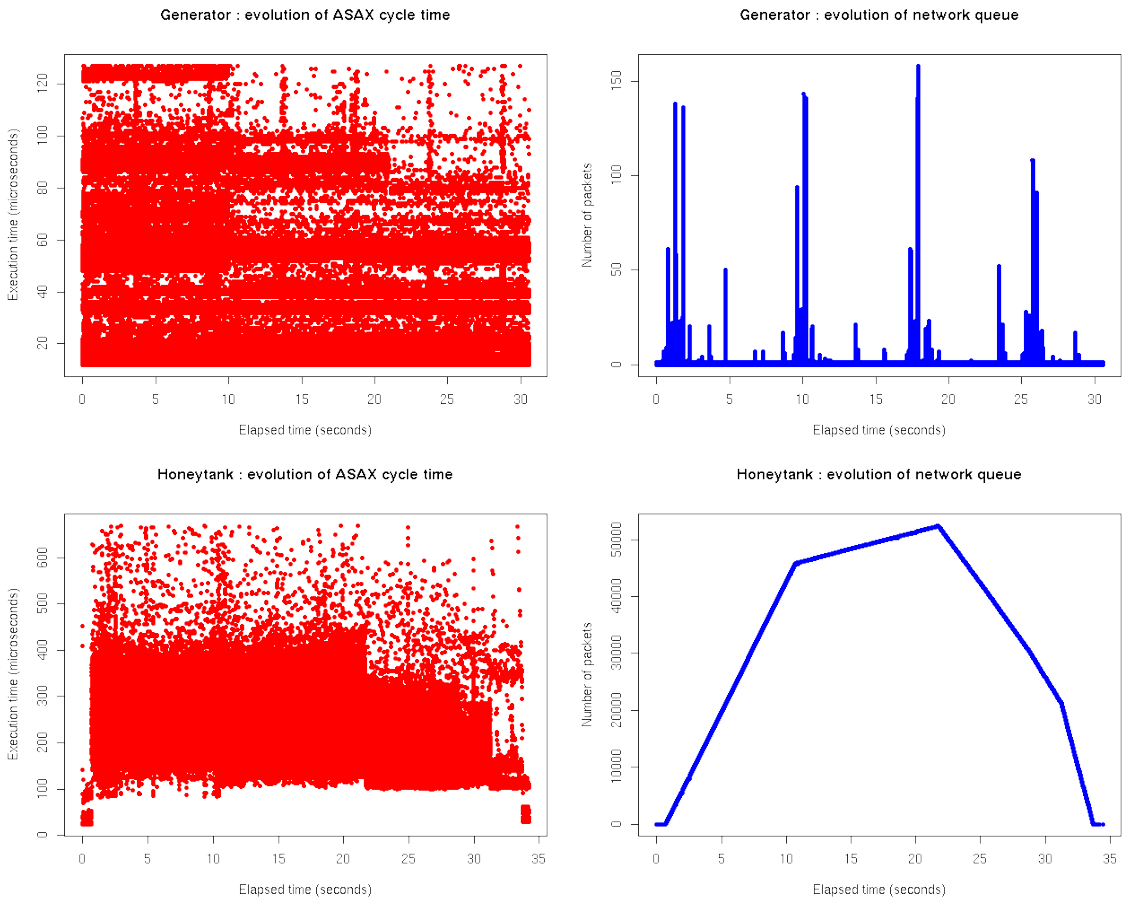

Figure 10.13: Unbounded generator Versus Unbounded stateful honeytank - Generation interval: 250 μsec

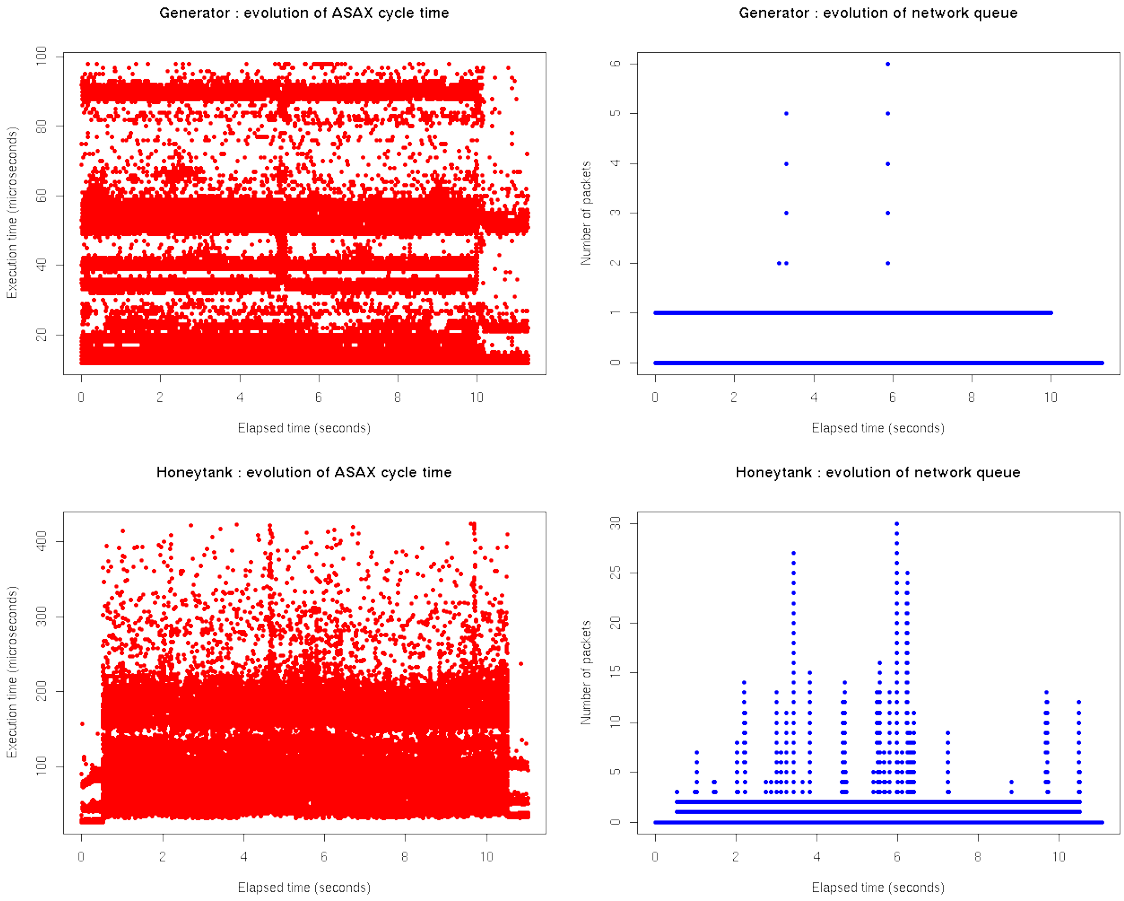

Figure 10.14: Unbounded generator Versus bounded stateful honeytank - Generation interval: 1000 μsec

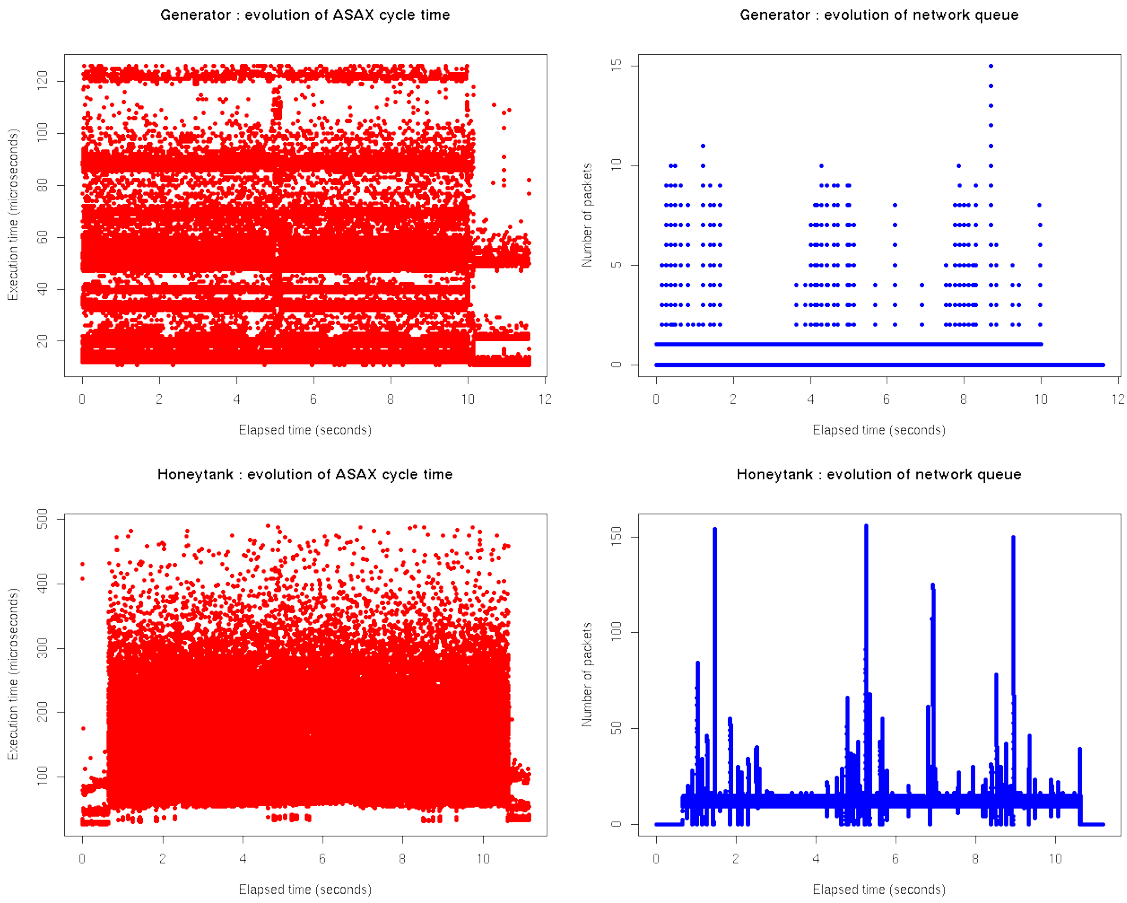

Figure 10.15: Unbounded generator Versus bounded stateful honeytank - Generation interval: 250 μsec

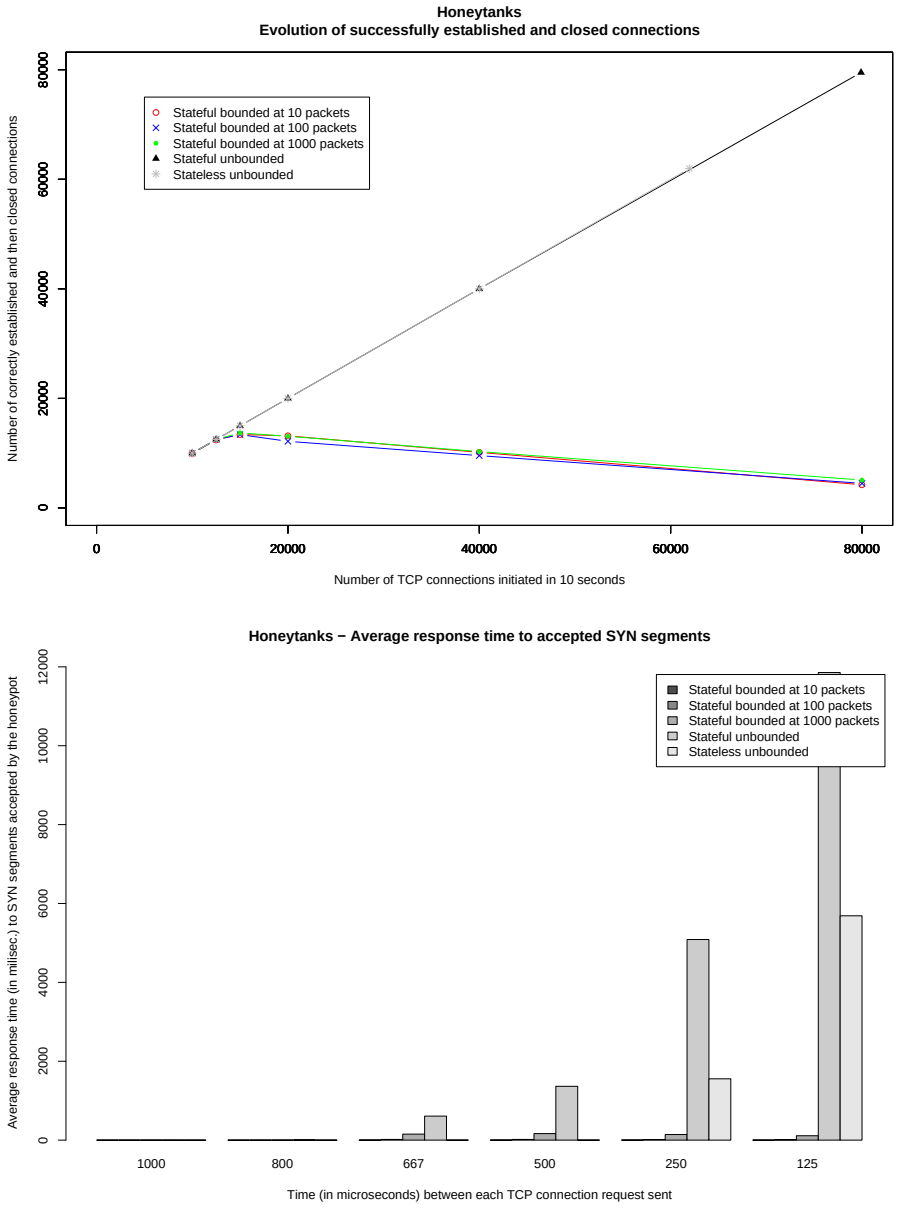

Figure 10.16: Efficiency of various honeytanks

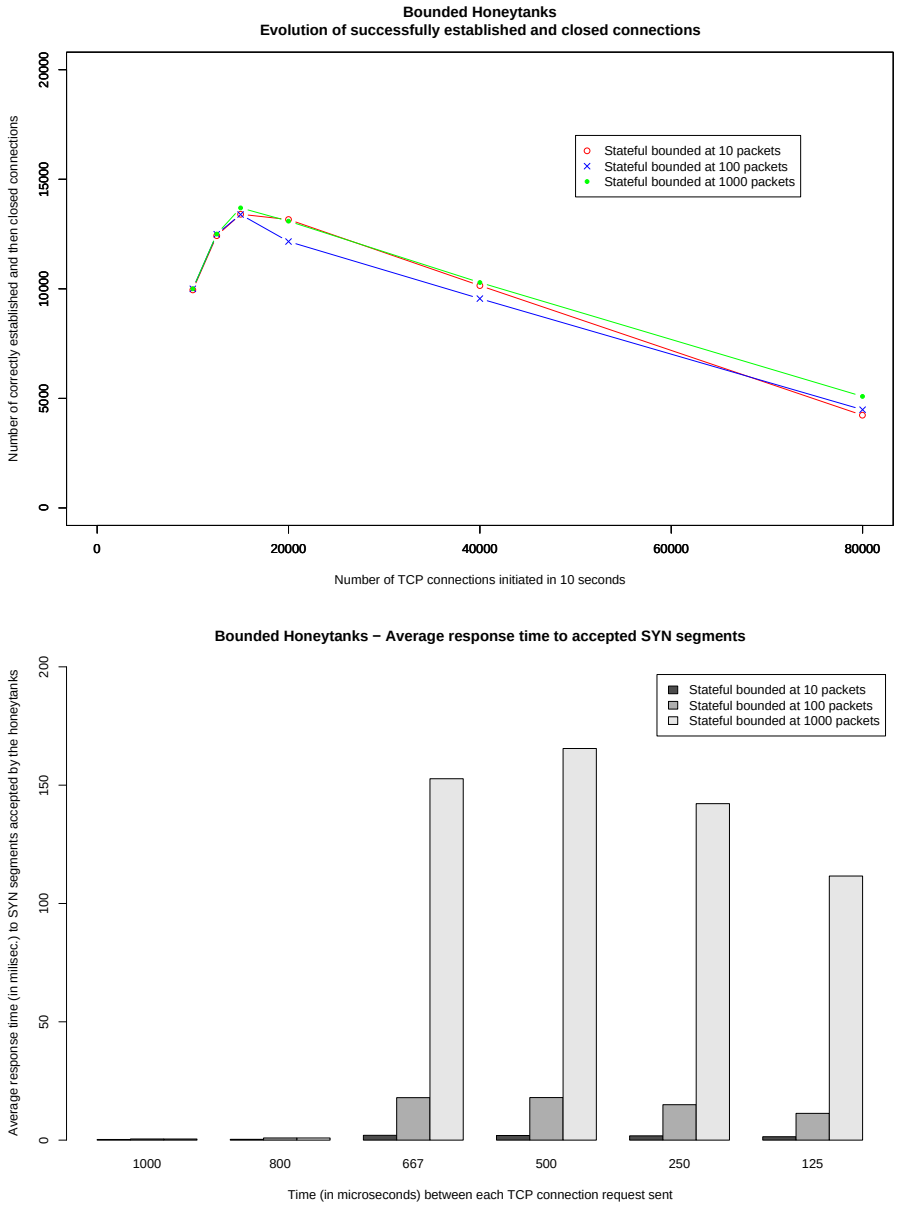

Figure 10.17: Efficiency of bounded stateful honeytanks

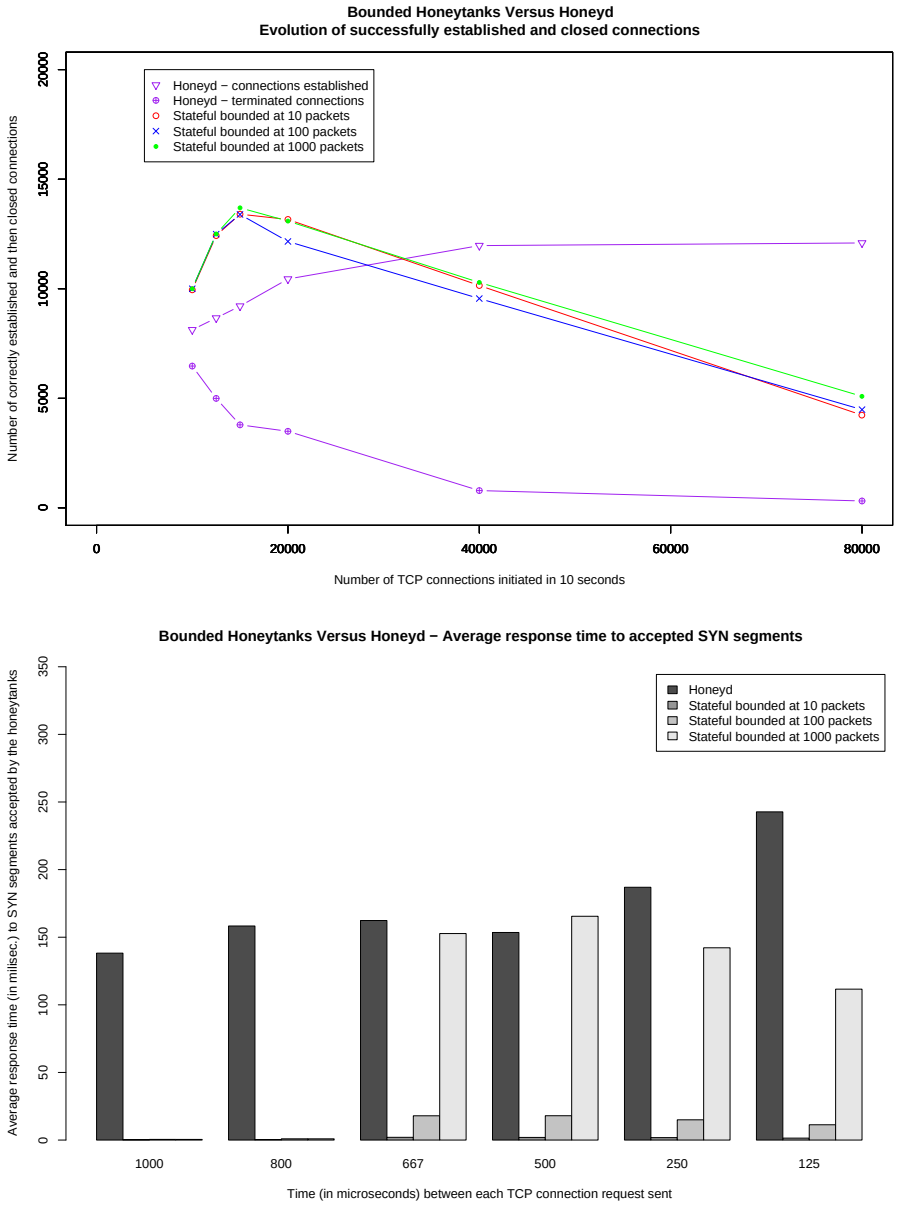

Figure 10.18: Efficiency of bounded stateful honeytanks compared with Honeyd

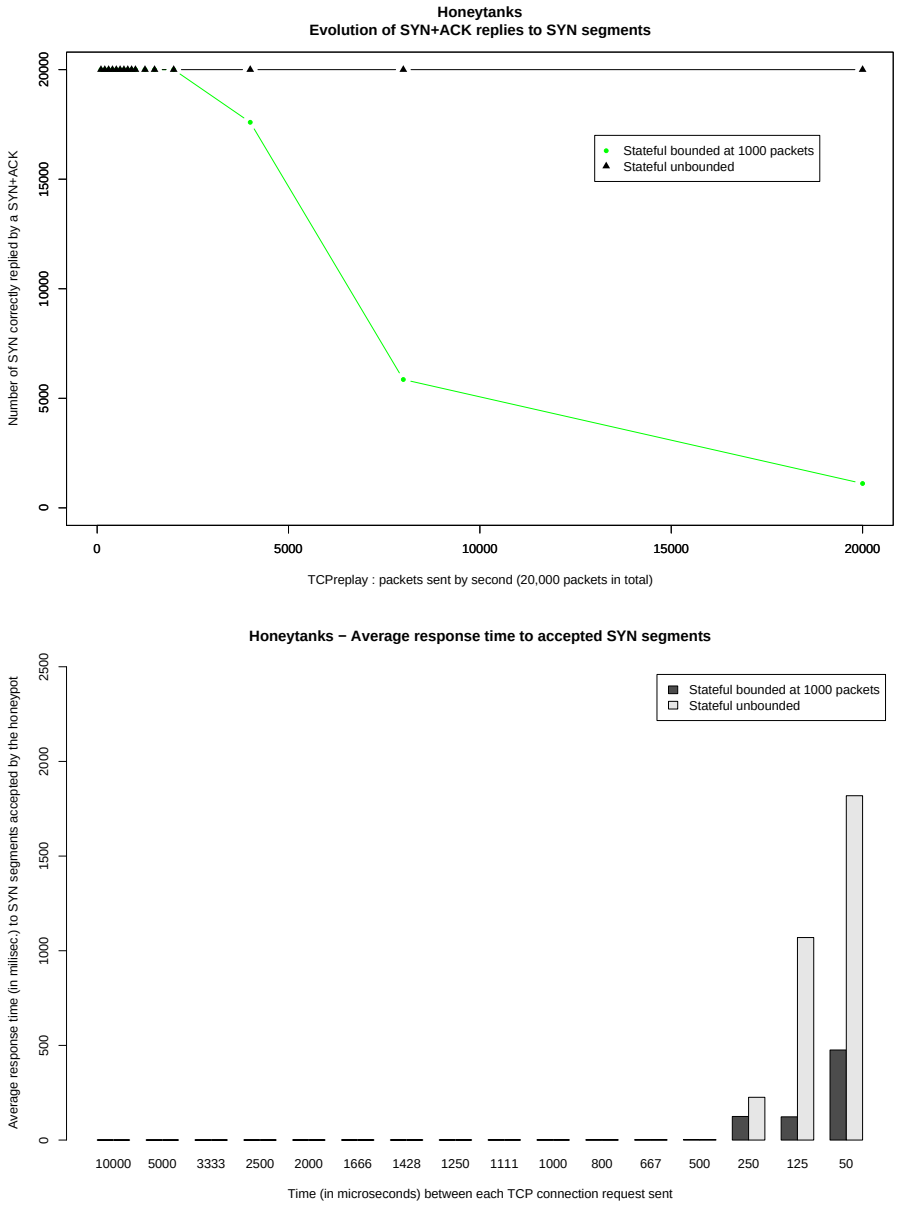

Figure 10.19: TCPReplay stress-test: efficiency of an unbounded an bounded stateful honeytank

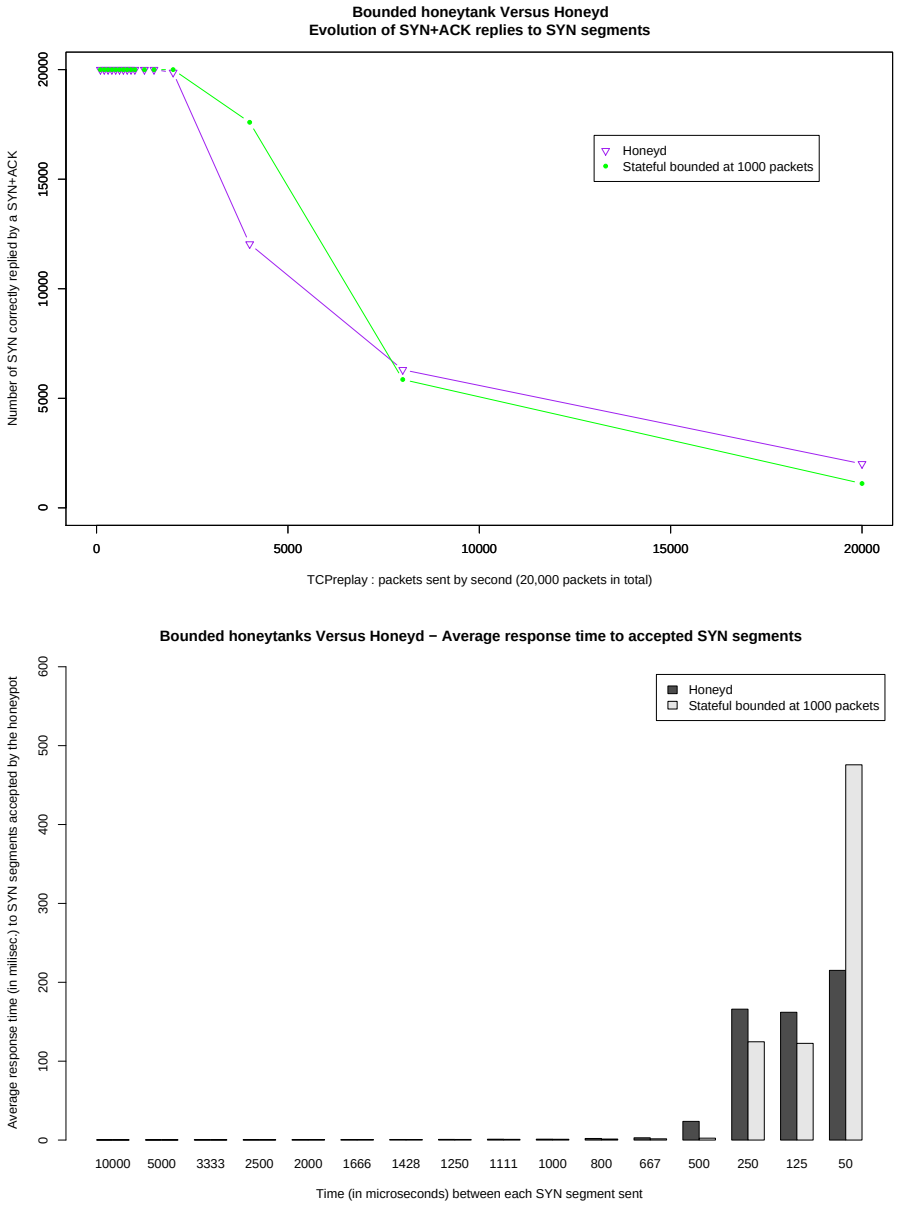

Figure 10.20: TCPReplay stress-test: efficiency of a bounded stateful honeytanks compared with Honeyd

Chapter 11

Qualitative analysis of collected traffic

> A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.
>
> Frank Herbert
> *Dune*

In this chapter, we present a qualitative analysis of the traffic collected by honeytanks. For several days, we have deployed honeytanks on an unused campus subnet. We performed two complementary experiments.

In the first experiment, we have executed alternatively a stateless and stateful honeytank to simulate the presence of hosts behind the unused 4.096 IP addresses of the network. Both honeytanks used stateless service simulators. We compare the traffic received by both honeytanks and we evaluate their respective performance. Moreover, we try to evaluate if the higher level of the TCP/IP simulation offered by the stateful honeytank allows it to collect more interesting traffic than the stateless honeytank. Analyses show that the stateful honeytank is able to cope with the incoming traffic. Even if it has to maintain much more state, its performance is equivalent to the stateless honeytank. Moreover, its additional abilities such as the retransmission of `SYN+ACK` segments proved to be useful when faced against "real-world" traffic.

In the second experiment, we used three honeytanks and deployed them on the same unused network than the one used for the first experiment. We translated some stateful vulnerability modules from the Nepenthes honeypot into RUSSEL and integrated them into a stateful honeytank. We configured a second stateful honeytank using an echo service in place of the translated Nepenthes modules. The third honeytank is a stateless honeytank. This

second experiment is complementary to the first one as we ran those three honeytanks in parallel and deployed stateful service simulators.

We compare the traffic they received and we evaluate if the higher level of realism of the honeytank using stateful service simulators impacts the quality of the collected traffic or the attractiveness of the honeytanks. Analyses show that attackers went further in their attacks when they were faced with stateful service simulators instead of an echo replier. However, simulated hosts running those stateful services did not attract more sources of traffic on those ports than the hosts running an echo replier on the same ports.

In Section 11.1, we present the configuration used for the first experiment. Section 11.2 presents a summary of the traces collected and in Section 11.3 we present the results of their analysis. Section 11.4 presents the conclusions of the first experiment.

In Section 11.5 we present the configuration of the three honeytanks used for the second experiment. Section 11.6 shows a global analysis of the collected traffic whereas Section 11.7 focuses on the impact of the stateful service simulators on the TCP traffic received by the three honeytanks. Section 11.8 presents the conclusion of this second experiment. Finally, Section 11.9 presents the global conclusion of this chapter.

11.1 Configuration used for the first experiment

For this experiment, we had access to an unused /20 subnet[1] of a campus network. The following configuration is used. A router is configured to forward all packets sent to IP addresses of this subnet to a single computer. This computer runs `TCPDump` to capture the exchanged traffic into files for later analysis. We first let `TCPDump` collect the received traffic without replying to it. Second, we use the stateless honeytank to reply to the incoming traffic. Third, the stateful honeytank is used. Those three steps are repeated several times.

The stateful honeytank is used in dynamic configuration, i.e., hosts and sessions are created on demand. Both honeytanks are used in their bounded configuration with a threshold set first at 10 packets. We later increased this limit to 2,000 packets for the stateful honeytank. In this configuration, when the network packet queue of the format adapter holds more packets than the threshold value, the stateless honeytank ignores incoming `SYN` segments and the stateful honeytank does not create new rule instances simulating hosts or sessions. Both honeytanks have the TCP ports 25, 80 and 111 open with an application-level simulator for SMTP, HTTP and RPCInfo bound to these ports.

[1] 4096 IP addresses

11.2 Presentation of the collected traces

We collected fourteen traces representing about 300 hours of traffic and over 6 millions incoming packets. Table 11.1 presents those traces. The *Honeytank* column shows which honeytank was used to reply to the incoming traffic. The value *TCPDump* means that the traffic was just collected without being replied. The remainder columns shows respectively the date and time at which the experiment started, its duration and the total number of received IP packets.

No	Honeytank	Start time and date	Duration	Received IP packets
1	TCPDump	10:11:25 01/08/2007	12:06:10	219,658
2	Stateless	23:31:59 01/08/2007	14:13:00	273,417
3	Stateful	13:50:47 02/08/2007	12:04:39	339,690
4	TCPDump	01:57:21 03/08/2007	12:08:42	143,581
5	Stateless	14:06:38 03/08/2007	22:03:51	352,184
6	Stateful	12:18:27 04/08/2007	75:10:36	1,405,884
7	Stateful	23:58:10 19/08/2007	11:21:39	164,240
8	Stateless	11:21:15 20/08/2007	13:19:56	361,251
9	TCPDump	00:41:43 21/08/2007	13:59:32	189,569
10	Stateful	14:42:14 21/08/2007	09:48:25	219,257
11	Stateless	00:31:36 22/08/2007	24:04:56	424,007
12	TCPDump	00:37:21 23/08/2007	24:07:53	689,851
13	Stateful	00:46:53 24/08/2007	25:20:49	717,673
14	Stateless	02:08:33 25/08/2007	24:08:28	573,092

Table 11.1: Presentation of the collected traffic

11.3 Analysis of the collected traces

11.3.1 Searching for NMap OS fingerprinting scans

We searched in the collected traces for clues indicating that an attacker used NMap to perform an OS fingerprinting scans towards some of our emulated hosts. We wrote a RUSSEL program which searches in all the traces for packets corresponding to a specific test made by NMap: packets with the TCP flags SYN, FIN, PSH and URG set. Those kinds of packets should never appear in a valid TCP session and are characteristic of a fingerprinting scan.

If such packets were present, we could try to determine if attackers performing fingerprinting scans use the result of such scans to later attack specific hosts by exploiting vulnerabilities only found on the operating systems reported by the scan. However, no such packets were found in the collecting

trace, indicating that no NMap fingerprinting scans were performed against our network during this experiment.

11.3.2 Statistics over collected packets

We wrote a RUSSEL program that computes various statistics about the received packets. This allows us to have a general overview of the received traffic.

Traffic collected without honeytanks

Table 11.2 shows a summary of the traffic collected without a honeytank, i.e., by simply listening to the traffic sent to unused IP addresses. Although the monitored IP are unused and are not advertised, they still receive a substantial amount of `SYN` segments and UDP messages.

Trace number	1	4	9	12
Duration				
	12:06:10	12:08:42	13:59:32	24:07:53
IP				
Total	219.658	143.581	189.569	689.851
ICMP				
Total	91,277	101,391	107,205	228,532
ECHO Req.	89,070	99,604	105,382	224,608
ECHO Reply	0	0	0	0
Unreachable	2,065	1,647	1,692	3,664
UDP				
Total	2,901	4,373	9,208	5,758
Total payload	1,830,991	3,154,388	5,130,466	4,045,589
TCP				
Total	125,480	37,817	73,156	455,561
With payload	5	6	24	3,846
Total payload	35	21	91	76,615
SYN	111,306	27,836	58,532	422,758
SYN+ACK	3,613	4,087	9,469	21,564
FIN	7	18	7	25
FIN+ACK	0	0	0	0
RST	7	18	7	25
RST+ACK	0	0	0	0
ACK	0	0	0	12
PSH	0	0	0	0
URG	0	0	0	0

Table 11.2: Received traffic without honeytanks

When aggregating those traces and looking at the TCP port destination distribution, we found that 333,207 `SYN` segments are destined to the port 5168. In August 22, 2007 a buffer overflow was announced[2] for the Trend Micro ServerProtect software, which listens on this TCP port. As those traces have been collected during the month of August 2007, we believe that we collected attempts to exploit this fresh vulnerability. Some other ports received a substantial amount of `SYN` segments. The TCP port 25 received 31961 `SYN` of them. This port is used by mail servers. Those segments are probably from spammers trying to find open SMTP relay in order to send some SPAM messages. The port 80 used by web server received 24715 segments. The port 22 received 16718 `SYN` segments. This port is used by the SSH program. The port 21 used by FTP received 12222 connections requests.

Several UDP messages are destined towards Windows Messenger. This tool is used to send short messages between hosts that are displayed by a pop-up screen. The packets contain a message that invites the user to download a Windows Registry Cleaner on a given website. Chances are that this program is in fact a Trojan Horse.

The presence of `SYN+ACK`, `FIN` and `RST` segments typically means that some of our IP addresses have been spoofed to initiate connections. Those segments are the replies of the contacted hosts. Moreover, although there was no honeytank to reply to the incoming traffic, some TCP segments contains data. This is unusual as a connection must be established before sending any data. Upon inspecting, we found that some `RST` segments contained a payload made of the string `cko` or `rctcpo`. In the trace number 1, one `RST` segment has as payload the string `Go away, we're not home`. More surprisingly, the trace number 12 contains several `SYN+ACK` segment with a payload. Those kind of segments are not supposed to contain data. This payload appears to be binary data but we were not able to find its meaning.

This traffic collected by merely listening to unused IP addresses confirms that the Internet carries a substantial amount of malicious traffic and that monitoring unused IP addresses is an interesting way of collecting such malicious traffic.

Traffic collected with a stateless honeytank

Table 11.3 shows a summary of the traffic collected with a stateless honeytank. The number of `ACK` segments received as well as the amount of payload in TCP segments confirms that traffic has been exchanged between attackers and the simulated hosts.

Upon analysis of the traces collected by the stateless honeytank, we again found lots of (i.e., more than 110,000) connection attempts to the port 5168.

[2] `http://secunia.com/advisories/26523/`

Trace number	2	5	8	11	14
Duration					
	14:13:00	22:03:51	13:19:56	24:04:56	24:08:28
IP					
Total	273,417	352,184	361,251	424,007	573,092
ICMP					
Total	125,047	215,387	142,901	222,987	143,282
ECHO Req.	123,042	211,877	140,863	215,142	139,088
ECHO Reply	0	0	0	5	1
Unreachable	1,790	2,971	1,954	3,548	3,454
UDP					
Total	4,390	7,227	5,607	12,653	18,603
Total payload	3,010,971	5,079,691	2,456,424	7,536,421	3,377,132
TCP					
Total	143,980	129,570	212,743	188,367	411,207
With payload	26,783	482	4,316	2,548	9,295
Total payload	2,315,475	121,171	635,029	280,209	476,756
SYN	72,599	80,123	165,156	117,320	258,290
SYN+ACK	2,580	5,748	11,556	19,595	62,639
FIN	0	0	0	0	0
FIN+ACK	6,022	12,515	4,171	12,896	24,693
RST	0	0	0	0	0
RST+ACK	6,022	12,515	4,171	12,896	24,693
ACK	56,321	25,520	21,944	22,107	58,163
PSH	26,606	349	4,058	643	8,497
URG	0	0	0	0	0

Table 11.3: Received traffic with a stateless honeytank

We also found more than 90,000 `SYN` segments sent to the port 5900 used by VNC, a graphical desktop sharing system. More interestingly, the traces show thousands of connexions attempts to the ports 80 and 25. These ports are used respectively by web and mail servers. As the stateless honeytank has an application-level simulator for those protocols, we inspected these traces manually to analyze the traffic exchanged.

We found several regular HTTP GET requests, probably from robot crawlers in search of new web servers. We also found more unusual requests like POST commands with a Windows DLL as parameter. As for the port 25, we found that many mails have been received by our fake SMTP servers. Several mails are SPAM messages promoting a drug used to treat male erectile dysfunction. A lot of other messages contain only a subject field with the IP address of the fake SMTP server. We believe that the spammer is sending those mails to itself. If it receives those mails, he knows that the

SMTP server he found is functional and can retrieve its IP address stored in the subject field of the message.

The collected traffic confirms that the approach used by the stateless honeytank as well as the HTTP and SMTP simulators are "good enough" to collect malicious traffic.

Traffic collected with a stateful honeytank

Table 11.4 shows a summary of the traffic collected with a stateless honeytank. The number of `ACK` segments received as well as the amount of payload in TCP segments confirms that traffic has been exchanged between attackers and the simulated hosts. The analysis of the traces shows again lots of TCP connections requests toward the ports 5168, 80 and 25. A manual inspection of the trace shows that the traffic collected seems similar to the traffic collected by the stateless honeytank.

Trace number	3	6	7	10	13
Duration					
	12:04:39	75:10:36	11:21:39	09:48:25	25:20:49
IP					
Total	339,690	1,405,884	164,240	219,257	717,673
ICMP					
Total	144,885	717,692	81,307	95,422	172,532
ECHO Req.	143,043	706,509	79,526	93,520	168,067
ECHO Reply	0	375	0	0	0
Unreachable	1,636	9,658	1,711	1,542	4,120
UDP					
Total	4,873	22,432	6,840	2,317	7,234
Total payload	3,426,922	13,979,350	2,961,288	385,344	2,991,962
TCP					
Total	189,932	665,760	76,093	121,518	537,907
With payload	5,094	31,947	254	3,401	11,032
Total payload	262,220	3,645,580	33,051	1,898,548	2,823,009
SYN	157,346	413,069	46,594	86,570	342,645
SYN+ACK	1,634	39,360	18,398	13,845	16,861
FIN	0	0	0	0	0
FIN+ACK	2,296	38,229	227	2,194	45,682
RST	0	0	0	0	0
RST+ACK	2,296	38,229	227	2,194	45,682
ACK	22,008	137,088	936	14,290	122,929
PSH	5,083	25,931	236	2,191	4,695
URG	0	0	0	0	0

Table 11.4: Received traffic with a stateful honeytank

11.3.3 Snort analysis of the received traffic

To have a better view of the attacks contained in the collected traffic, we analyze them with Snort, a network intrusion detection system. We use Snort 2.3.3 (Build 14) with the default detection rules as well as the latest set of *bleeding edges*[3] rules. Table 11.5 shows the Snort analysis of the traffic collected without a honeytank. Since the incoming traffic is not replied to, no TCP sessions are opened and the attackers have not the opportunity to send their malicious payload. Snort detects mainly ICMP-based scans.

Table 11.6 and Table 11.7 show the attacks found by Snort in the trace collected respectively by a stateless and a stateful honeytank. Those honeytanks reply to the incoming traffic and simulate the HTTP, SMTP and RPCInfo protocols. Snort finds many attacks based on the HTTP protocol. The stateful honeytank collected attacks which are not present in the trace collected by the stateless honeytank. For example, the stateful honeytank attracted WebDAV attacks, caught several CodeRed worms trying to infect IIS web servers and was the target of UDP and TCP portscans. Globally, the number of occurrences of attacks targeting HTTP servers is higher in the traces collected by the stateful honeytank.

Occurrences	Attack found
3825	TCP Data Offset less than 5
1811	ICMP PING CyberKit 2.2 Windows
1423	ICMP Dest. Unreach. Communication Administr. Prohibited
230	ICMP PING NMAP
188	BLEEDING-EDGE WORM Allaple ICMP Sweep Ping Inbound
103	ICMP redirect host
57	ICMP Dest. Unreach. Dest. Host Administr. Prohibited
31	ICMP Dest. Unreach. Dest. Network Administr. Prohibited
14	Short UDP packet, length field greater than payload length
12	SCAN FIN

Table 11.5: Snort alerts found in the traffic collected without a honeytank

11.3.4 Evaluation of the stateful honeytank

We finally examine the behavior of the stateful honeytank. First, we observe if the `SYN+ACK` retransmission feature is used when the stateful honeytank is faced with real traffic. Then, we observe if it had some difficulties to handle this traffic.

`SYN+ACK` retransmission and opened sessions To evaluate if the `SYN+ACK` retransmission feature implemented in the stateful honeytank is useful when it is used in a real environment, we wrote a RUSSEL program to track the

[3] http://www.bleedingsnort.com/

Occurrences	Attack found
4428	ICMP PING NMAP
3841	ICMP redirect host
2740	ICMP PING CyberKit 2.2 Windows
1749	ICMP Dest. Unreach. Communication Administr. Prohibited
1549	TCP Data Offset less than 5
300	BLEEDING-EDGE WORM Allaple ICMP Sweep Reply Outbound
300	BLEEDING-EDGE WORM Allaple ICMP Sweep Ping Inbound
155	SHELLCODE x86 NOOP
63	ICMP Dest. Unreach. Host is Administr. Prohibited
55	(http_inspect) DOUBLE DECODING ATTACK
48	(http_inspect) BARE BYTE UNICODE ENCODING
19	(http_inspect) WEBROOT DIRECTORY TRAVERSAL
11	(portscan) TCP Portsweep
9	ICMP Dest. Unreach. Network is Administr. Prohibited
9	BLEEDING-EDGE WEB Proxy GET Request
4	WEB-IIS cmd.exe access
2	ICMP Large ICMP Packet
2	WEB-MISC http directory traversal
1	ICMP webtrends scanner
1	WEB-IIS httpodbc.dll access - nimda
1	(portscan) TCP Decoy Portscan
1	(http_inspect) OVERSIZE REQUEST-URI DIRECTORY
1	WEB-MISC Chunked-Encoding transfer attempt
1	WEB-ATTACKS tftp command attempt
1	WEB-FRONTPAGE /_vti_bin/ access
1	WEB-FRONTPAGE rad fp30reg.dll access

Table 11.6: Snort alerts found in the traffic collected by the stateless honeytank

`SYN+ACK` segments sent by a honeytank, count the number of time it is retransmitted and count the number of session opened.

The hosts emulated by the stateful honeytank have the TCP ports 25, 80 and 111 opened and a simulator of respectively the SMTP, HTTP and RPCInfo protocol is bound to those ports. The stateful honeytank received globally 101,835 `SYN` segments to these opened ports. It replied with 177,978 `SYN+ACK` segments. Over the 101,835 connections requests received on opened ports, the stateful honeytank handled 97,631 of them, the other were discarded because the network queue exceeded its threshold. In total, 97,367 sessions completed the 3-way handshake.

Since it replied more `SYN+ACK` segments than it received `SYN` segments, we can conclude that the `SYN+ACK` retransmission feature was indeed used during this experiment. More precisely, we computed that, amongst `SYN+ACK` segments that finally lead to a TCP session opening, 41,556 of them were not retransmitted. However, 38,481 of them were retransmitted once, 12,966 of

Occurrences	Attack found
3950	ICMP superscan echo
3790	ICMP PING CyberKit 2.2 Windows
2805	ICMP Dest. Unreach. Communication Administr. Prohibited
758	WEB-MISC Chunked-Encoding transfer attempt
758	WEB-FRONTPAGE /_vti_bin/ access
758	WEB-FRONTPAGE rad fp30reg.dll access
458	ICMP PING NMAP
446	TCP Data Offset is less than 5
433	BLEEDING-EDGE WORM Allaple ICMP Sweep Reply Outbound
433	BLEEDING-EDGE WORM Allaple ICMP Sweep Ping Inbound
294	BLEEDING-EDGE WEB Proxy GET Request
245	(portscan) TCP Portscan
201	ICMP redirect host
178	(http_inspect) BARE BYTE UNICODE ENCODING
100	WEB-IIS httpodbc.dll access - nimda
99	WEB-IIS cmd.exe access
96	(http_inspect) OVERSIZE REQUEST-URI DIRECTORY
82	SHELLCODE x86 NOOP
75	WEB-ATTACKS tftp command attempt
66	ICMP Dest. Unreach. Host is Administr. Prohibited
61	(portscan) UDP Portscan
44	WEB-MISC http directory traversal
42	BLEEDING-EDGE WEB WebDAV search overflow
42	WEB-MISC WebDAV search access
37	(http_inspect) DOUBLE DECODING ATTACK
26	(portscan) TCP Portsweep
8	BLEEDING-EDGE WEB-MISC Poison Null Byte
8	(http_inspect) WEBROOT DIRECTORY TRAVERSAL
7	WEB-IIS CodeRed v2 root.exe access
4	ICMP Dest. Unreach. Network is Administr. Prohibited
2	WEB-MISC robots.txt access
1	ICMP webtrends scanner
1	(portscan) UDP Portsweep

Table 11.7: Snort alerts found in the traffic collected by the stateful honeytank

them were retransmitted twice and 3,195 of them were retransmitted three times.

We can see that the situation where a `SYN+ACK` segment must be retransmitted occurs rather frequently. Thus, this feature is interesting as it can increase the number of TCP sessions successfully opened.

Performance evaluation The stateful honeytank is used in bounded configuration, i.e., it does not create new rule instances simulating hosts or sessions when the network packet queue of the format adapter holds more

packets than a given threshold value. We instrumented the stateful honeytank to print the number of time the threshold is exceeded as well as the duration of each exceedance. This information was collected for the traces number 7, 10 and 13, which used a threshold value of 2,000. Unfortunately, due to a configuration error, the rule computing the duration of each exceedance was still configured for a threshold value of 10.

Analysis of the trace number 7 shows that during more than 11 hours of traffic, the queue of the stateful honeytank went 346 times over the threshold value of 10. All but two exceedances lasted less than 35,000 microseconds. One lasted nearly 2 seconds the other lasted nearly 25 seconds. However, the maximum number of packets in the queue was 1,932.

Analysis of the trace number 10 shows that during more than 9 hours of traffic, the queue of the stateful honeytank went 659 times over the threshold value of 10. All exceedances lasted less than 40,000 microseconds. The maximum number of packets in the queue was 160.

Analysis of the trace number 13 shows that during more than 25 hours of traffic, the queue of the stateful honeytank went 1681 times over the threshold value of 10. Most exceedances lasted less than 100,000 microseconds. However, 39 exceedances lasted between 500,000 and 750,000 microseconds and one lasted about 25 seconds. The maximum number of packets in the queue was 2,795. In total, the stateful honeytank refused to create 18,902 new rules simulating a host because its network queue was over the threshold value of 2,000 packets.

Globally, the stateful honeytank was able to handle the load of the incoming traffic and its network queue went infrequently over 10 packets. However, on one occasion, it had to refuse to create new hosts as its queue went over the threshold value of 2,000. This shows that, even if the stateful honeytank has to maintain lots of state to achieve a realistic simulation, it is still able to cope with the traffic received in a "real-world" environment.

11.4 Conclusion of the first experiment

The qualitative analysis of the traffic collected by honeytanks clearly shows that the approach taken is viable. It confirms that using a honeytank to reply to the incoming traffic allows one to collect more malicious traffic with more information than merely listening to the incoming traffic without replying to it. Moreover, the stateful honeytank proved to be as efficient as the stateless honeytank. It was able to handle most of the received traffic while providing a more realistic simulation, thus proving the usefulness of the optimized version of ASAX. The `SYN+ACK` retransmission implemented in the stateful honeytank is useful as it allows to open sessions that might have otherwise been left unopened.

Both honeytanks were used in alternance during several days. On aver-

age, they received about the same amount of traffic. The stateful honeytank received 2,846,744 over 133 hours, 46 minutes and 8 seconds of traffic. That is 5,9 packets per second on average. On its hand, the stateless honeytank received 1,983,951 packets over 97 hours, 50 minutes and 11 seconds of traffic. That is 5,6 packets per second on average. However, Snort analysis of the collected traces shows that the stateful honeytank collected more different types of attacks than the stateless honeytank.

11.5 Configuration used for the second experiment

For this experiment, we reused the /20 subnet (i.e. 4096 IP addresses) used for the first experiment. This time, we ran 3 different honeytanks in parallel. Each honeytank replies to one third of the address space of the network.

The first honeytank, **H1**, is our stateful honeytank. The stateless SMTP, HTTP and RPCinfo repliers are bound to the ports 25, 80 and 111 respectively. Using the technique presented in Chapter 8, we translated several stateful vulnerability modules from Nepenthes and integrated them into **H1**. Those modules are the Microsoft-DS (bound to the port 445), the DCOM module (bound to the ports 135 and 1025) and the Real VNC module (bound to the port 5900). The source code of those modules are given in Appendix D. All other ports are closed. All IP addresses handled by `H1` reply to `ICMP echo` requests. This honeytank is used in its bounded configuration with a limit on the network queue set at 50 packets. It means that the honeytank will not accept new connection requests when there are more than 50 packets awaiting in its buffer. However, packets belonging to already established connections are processed as usual.

The second honeytank, **H2**, has the same configuration as **H1** but does not use the modules translated from Nepenthes. Instead, it uses on the same ports (135, 445, 1025 and 4900), an "echo reply" service. Once a session has been established on one of those ports, it sends an appropriate "welcome banner" if it is required by the protocol. Then, each time a data segment is received on those ports, the echo replier sends back a data segment containing the same payload. We later verify if this difference in the way the traffic to these ports is handled impacts the traffic sent by attackers.

Finally, the third honeytank, **H3**, is a stateless honeytank. The stateless SMTP, HTTP and RPCinfo repliers are respectively bound to the ports 25, 80 and 111. The ports 135, 445, 1025 and 4900 are opened but no replier at all is associated with them. It means that a connection request to one of those port will succeed but then, any data segment sent to these ports will be ignored. This is similar to the approach taken by the Labrea [26] honeypot. We check later on if this behavior has an impact on the attractiveness of the honeytank.

We ran this setup during 29 days from the 21 October 2007 to the 18

November 2007. All the traffic exchanged was captured by TCPDump and written in a distinct file for each day of the experiment. Unfortunately, the trace for the 13^{th} day is corrupted. It is thus ignored in the analysis further described.

Those traces contain 51,724,897 IP packets and represent about 4GBytes of data. To analyze those traces, we developed several RUSSEL modules that, amongst other things,

1. count the global number of packets received and sent by the three honeytanks,
2. count the number of packets exchanged for each opened port of each honeytank,
3. count the number of unique sources that contacted the honeytanks regardless of the destination port,
4. count the number of unique sources that contacted the honeytanks for each open port,
5. count the number of segments containing a payload received in sessions established on each honeytank.

We used the programming methodology presented in Chapter 7 to write those rules. Their source code is given in Appendix E. Thanks to the modularity of ASAX, we executed those different modules in parallel. On our computer[4], the 4 GBytes of network traces were analyzed in about 9 minutes and 30 seconds.

11.6 Global analysis of the collected traffic

We first analyze the collected traces in a global way in order to have a better understanding of their contents.

Figure 11.1 shows the evolution of the number of received IP packets for each honeytank. At first sight, the number of received packets greatly varies from one honeytank to another on a daily basis. During the first 16 days, **H3** received generally much less packets than the other honeytanks. During this same period of time, **H1** and **H2** vary in the same manner but **H1** received more packets. However, at days 1, 4, 8 and 9, all honeytanks received approximately the same number of packets. This situation is also observed during days 17 to 25. So, for 12 of the 29 collected traces, all honeytanks received about the same number of packets. On day 11 and 12, we can observe a clear peak in the received traffic of **H2**. We investigate this later.

[4]Our computer has an Intel Core 2 CPU 4300 clocked at 1.8 GHz with 1GBytes of RAM

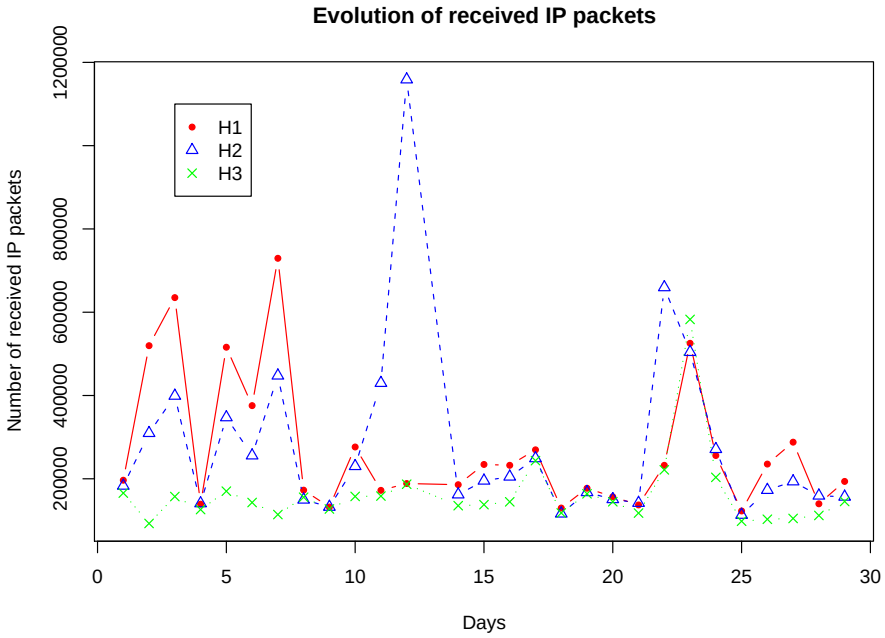

Figure 11.1: Evolution of received IP packets

We now extract the ICMP packets from the set of IP packets and we observe their evolution. Figure 11.2 shows the number of received ICMP packets for each honeytank. We can see that this figure is radically different from the previous one. Excepted for day 12 where **H1** received more ICMP packets, all honeytanks received about the same amount of ICMP packets. So, it seems that ICMP packets are sent by attackers regardless of the other simulated protocols.

Figure 11.3 shows the evolution of unique sources contacting the honeytanks. That is, for each day and for each honeytank, we show the number of different IP addresses which have sent a packet to a given honeytank. There is no clear pattern emerging from this figure. It seems that there is no obvious correlation between the number of unique sources that contacted a honeytank and the number of IP packets it received. Nonetheless, the stateful honeytanks (**H1** and **H2**) are globally contacted by more different sources than the stateless honeytank (**H3**).

We then searched in the collected traces for packets which are typical of a NMap OS fingerprinting scan. As explained in Chapter 6, NMap is able to identify the type of a TCP/IP stack by analyzing its responses to a set of very specific TCP segment. For example, one of this test consists of sending a TCP segment with the flags `SYN`, `FIN`, `PSH` and `URG`. However, we did not find any such specific segments in the collected trace.

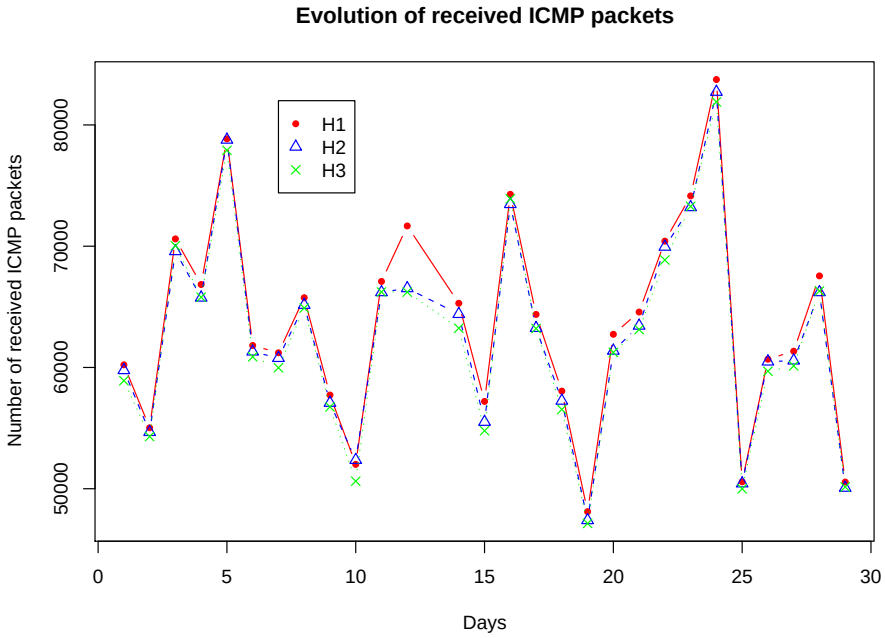

Figure 11.2: Evolution of received ICMP packets

11.7 Analysis of the TCP connections

The previous section gave a global view of the collected trace. However, it is not yet possible to explain the variations in the number of received packets. We now focus on the TCP sessions.

11.7.1 Analysis of received SYN segments

Figure 11.4 shows the evolution of the TCP `SYN` segments received by each honeytank. A `SYN` segment indicates that a source tries to initiate a TCP connection with an emulated host. We can observe that from day 1 to day 7, **H3**, the stateless honeytank receives less TCP connection initiation requests than the two other honeytanks. During the same period of time, **H1** and **H2** followed the same pattern but **H1** received more `SYN` segments than **H2**. Then, from day 8 to day 25, all three honeytanks globally received the same number of `SYN` segments with the exception of day 12 and 23. It is interesting to compare this figure with Figure 11.1 depicting the number of IP packets received as they show some similarities. Apart from the peak on day 12, the pattern is globally the same. Indeed, on day 12, **H2** has received a little more `SYN` segments than the other honeytanks but has received much more IP packets than the others.

We opened a limited set of ports on our deployed honeytanks. Thus, not

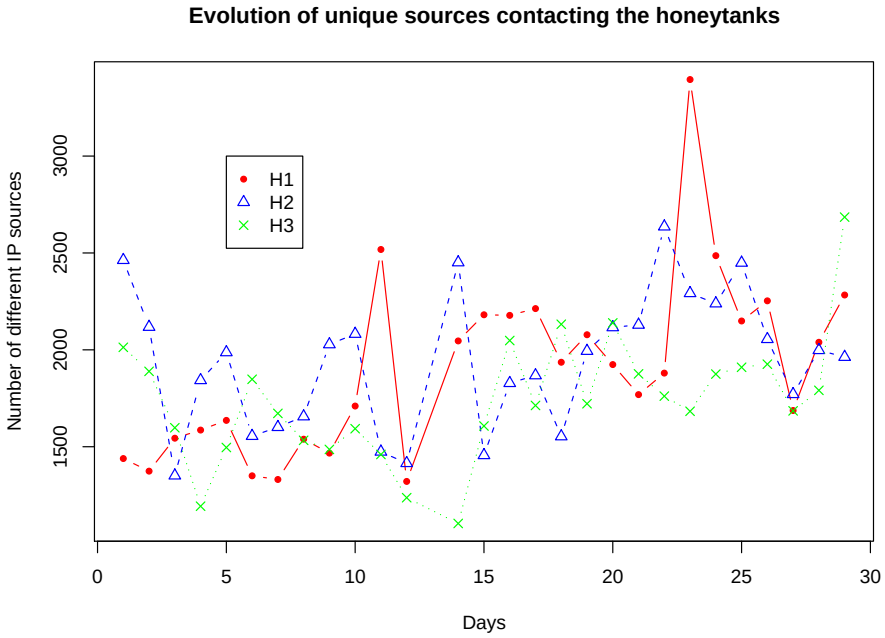

Figure 11.3: Evolution of unique sources contacting the honeytanks

all received SYN segments do lead to a connection opening as the honeytanks refuse to open a connection on a closed port. We now analyze the number of received SYN segments for each of the TCP ports opened on the honeytanks. No SYN segments for the TCP port 111 were received during this experiment. The **H1** honeytank received 7 SYN segments on the port 1025 during the whole experiment. The two other honeytanks did not receive SYN segment for this port. On day 17 and on day 23, all honeytanks received respectively about 38,000 and 2,500 SYN segments on the port 135. During the other days, no SYN segments were received for this port. Similarly, our three honeytanks received on day 11 and 27 respectively about 300 and 2600 SYN segments on the port 445. During the other days, no traffic was received for this port. The evolution of SYN segments received on ports 25, 80 and 5900 is more diversified. We now inspect traffic received on those ports.

Figure 11.5 shows the evolution of received SYN segments on the port 80. We can observe that, during the first seven days, **H1** and **H2** followed the same pattern but **H1** received more SYN segments to the port 80 than **H2**. The **H3** honeytank received much less connection requests than the others during this period of time. However, we can observe that from day 8 to day 25, the number of segments received is about the same on all honeytanks. If we compare this figure with Figure 11.4 which shows the global number of received SYN, we can see that they are quite similar. If we compare the

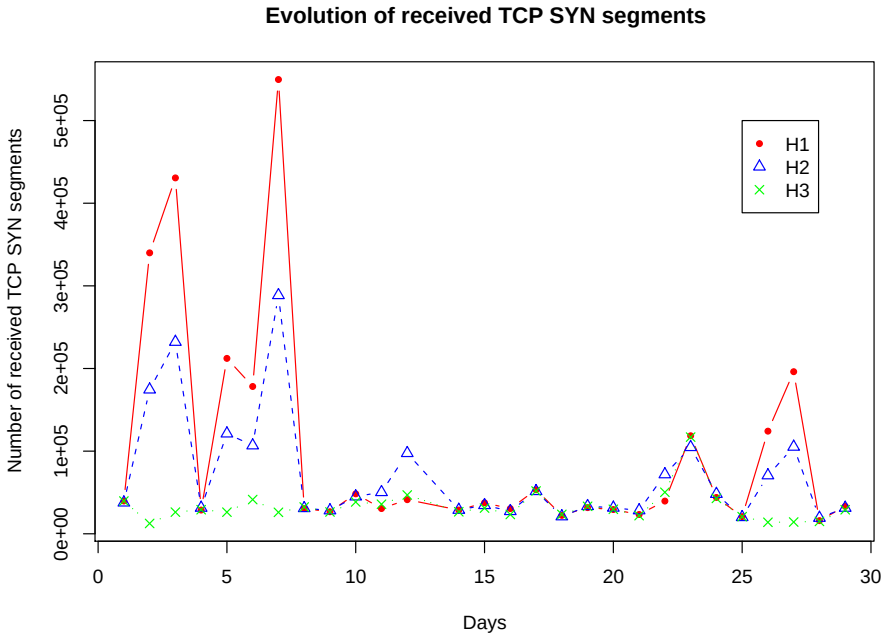

Figure 11.4: Evolution of received TCP SYN segments

number of `SYN` received on the port 80 with the number of `SYN` received regardless of the destination port during the first 8 days and during the last 9 days, we observe that most received `SYN` are in fact sent to the port 80.

Figure 11.6 shows the evolution of received `SYN` segments to port 25. We can see that, on days 11, 12 and 22, the stateful honeytank **H1** received much less `SYN` segments than the other honeytanks. However, on day 12, there is a clear peak of SMTP connection requests received by **H2**. This can explain the peak of received packets by **H2** on day 12 shown in Figure 11.1. The SMTP protocol is used to send e-mails, so the peak is probably due to a spammer. A manual inspection of the trace of day 12 confirmed this hypothesis. We found that lots of e-mail had been sent to the stateless SMTP server bound to the simulated hosts by **H2**.

Those mail are sent to a @mail2000.com.tw e-mail address, a Taiwanese free e-mail provider. The subject of those mails is the IP address of the simulated SMTP server contacted on **H2**. We believe that the spammer is sending those mails to an e-mail account he controls. If the mails are correctly sent to this address, it means that the SMTP server he used is valid and functional and he can re-use this server to send SPAM mails to other people. Figure 11.7 shows the number of unique IP addresses which sends `SYN` segments to the port 25. The number of different sources is very low. Only 3 sources are responsible for the peak of SMTP traffic observed

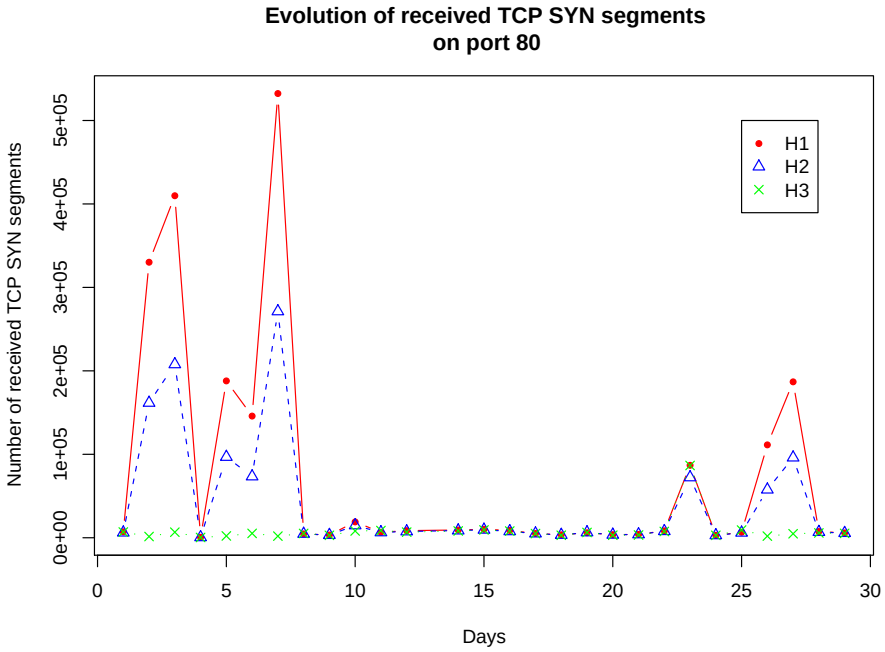

Figure 11.5: Evolution of received TCP SYN segments on the port 80

on **H2** during day 12.

11.7.2 Traffic received by the stateful VNC service simulator

Finally, we analyze the TCP traffic received on the port 5900. This port is associated to the VNC (Virtual Network Computing) protocol. We have explained earlier that the way the TCP connections on this port are handled greatly varies from one honeytank to another. The **H1** honeytank uses a vulnerability module translated from Nepenthes to process data sent on a connection to the port 5900. The **H2** honeytank echoes all received data on this port and **H3** accepts incoming connections to this port but ignores further received segments.

Thus, we first evaluate if one of the honeytanks receives more connections requests on the port 5900 than the others. Figure 11.8 shows the evolution of TCP connections requests received on this port. We can observe that the number of `SYN` segments received is about the same for all honeytanks. It means that despite such differences in the way the traffic to the port 5900 is handled, the three honeytanks receive about the same amount of connection requests. So it seems that attackers keep contacting hosts simulated by **H2** and **H3** even if they do not simulate correctly the VNC protocol.

Then, we evaluate if a honeytank received traffic on the port 5900 from

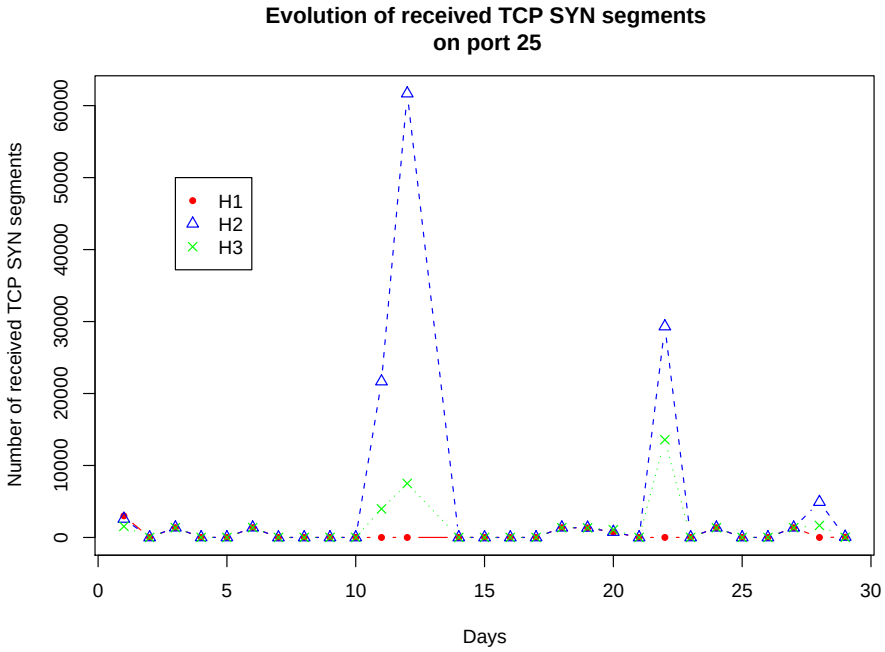

Figure 11.6: Evolution of received TCP SYN segments on the port 25

more different sources than the others. Since **H1** offers a better simulation of the VNC protocol, it might attract more different attackers. Figure 11.9 shows the evolution of unique sources contacting the honeytanks. Again, we can observe that about the same number of attackers established TCP connections on the port 5900 for the three deployed honeytanks.

Finally, we take a closer look to the data exchanged through the connections established on the port 5900. More specifically, for each TCP connection successfully established on this port, we count the number of received segments carrying a payload that were received by the honeytank for this connection. On the following figures, lines labeled with a 0 are connections that have been correctly established but for which the honeytank never received a TCP segment containing a payload. It means that the 3-way handshake was correctly performed but the attacker never sent a data segment. The lines labeled with 1 represents sessions in which one segment containing a payload was received and lines labeled with a 2 are connections that received two data segments.

Figure 11.10 shows this analysis for **H1**. To interpret this figure, we must understand how the VNC service simulation translated from Nepenthes works. Once a connection has been established on the port 5900, **H1** sends a "welcome banner" as it is required by the simulated protocol. In this case, the banner consists of the string "RFB 003.008", which is the version of

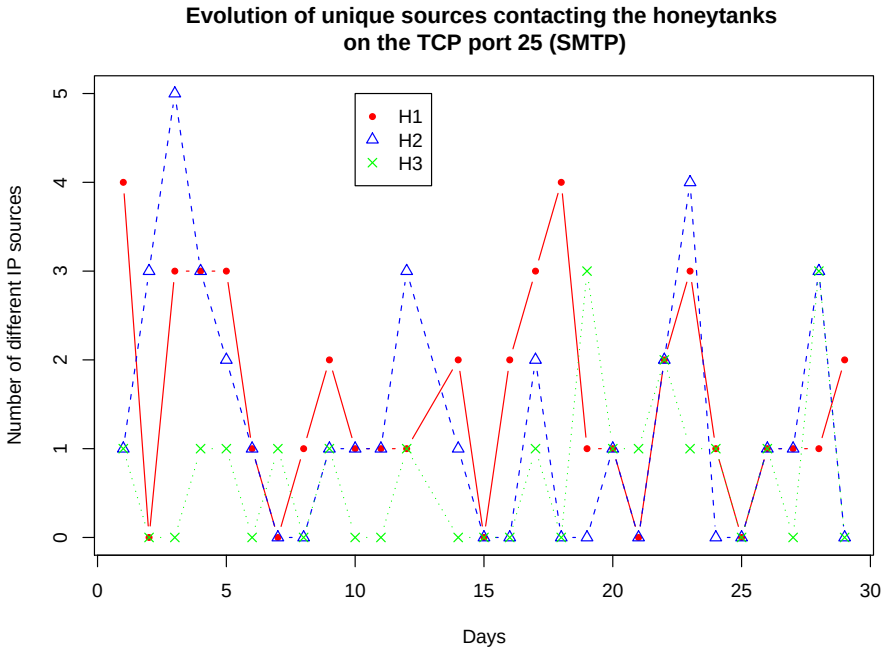

Figure 11.7: Evolution of unique sources contacting the honeytanks on the port 25

the protocol used. Then, the attacker is supposed to send its own protocol version string. The original Nepenthes module, and thus its RUSSEL translation, expects to also receive the string "RFB 003.008". If it is not the case, the simulation is desynchronized and it is not able to continue.

We manually inspected the collected trace to explain the various lengths of the connections established on **H1**. We observed that some attackers sent the string "RFB 003.005" instead of the string "RFB 003.008". As explained, the simulation cannot be continued in this case. Thus, those connections contain only one data segment, i.e., the protocol version string sent by the attacker. Connections containing no data segment might be due to attackers expecting another protocol version from the honeytank: they did not go further in their attack. Finally, connections containing two received data segments show that the attacker accepted the welcome banner sent by the honeytank and sent its own protocol version string. This string was recognized by the service simulator which sent a reply. This reply was accepted by the attacker which in turn sent a second packet to exploit a vulnerability of the simulated service.

Then, we perform the same analysis for the traffic received by **H2**. The results are shown in Figure 11.11. We observe that **H2** received at most one data segment for connections that were correctly initialized. Like **H1**, this

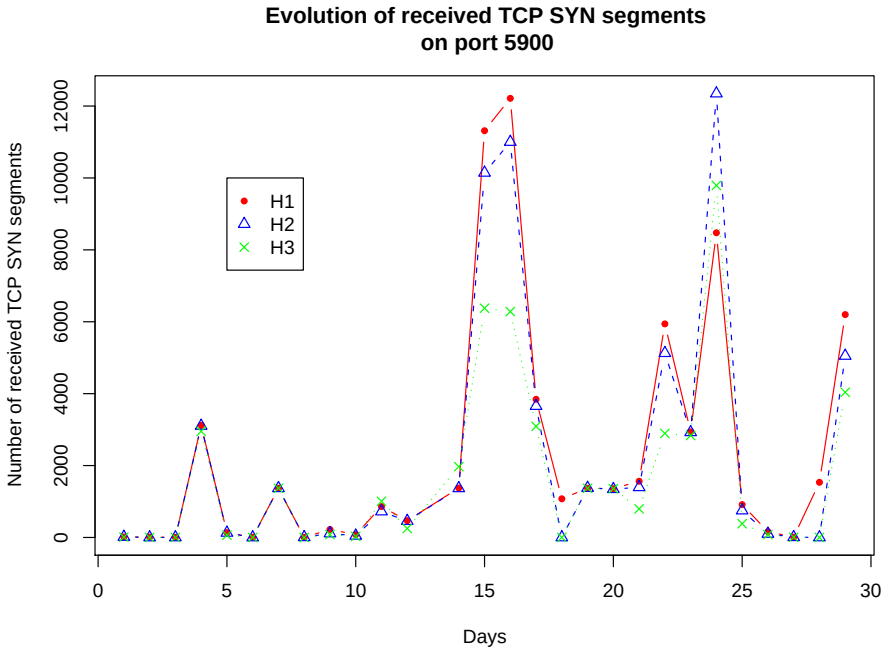

Figure 11.8: Evolution of received TCP SYN segments on the port 5900

honeytank sends its protocol version string to the attacker once a connection has been established on the port 5900. But then, each received data segment is echoed back to the attacker. Sending the protocol version string allows some attackers to send the first packet of their attack, i.e. their own protocol version string. However, when they receive the echoed reply translated from **H2** instead of the expected reply, they do not go further in their attack.

Finally, Figure 11.12 shows the results for **H3**. Connections are established but no data segment is ever received. The **H3** honeytank accepts incoming connections to the port 5900 but unlike **H1** or **H2**, it does not send a protocol version string once a connection is established. It seems that attackers expect to receive this welcome banner before performing their attack.

11.8 Conclusion of the second experiment

In this experiment, we ran three honeytanks in parallel during about a month. The opened TCP ports are identical on all honeytanks and they all use the same stateless service simulators for SMTP, HTTP and RPCinfo (bound to the ports 25, 80 and 111 respectively). However, the way they simulate some application-level protocols varied. For some the ports 135, 445, 1025 and 5900, the first honeytank **H1** uses some stateful simulators

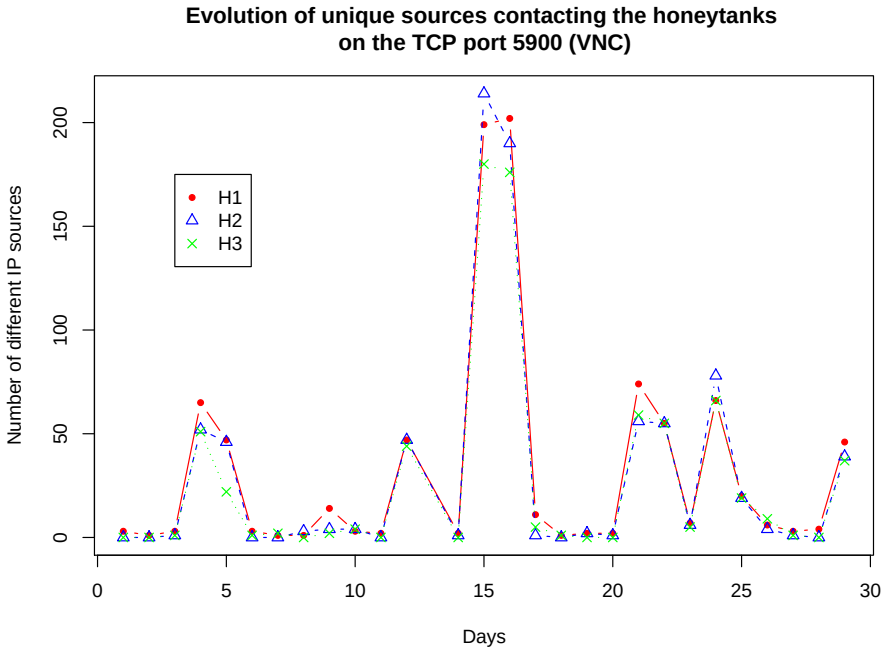

Figure 11.9: Evolution of unique sources contacting the honeytanks on the port 5900

translated from Nepenthes vulnerability modules. For the same ports, the second honeytank **H2** uses a simple echo service. Finally, the third honeytank **H3** accepts connections on those ports but ignored further received segments.

Connections requests on the port 25 and 80 shows variations from one honeytank to another. For some period of time, all three honeytanks received about the same traffic on those ports. However, for some other periods of time, the stateless honeytank received less traffic than the other two. We also observed than on some days, **H2** and **H3** were targeted by a spammer whereas **H1** received much less SMTP traffic. Further analysis is required to understand why those variations occurred.

The analysis of the collected traces shows that, when an attacker talks to a VNC service simulator on the port 5900, he goes further in his attack when he talks to the stateful simulator deployed on the first honeytank than when he talks to other honeytanks that do not implement this stateful replier. This confirms the hypothesis that stateful service simulators are needed to improve the quality of the collected traffic for some protocols. However, in regards to the ports 135, 445, 1025 and 5900, the attractiveness of the three honeytanks seems identical as they received about the same number of connections requests from about the same number of unique sources.

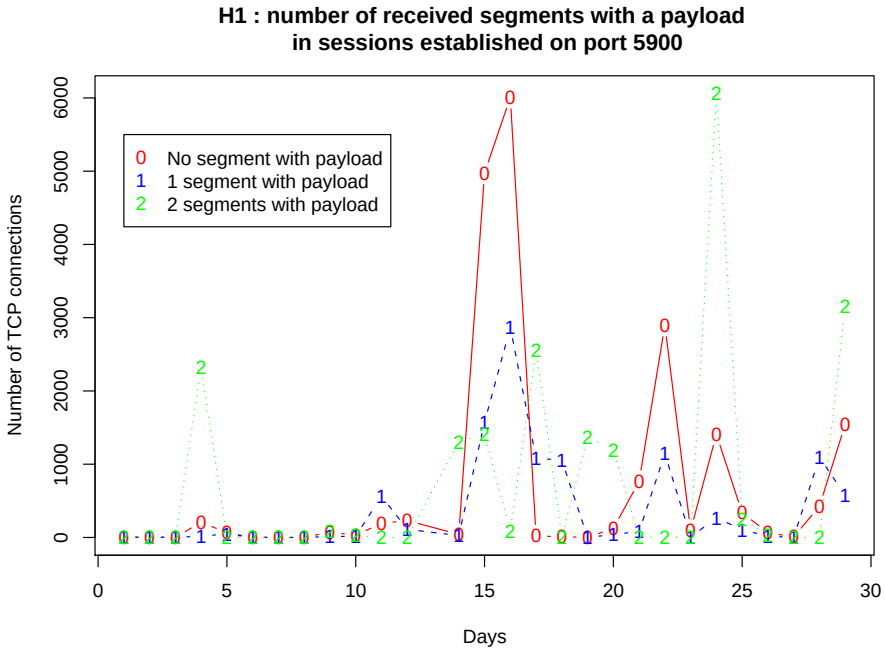

Figure 11.10: Identification of connections established on the port 5900 on H1 according to the number of data segments received for each connection

11.9 Conclusion

In this chapter, we have presented two experiments in which we deploy various honeytanks on an unused campus network. Those honeytanks are able to accept and establish incoming TCP connections and, thanks to the simulated services, they are able to collect various types of attacks. Moreover, we have shown that stateful honeytanks running stateful service simulators can improve the quality of the collected traffic for some protocols. Especially interesting is the analysis of the results on the port 5900: it shows that the most precise the simulation is, the most interesting are the collected results. In particular, only the most accurate stateful simulation is able to complete a full interaction with the attacker.

On the methodological side, these experiments show that our framework and our programming methodology are convenient to develop these honeytanks and their various service simulators. In particular, stateful Nepenthes vulnerability modules have been straightforwardly adapted. Moreover, our framework is also adequate and efficient to analyze the received traffic. Those practical examples show the versatility of our approach as compared to others.

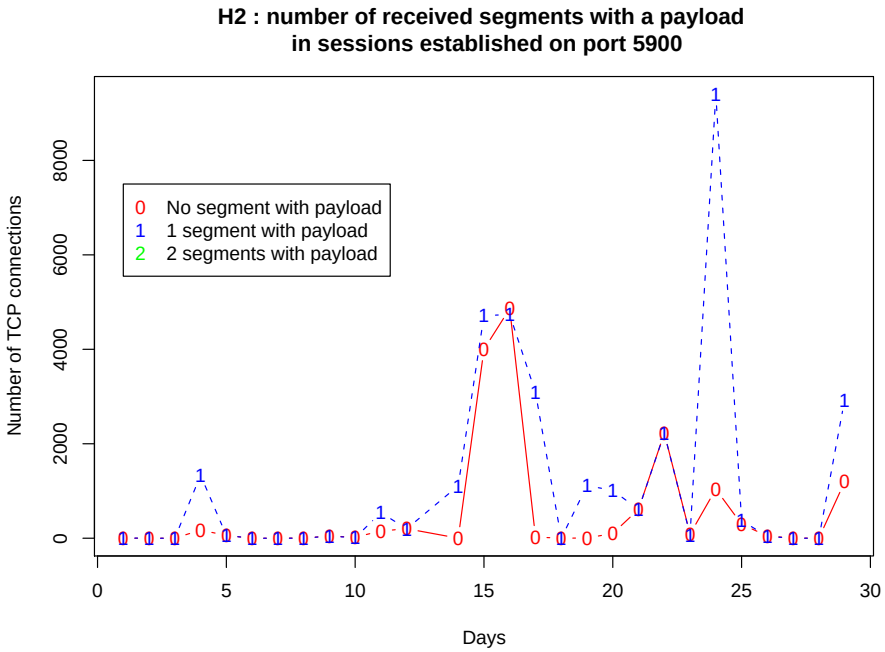

Figure 11.11: Identification of connections established on the port 5900 on H2 according to the number of data segments received for each connection

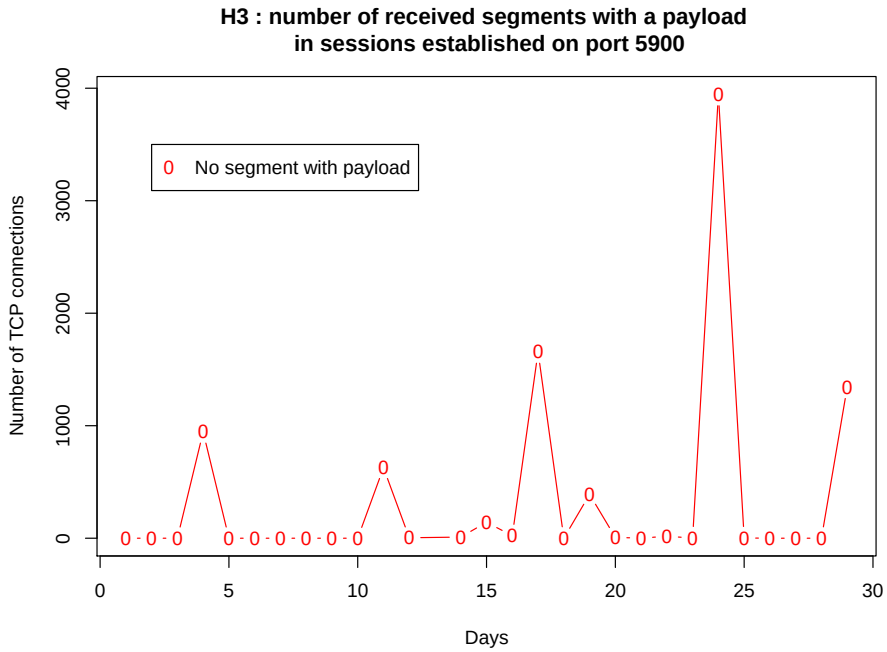

Figure 11.12: Identification of connections established on the port 5900 on H3 according to the number of data segments received for each connection

Part IV

Conclusion

Chapter 12

Conclusion

> Sometimes, the best answer is a more interesting question.
>
> Terry Pratchett
> *The Science of Discworld*

In this thesis, we have proposed a simple and systematic framework to build efficient network tools such as honeytanks, traffic generators and trace analyzers. This framework is based on the idea that such tools are best described as independent or concurrent parallel processes because they have to simulate, analyze, or interact with many independent hosts running several different services. Since many hosts and services have to be dealt with, such a framework must be supported by a light an efficient software architecture. Such an architecture is provided by ASAX, carefully optimized [27], and suitably extended to handle network traffic. In this work, we extend ASAX to properly handle network traffic and we introduce a concurrent programming methodology to simulate many hosts and services inside a single ASAX process, which executes a large number of RUSSEL rule instances in non determinate order.

We have applied the programming methodology to define several stateless and stateful honeytanks, traffic generators, and trace analyzers. We have performed both theoretical and experimental evaluation of these tools and we have compared their efficiency with other existing tools such as Honeyd, Nepenthes, and Scapy. These experiments show that our approach compares well with existing tools with respect to generality, convenience, and efficiency. We have also used the implemented honeytanks to collect real network traffic. The analysis of the collected traffic shows promising results.

Since the results presented in this thesis supports its usefulness, further research could use our approach for designing more ambitious and complete tools. To start with, existing tools such as Nepenthes, could be re-

implemented within our framework: they would become more efficient, more flexible, and more independent of pre-existing technologies. Moreover, such re-implementations of existing tools could be more easily integrated.

Bibliography

[1] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix C. Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. In Diego Zamboni and Christopher Krügel, editors, *RAID*, volume 4219 of *Lecture Notes in Computer Science*, pages 165–184. Springer, 2006.

[2] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19:2:32–48, 1989.

[3] P. Biondi. Scapy. `http://www.secdev.org/projects/scapy/`.

[4] Antanas Cenys, Darius Rainys, Lukas Radvilavicius, and Nikolaj Goranin. Implementation of Honeytoken Module in DBMS Oracle 9iR2 Enterprise Edition for Internal Malicious Activity Detection, July 2005. Presented at Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) 2005.

[5] CERT. CERT Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. `http://www.cert.org/advisories/CA-2001-19.html`.

[6] William Cheswick. An Evening with Berferd in Which a Cracker is Lured, Endured, and Studied. In *Proceedings of the Winter USENIX Conference*, pages 163–174, San Francisco, January 1992.

[7] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *NDSS*. The Internet Society, 2006.

[8] The Team Cymru. The team cymru darknet project. `http://www.cymru.com/Darknet/index.html`.

[9] Marc Dacier, Fabien Pouget, and Hervé Debar. Attack processes found on the Internet. In *NATO Research and technology symposium IST-041/RSY-013 "Adaptive Defence in Unclassified Networks", 19 April 2004, Toulouse, France*, April 2004.

[10] Marc Dacier, Fabien Pouget, and Hervé Debar. Honeypots: Practical means to validate malicious fault assumptions. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'04)*, pages 383–388. IEEE Computer Society, 2004.

[11] R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Rfc 2616 : Hypertext Transfer Protocol – HTTP/1.1. `http://www.ietf.org/rfc/rfc2616.txt`, August 1982.

[12] Fyodor. Remote OS detection via TCP/IP Stack Fingerprinting. `http://www.insecure.org/nmap/nmap-fingerprinting-article.html`.

[13] Stephane Grundschober and Marc Dacier. Design and implementation of a sniffer detector. In *Recent Advances on Intrusion Detection Workshop (RAID98)*, 1998.

[14] Naji Habra, Baudouin Le Charlier, and Aziz Mounji. Advanced Security Audit Trail Analysis on Unix: Implementation Design of the NADF evaluator. Technical Report RP 92-007, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgique, March 1992.

[15] Naji Habra, Baudouin Le Charlier, and Aziz Mounji. Preliminary Report on Advanced Security Audit Trail Analysis on Unix. Technical Report RP 92-001, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgique, January 1992.

[16] Naji Habra, Baudouin Le Charlier, Aziz Mounji, and Isabelle Mathieu. ASAX: Software Architecture and Rule-based language for Universal audit trail analysis. In Yves Deswarte, Gérard Eizenberg, and Jean-Jacques Quisquater, editors, *Proceedings of the third European Symposium on Research in Computer Security (ESORICS'92)*, volume 648 of *Lecture Notes in Computer Science*, pages 435–450, Toulouse, France, November 1992. Springer-Verlag.

[17] Seung-Sun Hong, Fiona Wong, Shyhtsun Felix Wu, Bjorn Lilja, Tony Y. Yohansson, Henric Johnson, and Ame Nelsson. TCPtransform: Property-Oriented TCP Traffic Transformation. In *DIMVA*, volume 3548 of *Lecture Notes in Computer Science*, pages 222–240. Springer, 2005.

[18] Seung-Sun Hong and Shyhtsun Felix Wu. On interactive internet traffic replay. In Alfonso Valdes and Diego Zamboni, editors, *RAID*, volume 3858 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2005.

[19] G. Huston. Bgp table report. http://bgp.potaroo.net, 2004.

[20] Insecure.Com LLC. XProbe : active OS fingerprinting tool. `http://xprobe.sourceforge.net/`.

[21] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. Request for Comments 1323, Internet Engineering Task Force, May 1992.

[22] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The click modular router. ACM Transactions on Computer Systems, August 2000.

[23] Corrado Leita, Marc Dacier, and Frédéric Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *RAID 2006, 9th International Symposium on Recent Advances in Intrusion Detection, September 20-22, 2006, Hamburg, Germany*, Sep 2006.

[24] Corrado Leita, Ken Mermoud, and Marc Dacier. ScriptGen: an Automated Script Generation Tool for Honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC 2005)*, December 2005.

[25] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, volume 2, pages 12–26. Lincoln Laboratory MIT, 2000.

[26] Tom Liston. LaBrea: Sticky Honeypot and IDS. `http://labrea.sourceforge.net/`.

[27] Xavier Martin. *Fast Sequential Implementation of a Lightweight, Data Stream Driven, Parallel Language with Application to Intrusion Detection.* PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 2007.

[28] Xavier Martin and Baudouin Le Charlier. Optimizing a rule-based intrusion detection language to handle a large number of signatures. Proceedings of the 5th Conference on Security and Network Architectures (SAR 2006), June 2006.

[29] David Moore. Network Telescopes: Observing Small or Distant Security Events. `http://www.caida.org/outreach/presentations/2002/usenix_sec/`, August 2002.

[30] David Moore. Network telescopes: Tracking denial-of-service attacks and internet worms around the globe. In *Proceedings of the 17th Conference on Systems Administration (LISA 2003)*. USENIX, 2003.

[31] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet Denial-of-Service activity. In *Proceedings of the 2001 USENIX Security Symposium*, pages 9–22, 2001.

[32] Aziz Mounji. *Languages and Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgique, September 1997.

[33] Aziz Mounji and Baudouin Le Charlier. Continuous Assessment of a Unix Configuration: Integrating Intrusion Detection and Configuration Analysis. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security (SNDSS'97)*, pages 27–37, San Diego, California, February 1997. IEEE.

[34] Aziz Mounji, Baudouin Le Charlier, Naji Habra, and Denis Zampuniéris. Distributed Audit Trail Analysis. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security (SNDSS'95)*, pages 102–113, San Diego, California, February 1995. IEEE.

[35] Aziz Mounji, Baudouin Le Charlier, Denis Zampuniéris, and Naji Habra. Preliminary Report on Distributed ASAX. Technical Report RP 94-007, Institut d'Informatique, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgique, May 1994.

[36] University of Cambridge Computer Laboratory. The Xen virtual machine monitor. `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/`.

[37] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet Background Radiation. In *Proceedings of ACM Internet Measurement Conference*, October 2004.

[38] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

[39] C. Perkins. Rfc 3344 : IP Mobility Support for IPv4. `http://www.ietf.org/rfc/rfc3344.txt`, August 2002.

[40] J. Postel. Transmission control protocol. Request for Comments 793, Internet Engineering Task Force, September 1981.

[41] J. Postel. Simple mail transfer protocol. Request for Comments 821, Internet Engineering Task Force, August 1982.

[42] J. Postel and J. K. Reynolds. RFC 959: File transfer protocol, October 1985.

[43] Fabien Pouget, Marc Dacier, and Hervé Debar. Honeypots: a comparative survey. Eurecom Report, RR-03-81, July 2003.

[44] Jean-Philippe Pouzol and Mireille Ducassé. From declarative Signatures to Misuse IDS. In Wenke Lee, Ludovic M, and Andreas Wespi, editors, *Recent Advances in Intrusion Detection (RAID 2001)*, LNCS 2212, Davis, CA, USA, October 2001. IRISA/INSA de Rennes, Springer-verlag.

[45] Jean-Philippe Pouzol and Mireille Ducassé. Formal Specification of intrusion Signatures and Detection Rules. In *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW) 2002*. IRISA/INSA de Rennes, 2002.

[46] The Honeynet Project. *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*. The Honeynet Project, 2002.

[47] N. Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

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

[49] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM.

[50] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of USENIX LISA'99*, 1999.

[51] Salvatore Sanfilippo. hping. `http://www.hping.org/`.

[52] Mike Schiffman. The libnet packet construction library. `http://www.packetfactory.net/libnet/`.

[53] Dug Song, G. Shaffer, and M. Undy. Nidsbench - a network intrusion detection test suite. In *Recent Advances in Intrusion Detection*, 1999.

[54] Lance Spitzner. *Honeypots — Tracking Hackers*. Addison-Wesley Professional, September 2002.

[55] Lance Spitzner. Honeytokens: The Other Honeypot. `http://www.securityfocus.com/infocus/1713`, July 2003.

[56] R. Srinivasan. Binding protocols for ONC RPC version 2. Request for Comments 1833, Internet Engineering Task Force, August 1995.

[57] R. Srinivasan. RPC: remote procedure call protocol specification version 2. Request for Comments 1831, Internet Engineering Task Force, August 1995.

[58] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[59] Clifford Stoll. Stalking the wily hacker. *Communications of the ACM*, 31(5):484–497, 1988.

[60] Clifford Stoll. *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage*. Doubleday, 1989.

[61] Tcpdump-team. The Packet Capture library (libpcap) / tcpdump. `http://www.tcpdump.org/`.

[62] MAFTIA Team. Malicious-and accidental-fault tolerance for internet applications. `http://www.maftia.org/`.

[63] The honeynet project. Know Your Enemy: Honeynets. `http://www.honeynet.org/papers/honeynet/`, May 2006.

[64] Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson, and Kevin Gibbs. iPperf. `http://dast.nlanr.net/Projects/Iperf/`.

[65] Aaron Turner. The TCPreplay homepage. `http://tcpreplay.sourceforge.net/`.

[66] Guido van Rossum. Python programming language. `http://www.python.org/`.

[67] 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*, 4(1/2), 2008.

[68] Nicolas Vanderavero and Baudouin Le Charlier. Towards a more stateful and accurate HoneyTank. Proceedings of the 5th Conference on Security and Network Architectures (SAR 2006), June 2006.

[69] 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. *SIGOPS Oper. Syst. Rev.*, 39(5):148–162, 2005.

[70] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, nov 2002.

[71] XProbe team. NMAP the network mapper. `http://www.insecure.org/nmap/`.

[72] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. In *RAID 2004 Symposium*, September 2004.

Appendices

Appendix A

Overview of the number of lines of code

In this appendix, we present a summary of the number of lines of code of our various implementations.

Concerning the honeytanks, the stateless honeytank is written in about 450 lines of RUSSEL while the stateful honeytank uses about 2,500 RUSSEL lines of code. The stateless application simulators are written in about 200 lines of RUSSEL code. Each stateful application simulator translated from Nepenthes uses about 500 lines of RUSSEL code.

The basic module of the traffic generator uses about 500 lines of code while the module initiating and closing TCP connections is written in about 1,000 lines of RUSSEL code.

Finally, the network format adapter is written in about 1,400 lines of C code and uses a NADF record creation library which is written in about 350 lines of codes.

Appendix B

Assessment of honeytanks and Honeyd

This appendix presents the raw results of the assessments of our honeytanks and Honeyd. Those tables were used to plot the figures presented in Chapter 10.

Inter-transaction time	Connections initiated	Connections established	Connections closed	Response time to accepted SYN (in microsec.)
10000	1000	1000	1000	315.758
5000	2000	2000	2000	296.387
3333	3000	3000	3000	346.4103
2500	4000	4000	4000	357.2935
2000	5000	5000	5000	389.2334
1666	6002	6002	6002	507.4932
1428	7002	7002	7002	1161.633
1250	8000	7709	7455	66469.1
1111	9000	7975	7070	127634.9
1000	10000	8128	6472	138222.1
800	12500	8664	4996	158305.1
667	14992	9208	3789	162346.3
500	20000	10444	3495	153503.7
250	40000	11969	789	186936.4
125	80000	12090	315	242693.3

Table B.1: Honeyd with Python internal service tested with our generator

Inter-transaction time	Connections initiated	Connections established	Connections closed	Response time to SYN (microsec.)
10000	1000	1000	1000	386.6611
5000	2000	2000	2000	331.608
3333	3000	3000	3000	279.4880
2500	4000	4000	4000	353.925
2000	5000	5000	5000	277.1386
1666	6002	6002	6002	355.5839
1428	7002	7002	7002	421.4719
1250	8000	8000	8000	506.7437
1111	9000	9000	9000	592.5822
1000	10000	10000	10000	493.4088
800	12500	12500	12500	5263.661
667	14992	14992	14992	608953.6
500	20000	20000	20000	1364395
250	40000	40000	40000	5086422
125	79899	79485	79484	11855475

Table B.2: Unbounded stateful honeytank tested with our generator

Inter-transaction time	Connections initiated	Connections established	Connections closed	Response time to accepted SYN (in microsec.)
1000	10000	9951	9951	274.5302
800	12500	12427	12427	338.8093
667	14992	13403	13403	2031.852
500	20000	13163	13163	1950.353
250	40000	10148	10148	1808.222
125	79982	4233	4233	1434.733

Table B.3: Bounded (10 packets) stateful honeytank tested with our generator

Inter-transaction time	Connections initiated	Connections established	Connections closed	Response time to accepted SYN (in microsec.)
1000	10000	10000	10000	487.1564
800	12500	12491	12491	912.1351
667	14992	13391	13391	17947.65
500	20000	12158	12158	17988.78
250	40000	9553	9553	14971.96
125	80000	4483	4483	11307.55

Table B.4: Bounded (100 packets) stateful honeytank tested with our generator

Inter-transaction time	Connections initiated	Connections established	Connections closed	Response time to accepted SYN (in microsec.)
1000	10000	10000	10000	473.7885
800	12500	12500	12500	908.6275
667	14992	13693	13693	152712.6
500	20000	13088	13088	165487.5
250	40000	10288	10288	142183.3
125	79990	5090	5090	111600.9

Table B.5: Bounded (1000 packets) stateful honeytank tested with our generator

Inter-transaction time	Connections initiated	Connections established	Connections closed	Response time to accepted SYN (in microsec.)
1000	10000	10000	10000	154.7104
800	12500	12500	12500	155.6699
667	14992	14992	14992	156.3232
500	20000	20000	20000	213.1483
250	40000	40000	40000	1554610
125	61948	61948	61948	5687847

Table B.6: Unbounded stateless honeytank tested with our generator

SYN segments sent pers second	Number of correct SYN+ACK segments received	Response time to accepted SYN (in microsec.)
100	20000	303.0412
200	20000	309.6263
300	20000	392.0872
400	20000	444.8066
500	20000	529.5771
600	20000	593.9851
700	20000	716.1384
800	20000	819.9994
900	20000	1067.990
1000	20000	1117.654
1250	20000	2006.465
1500	20000	2845.902
2000	19872	23668.00
4000	12041	165937.8
8000	6303	161988.2
20000	2010	215105.1

Table B.7: Honeyd with Python internal service tested with TCPreplay

SYN segments sent pers second	Number of correct SYN+ACK segments received	Response time to accepted SYN (in microsec.)
100	20000	327.3464
200	20000	321.895
300	20000	387.9648
400	20000	451.1053
500	20000	528.5343
600	20000	610.7662
700	20000	700.1898
800	20000	744.4068
900	20000	898.3839
1000	20000	976.4918
1250	20000	1263.842
1500	20000	1572.068
2000	20000	2318.836
4000	20000	225854.3
8000	20000	1069758
20000	20000	1818845

Table B.8: Unbounded stateful honeytank tested with TCPreplay

SYN segments sent pers second	Number of correct SYN+ACK segments received	Response time to accepted SYN (in microsec.)
100	20000	327.2631
200	20000	328.8523
300	20000	382.1055
400	20000	452.3050
500	20000	534.5714
600	20000	612.6441
700	20000	682.0335
800	20000	753.3024
900	20000	896.0457
1000	20000	1006.078
1250	20000	1268.428
1500	20000	1575.332
2000	20000	2402.183
4000	17597	124598.8
8000	5858	122595.5
20000	1112	475712.5

Table B.9: Bounded (1000 packets) stateful honeytank tested with TCPreplay

Appendix C

Stateless repliers RUSSEL source code

Listing C.1: Stateless HTTP service simulator in RUSSEL

```
1  rule http_replier;
2  var reply : string;
3  begin
4    if
5      match('GET (.*) HTTP/(.*)', tcp_payload) = 1 -->
6      begin
7        reply := concat('HTTP/1.0 200 OK', X'0D0A');
8        reply := concat(reply, X'0D0A0D0A');
9        reply := concat(reply, '<?xml version="1.0" encoding="utf-8"?><!
             DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http
             ://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"><html><head></
             head><body></body></html>')
10     end;
11
12     match('HEAD (.*) HTTP/(.*)', tcp_payload) = 1 -->
13     begin
14       reply := concat('HTTP/1.0 200 OK', X'0D0A');
15       reply := concat(reply, 'Last-modified: Tue, 04 Nov 2005 12:38:41
             GMT');
16       reply := concat(reply, X'0D0A0D0A')
17     end;
18
19     match('CONNECT (.*) HTTP/(.*)', tcp_payload) = 1 -->
20     reply := concat('HTTP/1.0 200 OK', X'0D0A0D0A');
21
22     match('TRACE (.*) HTTP/(.*)', tcp_payload) = 1 -->
23     begin
24       reply := concat('HTTP/1.0 200 OK', X'0D0A');
25       reply := concat(reply, 'Content-Type: text/html');
26       reply := concat(reply, X'0D0A0D0A')
27     end;
28
29     match('OPTIONS (.*) HTTP/(.*)', tcp_payload) = 1 -->
30     begin
31       reply := concat('HTTP/1.0 200 OK', X'0D0A');
32       reply := concat(reply, 'Allow: HEAD, GET, POST, TRACE, OPTIONS,
             PUT, DELETE');
33       reply := concat(reply, X'0D0A0D0A')
34     end;
35
36     match('PUT (.*) HTTP/(.*)', tcp_payload) = 1 -->
37     reply := concat('HTTP/1.0 201 Created', X'0D0A0D0A');
38
39     match('DELETE (.*) HTTP/(.*)', tcp_payload) = 1 -->
40     reply := concat('HTTP/1.0 200 OK', X'0D0A0D0A');
41
42     match('POST (.*) HTTP/(.*)', tcp_payload) = 1 -->
43     reply := concat('HTTP/1.0 200 OK', X'0D0A0D0A');
44
45     true --> reply := concat('HTTP/1.0 404 ERROR', X'0D0A0D0A')
46   fi;
47
48   trigger off for_current send_reply(ethernet_dest, ethernet_source,
       ip_dest, ip_source, default_ip_tos, default_ip_id, default_ip_frag
       , default_ip_ttl, rawToInt(tcp_dest), rawToInt(tcp_source),
       rawToInt(tcp_ack), rawToInt(tcp_sequence) + stringlen(tcp_payload)
       , 17, default_tcp_window, default_tcp_urgent, reply)
49 end;
```

Listing C.2: Stateless SMTP service simulator in RUSSEL

```
1  rule smtp_replier;
2  var reply : string;
3      quit : integer;
4  begin
5    quit := 0;
6
7    if
8      match('^[Hh][Ee][Ll](.*)', tcp_payload) = 1 --> reply := concat('250
           Hello, pleased to meet you', X'0D0A');
9      match('^[Mm][Aa][Ii][Ll] [Ff][Rr][Oo][Mm]:(.*)', tcp_payload) = 1
           --> reply := concat('250 Sender OK', X'0D0A');
10     match('^[Rr][Cc][Pp][Tt] [Tt][Oo]:(.*)', tcp_payload) = 1 --> reply
           := concat('250 Recipient OK', X'0D0A');
11     match('^[Dd][Aa][Tt][Aa](.*)', tcp_payload) = 1
12     --> reply := concat('354 Enter mail, end with "." on a line by
           itself', X'0D0A');
13
14     match('^[Rr][Ss][Ee][Tt](.*)', tcp_payload) = 1 --> reply := concat(
           '250 Reset state', X'0D0A');
15     match('^[Nn][Oo][Oo][Pp](.*)', tcp_payload) = 1 --> reply := concat(
           '250 OK', X'0D0A');
16     match('^[Ee][Hh][Ll][Oo](.*)', tcp_payload) = 1 --> reply := concat(
           '502 COMMAND NOT SUPPORTED', X'0D0A');
17     match('^\.(.*)', tcp_payload) = 1 --> reply := concat('250 Message
           accepted for delivery', X'0D0A');
18     match('^[Qq][Uu][Ii][Tt](.*)', tcp_payload) = 1 -->
19     begin
20         reply := concat('221 Closing connection', X'0D0A');
21         quit := 1
22       end;
23     true --> reply := ''
24   fi;
25
26   if
27     quit = 0 -->
28     trigger off for_current send_tcp_datagram(ethernet_dest,
           ethernet_source, ip_dest, ip_source, default_ip_tos,
           default_ip_id, default_ip_frag, default_ip_ttl, rawToInt(
           tcp_dest), rawToInt(tcp_source), rawToInt(tcp_ack), rawToInt(
           tcp_sequence) + stringlen(tcp_payload), 16, default_tcp_window,
           default_tcp_urgent, reply);
29
30     true -->
31     trigger off for_current send_tcp_datagram(ethernet_dest,
           ethernet_source, ip_dest, ip_source, default_ip_tos,
           default_ip_id, default_ip_frag, default_ip_ttl, rawToInt(
           tcp_dest), rawToInt(tcp_source), rawToInt(tcp_ack), rawToInt(
           tcp_sequence) + stringlen(tcp_payload), 17, default_tcp_window,
           default_tcp_urgent, reply)
32
33   fi
34 end;
```

Listing C.3: Stateless RPCInfo service simulator in RUSSEL

```
1  rule rpc_replier;
2  var reply : string;
3  begin
4    if
5      substring(tcp_payload, 16, 19) = X'000186A0' -->
6      begin
7        reply := concat(X'800001AC', substring(tcp_payload, 4, 7));
8            reply := concat(reply, X'00000001000000000000000000000000000000000000000000000001');
9            reply := concat(reply, X'000186A000000002000000060000006F00000001');
10           reply := concat(reply, X'000186A100000002000000060000006F00000001');
11             reply := concat(reply, X'000186A200000002000000060000006F00000001');
12             reply := concat(reply, X'000186A300000002000000060000006F00000001');
13             reply := concat(reply, X'000186A400000002000000060000006F00000001');
14             reply := concat(reply, X'000186A500000002000000060000006F00000001');
15             reply := concat(reply, X'000186A600000002000000060000006F00000001');
16             reply := concat(reply, X'000186A700000002000000060000006F00000001');
17             reply := concat(reply, X'000186A800000002000000060000006F00000001');
18             reply := concat(reply, X'000186A900000002000000060000006F00000001');
19             reply := concat(reply, X'000186B000000002000000060000006F00000001');
20             reply := concat(reply, X'000186B100000002000000060000006F00000001');
21             reply := concat(reply, X'000186B200000002000000060000006F00000001');
22             reply := concat(reply, X'000186B300000002000000060000006F00000001');
23             reply := concat(reply, X'000186B400000002000000060000006F00000001');
24             reply := concat(reply, X'000186B500000002000000060000006F00000001');
25             reply := concat(reply, X'000186B600000002000000060000006F00000001');
26             reply := concat(reply, X'000186B700000002000000060000006F00000001');
27             reply := concat(reply, X'000186B800000002000000060000006F00000001');
28             reply := concat(reply, X'000186B900000002000000060000006F00000000')
29         end;
30     true --> reply := ''
31   fi;
32
33   trigger off for_current send_tcp_datagram(ethernet_dest, ethernet_source,
34   ip_dest, ip_source, default_ip_tos, default_ip_id, default_ip_frag, default_ip_ttl,
35   rawToInt(tcp_dest), rawToInt(tcp_source),
36   rawToInt(tcp_ack), rawToInt(tcp_sequence) + stringlen(tcp_payload), 17,
37   default_tcp_window, default_tcp_urgent, reply)
38
39 end;
```

Appendix D

Source code of stateful Nepenthes vulnerability modules translated in RUSSEL

This appendix presents the source code of the Nepenthes vulnerability modules translated in RUSSEL that have been deployed in a stateful honeytank used in the experiment presented in Chapter 11. The technique for translating Nepenthes vulnerability modules in RUSSEL has been presented in Chapter 8. It uses the programming methodology for writing optimizable rules presented in Chapter 7.

Listing D.1: Translation of VNC vulnerability module

```
1  global rfb_version_003_008 : string;
2  global vnc_image : string;
3
4
5  init_action;
6  begin
7    rfb_version_003_008 := X'524642203030332E3030380D0A';
8    vnc_image := get_vnc_image
9  end;
10
11
12 external rule emulate_application_realvnc_opt(
13       hash                      : integer; /* hash of the 4-uple
             identifier identifying the TCP session
14                                                 associated with this
                                                      emulation */
15       ip_s, ip_d, tcp_s, tcp_d  : integer; /* 4-uple identifying the TCP
             session associated with this emulation */
16       buffer                    : string;  /* Input stream buffer */
17       state                     : integer; /* State to maintain */
```

```
18      sendimage                   : integer
19  );
20  if
21      stop_all = 1 --> skip;
22
23      rawToInt(tcp_session_hash) = hash and  h1_valid_tcp_segment = 1 -->
24    trigger off for_current emulate_application_realvnc(hash, ip_s, ip_d,
          tcp_s, tcp_d, buffer, state, sendimage);
25
26    true -->
27      trigger off for_next emulate_application_realvnc_opt(hash, ip_s,
            ip_d, tcp_s, tcp_d, buffer, state, sendimage)
28  fi
29  end;
30
31
32  rule emulate_application_realvnc(
33    hash                        : integer; /* hash of the 4-uple identifier
          identifying the TCP session
34                                                  associated with this
                                                       emulation */
35      ip_s, ip_d, tcp_s, tcp_d  : integer; /* 4-uple identifying the TCP
            session associated with this emulation */
36      buffer                      : string;  /* Input stream buffer */
37      state                       : integer;  /* State to maintain */
38    sendimage                   : integer
39  );
40
41
42  var
43      new_record_num_timeout : integer;
44      new_timeout_sec, new_timeout_usec : integer;
45      new_buffer : string;
46      new_state : integer;
47
48    retrigger : integer;
49
50    reply : string;
51
52    needmore : integer;
53    c : string;
54    num : integer;
55    cpbytes : integer;
56    new_sendimage : integer;
57    rc : integer;
58
59  if
60    /* Is the current segment destinated to the session used by this
          application and is it valid ? */
61    rawToInt(ip_source)  = ip_s  and rawToInt(ip_dest)  = ip_d  and
62    rawToInt(tcp_source) = tcp_s and rawToInt(tcp_dest) = tcp_d -->
63    begin
64      new_sendimage := sendimage;
65
```

```
66      retrigger := 1;
67      reply := '';
68
69      new_buffer := string_concat(buffer, tcp_payload);
70      new_state := state;
71
72      if
73        new_state = 1 -->
74        if
75          string_startsWith(new_buffer, rfb_version_003_008) = 1 -->
76          begin
77            string_remove(new_buffer, 0, string_len(rfb_version_003_008)
                  -1);
78
79            reply := X'0102';
80            retrigger := 1;
81
82            new_state := 2
83          end
84        fi;
85
86        new_state = 2 -->
87        if string_len(new_buffer) >= 1 -->
88          begin
89            string_remove(new_buffer, 0, 0);
90
91            reply := X'00000000';
92            retrigger := 1;
93
94            new_state := 3
95          end
96        fi;
97
98        new_state = 3 -->
99        if string_len(new_buffer) >= 1 -->
100         begin
101           string_remove(new_buffer, 0, 0);
102
103           reply := X'043a02ff2018000100ff00ff00ff1008000000000000000␣
                  d464f4f4241522d4d5554544552';
104           retrigger := 1;
105
106           new_state := 4
107         end
108       fi;
109
110       new_state = 4 -->
111       begin
112         needmore := 0;
113         reply := '';
114         retrigger := 1;
115
116         do
117           string_len(new_buffer) >= 1 and needmore = 0 -->
```

```
118             begin
119               c := string_substring(new_buffer, 0, 0);
120               if
121                 c = X'00' -->
122                 if
123                   string_len(new_buffer) >= 20 -->
124                   string_remove(new_buffer, 0, 19);
125                   true --> needmore := 1
126                 fi;
127
128                 c = X'01' --> skip;
129
130                 c = X'02' -->
131                 begin
132                   num := rawToIntNetwork(string_substring(new_buffer, 2,
                          3));
133                   if
134                     string_len(new_buffer) >= 4 + num * 4 -->
135                       string_remove(new_buffer, 0, (4 + num * 4) - 1);
136                     true --> needmore := 1
137                   fi
138                 end;
139
140                 c = X'03' -->
141                 begin
142                   if
143                     string_len(new_buffer) >= 10 -->
144                     string_remove(new_buffer, 0, 9);
145                     true --> needmore := 1
146                   fi;
147
148                   if
149                     sendimage = 0 -->
150                     begin
151                       reply := vnc_image;
152                       retrigger := 1;
153
154                       new_sendimage := 1
155                     end
156                   fi
157                 end;
158
159                 c = X'04' -->
160                 if
161                   string_len(new_buffer) >= 8 -->
162                   string_remove(new_buffer, 0, 7);
163                   true --> needmore := 1
164                 fi;
165
166                 c = X'05' -->
167                 if
168                   string_len(new_buffer) >= 6 -->
169                   string_remove(new_buffer, 0, 5);
170                   true --> needmore := 1
```

```
171                 fi;
172
173                 c = X'06' -->
174                 if
175                   string_len(new_buffer) >= 8 -->
176                   begin
177                     cpbytes := rawToIntNetwork(string_substring(new_buffer
                            , 4, 7));
178                     if
179                       string_len(new_buffer) >= 8+cpbytes -->
180                       string_remove(new_buffer, 0, 8+cpbytes-1);
181                       true --> needmore := 1
182                     fi
183                   end;
184                   true --> needmore := 1
185                 fi;
186
187                 true --> needmore := 1
188               fi
189             end
190           od
191         end
192       fi;
193
194       if
195         retrigger = 0 -->
196         /* Send reply or acknowledge the received segment and send a RST
                */
197         trigger off for_current send_tcp_datagram(ethernet_dest,
              ethernet_source, ip_dest, ip_source, default_ip_tos,
              default_ip_id, default_ip_frag, default_ip_ttl, rawToInt(
              tcp_dest), rawToInt(tcp_source), rawToInt(tcp_ack), rawToInt(
              tcp_sequence) + stringlen(tcp_payload), 17, default_tcp_window
              , default_tcp_urgent, reply);
198
199
200         true -->
201         begin
202           /* Send reply or acknowledge the received segment */
203           trigger off for_current send_tcp_datagram(ethernet_dest,
                ethernet_source,
204           ip_dest, ip_source, default_ip_tos, default_ip_id,
                default_ip_frag, default_ip_ttl,
205           rawToInt(tcp_dest), rawToInt(tcp_source),
206           rawToInt(tcp_ack), rawToInt(tcp_sequence) + stringlen(
                tcp_payload), 16,
207           default_tcp_window, default_tcp_urgent, reply);
208
209
210           /* Retrigger the rule */
211           trigger off for_next emulate_application_realvnc_opt(hash, ip_s,
                 ip_d, tcp_s, tcp_d, new_buffer, new_state, new_sendimage)
212         end
213       fi
```

```
214   end
215 fi;
```

Listing D.2: Translation of LSASS vulnerability module

```
1  global lsass_hod_req1, lsass_hod_req2, lsass_hod_req3, lsass_hod_req5,
       lsass_hod_req6, lsass_hod_req7 : string;
2
3
4  init_action;
5  begin
6    lsass_hod_req1 :=
7  X'00000085FF534D4272000000001853C800000000000000000000000000000
8  0FFFE000000000062000250432O4E4554574F524B2050524F4752414D2031
9  2E3000024C414E4D414E312E30000257696E646F777320666F7220576F726
10 B67726F75707320332E316100024C4D312E325830303200024C414E4D414E
11 322E3100024E54204C4D20302E313200';
12
13   lsass_hod_req2 :=
14 X'000000A4FF534D4273000000001807C800000000000000000000000000000
15 FFFE000010000CFF00A40004110A000000000000002000000000D4000080
16 69004E544C4D5353500001000000978208E000000000000000000000000000
17 00000000570069006E0064006F00770073002000320030003000300020003 2
18 003100390035000000570069006E0064006F00770073002000320030003000
19 3000200035002E00300000000000';
20
21
22   lsass_hod_req3 :=
23 X'000000DAFF534D4273000000001807C800000000000000000000000000000
24 FFFE000820000CFF00DA0004110A000000000000057000000000D4000080
25 9F004E544C4D5353500003000000010001004600000000000000047000000000
26 0000004000000000000000040000000060006004000000010001000470000000
27 158A88E048004F0044000081196A7AF2E4491C28AF30257410675357006900
28 6E0064006F00770073002000320030003000300020003200310039003500000
29 00570069006E0064006F00770073002000320030003000300020003500 2E00
30 300000000000';
31
32
33   lsass_hod_req5 :=
34 X'00000064FF534D42A2000000001807C800000000000000000000000000008
35 DC040008400018FF00DEDE000E00160000000000000009F0102000000000000
36 0000000000000003000000010000004000000002000000031100005C006C00
37 7300610072007000630000000';
38
39   lsass_hod_req6 :=
40 X'0000009CFF534D4225000000001807C800000000000000000000000000008
41 DC040008500010000048000000000400000000000000000000000054004800
42 5400020026000040590010 5C00500049005000450 05C000000000050003B03
43 1000000048000000010000000B810B810000000000100000000000001006A2819
44 390CB1D0119BA800C04FD92EF500000000045D888AEB1CC9119FE808002B10
45 486002000000';
46
47   lsass_hod_req7 :=
48 X'00000CF4FF534D4225000000001807C800000000000000000000000000008
```

```
49 DC0400086000100000A00C000000040000000000000000000000005400A00C
50 5400020026000040B10C105C00500049005000045005C000000000005000003
51 10000000A00C000001000000880C000000000900EC03000000000000EC030000'
52
53 end;
54
55
56
57
58 external rule emulate_application_lsass_opt(
59      hash                        : integer; /* hash of the 4-uple
           identifier identifying the TCP session
60                                                  associated with this
                                                       emulation */
61      ip_s, ip_d, tcp_s, tcp_d   : integer; /* 4-uple identifying the TCP
           session associated with this emulation */
62      buffer                      : string;  /* Input stream buffer */
63      state                       : integer  /* State to maintain */
64 );
65
66 begin
67     rawToInt(tcp_session_hash) = hash and h1_valid_tcp_segment = 1 -->
68   trigger off for_current emulate_application_lsass(hash, ip_s, ip_d,
        tcp_s, tcp_d, buffer, state)
69
70   true -->
71     trigger off for_next emulate_application_lsass_opt(hash, ip_s, ip_d,
           tcp_s, tcp_d, buffer, state)
72 fi;
73
74
75
76 rule emulate_application_lsass(
77   hash                        : integer; /* hash of the 4-uple identifier
        identifying the TCP session
78                                                  associated with this
                                                       emulation */
79     ip_s, ip_d, tcp_s, tcp_d   : integer; /* 4-uple identifying the TCP
          session associated with this emulation */
80     buffer                      : string;  /* Input stream buffer */
81     state                       : integer  /* State to maintain */
82 );
83
84
85 var
86     new_record_num_timeout : integer;
87     new_timeout_sec, new_timeout_usec : integer;
88     new_buffer : string;
89     new_state : integer;
90
91   retrigger : integer;
92
93   reply : string;
94
```

```
95
96  if
97    /* Is the current segment destinated to the session used by this
          application and is it valid ? */
98    rawToInt(ip_source)  = ip_s  and rawToInt(ip_dest)  = ip_d  and
99    rawToInt(tcp_source) = tcp_s and rawToInt(tcp_dest) = tcp_d -->
100   begin
101     retrigger := 1;
102     new_state := state;
103     new_buffer := string_concat(buffer, tcp_payload);
104
105     reply := string_random(512);
106
107     if
108       state = 1 -->
109       if
110         string_startsWith(new_buffer, lsass_hod_req1) = 1 -->
111         begin
112           new_state := 2;
113           new_buffer := '';
114           string_replace(reply, X'00', 9);
115
116           reply := string_substring(reply, 0, 63);
117           retrigger := 1
118         end;
119
120         true -->
121         begin
122           reply := '';
123           retrigger := 0
124         end
125       fi;
126
127       state = 2 -->
128       if
129         string_startsWith(new_buffer, lsass_hod_req2) = 1 -->
130         begin
131           new_state := 3;
132           new_buffer := '';
133           string_replace(reply, X'00', 9);
134
135           reply := string_substring(reply, 0, 63);
136           retrigger := 1
137         end;
138
139         true -->
140         begin
141           reply := '';
142           retrigger := 0
143         end
144       fi;
145
146
147       state = 3 -->
```

```
148         if
149           string_startsWith(new_buffer, lsass_hod_req3) = 1 -->
150           begin
151             new_state := 4;
152             new_buffer := '';
153             string_replace(reply, 'W␣i␣n␣d␣o␣w␣s␣␣␣5␣.␣1␣', 48);
154
155             reply := string_substring(reply, 0, 255);
156             retrigger := 1
157           end;
158
159           true -->
160           begin
161               reply := '';
162             retrigger := 0
163           end
164         fi;
165
166
167         state = 4 -->
168         if
169           string_len(new_buffer) >= 50 -->
170           begin
171             new_state := 5;
172             new_buffer := '';
173
174             reply := string_substring(reply, 0, 63);
175             retrigger := 1
176           end;
177
178           true -->
179           begin
180             reply := '';
181             retrigger := 0
182           end
183         fi;
184
185
186         state = 5 -->
187         if
188           string_startsWith(new_buffer, lsass_hod_req5) = 1 -->
189           begin
190             new_state := 6;
191             new_buffer := '';
192
193             reply := string_substring(reply, 0, 63);
194             retrigger := 1
195           end;
196
197           true -->
198           begin
199             reply := '';
200             retrigger := 0
201           end
```

```
202       fi;
203
204
205       state = 6 -->
206       if
207         string_startsWith(new_buffer, lsass_hod_req6) = 1 -->
208         begin
209           new_state := 7;
210           new_buffer := '';
211
212           reply := string_substring(reply, 0, 63);
213           retrigger := 1
214         end;
215
216         true -->
217         begin
218           reply := '';
219           retrigger := 0
220         end
221       fi;
222
223       state = 7 -->
224       begin
225         reply := string_substring(reply, 0, 63);
226         retrigger := 0
227       end
228     fi;
229
230
231     if
232       retrigger = 0 -->
233       /* Send reply or acknowledge the received segment and send a RST
              */
234         trigger off for_current send_tcp_datagram(ethernet_dest,
              ethernet_source, ip_dest, ip_source, default_ip_tos,
              default_ip_id, default_ip_frag, default_ip_ttl, rawToInt(
              tcp_dest), rawToInt(tcp_source), rawToInt(tcp_ack), rawToInt
              (tcp_sequence) + stringlen(tcp_payload), 17,
              default_tcp_window, default_tcp_urgent, reply);
235
236       true -->
237       begin
238         /* Send reply or acknowledge the received segment */
239         trigger off for_current send_tcp_datagram(ethernet_dest,
              ethernet_source, ip_dest, ip_source, default_ip_tos,
              default_ip_id, default_ip_frag, default_ip_ttl, rawToInt(
              tcp_dest), rawToInt(tcp_source), rawToInt(tcp_ack), rawToInt
              (tcp_sequence) + stringlen(tcp_payload), 16,
              default_tcp_window, default_tcp_urgent, reply);
240
241
242         /* Retrigger the rule */
243         trigger off for_next emulate_application_lsass_opt(hash, ip_s,
              ip_d, tcp_s, tcp_d, new_buffer, new_state)
```

```
      end
    fi
  end
fi;
```

Listing D.3: Translation of DCOM vulnerability module

```
global dcom_bindstr, dcom2_bindstr : string;
global sol2k_request : string;
global unknown_req1, dcom_unknown_req2 : string;
global ntscan_req1 : string;
global w2kuuid_sig : string;
global rpcfp_inqifids : string;

init_action;
begin
  dcom_bindstr :=
X'05000B0310000000480000001000000D016D0160000000010000000000
010080BDA8AF8A7DC911BEF408002B10298901000000045D888AEB1CC9119F
E808002B10486002000000';

  dcom2_bindstr :=
X'05000B0310000000480000007F000000D016D0160000000010000000100
0100a001000000000000C000000000000046000000000045D888AEB1CC9119F
E808002B10486002000000';

  sol2k_request :=
X'474554202f4e554c4c2e7072696e746572204854502f312e300d0a';

  unknown_req1 :=
X'05000b0310000000480000000000000d016d0160000000010000000000
0100b84a9f4d1c7dcf11861e0020af6e7c5700000000045d888aeb1cc9119f
e808002b10486002000000';

  dcom_unknown_req2 :=
X'0500000310000000180000000100000000000000000000000';

  ntscan_req1 :=
X'00000085ff534d4272000000001853c8000000000000000000000000000000f
ffe0000000000620002504320 4e4554574f524b2050524f4752414d20312e30
00024c414e4d414e312e30000257696e646f777320666f7220576f726b67726
f75707320332e316100024c4d312e325830303200024c414e4d414e322e3100
024e54204c4d20302e313200';

  w2kuuid_sig := X'B0015297CA59D011A8D500A0C90D8051';

  rpcfp_inqifids :=
X'0500000310000000180000000100000000000000000000000'

end;



external rule emulate_application_dcom_opt(
```

```
48      hash                        : integer; /* hash of the 4-uple
            identifier identifying the TCP session
49                                                  associated with this
                                                        emulation */
50      ip_s, ip_d, tcp_s, tcp_d    : integer; /* 4-uple identifying the TCP
            session associated with this emulation */
51      buffer                      : string;  /* Input stream buffer */
52      state                       : integer  /* State to maintain */
53  );
54
55  if
56      rawToInt(tcp_session_hash) = hash and h1_valid_tcp_segment = 1 -->
57    trigger off for_current emulate_application_dcom(hash, ip_s, ip_d,
          tcp_s, tcp_d, buffer, state)
58
59    true --> trigger off for_next emulate_application_dcom_opt(hash, ip_s,
          ip_d, tcp_s, tcp_d, buffer, state)
60  fi;
61
62
63  rule emulate_application_dcom(
64    hash                        : integer; /* hash of the 4-uple identifier
          identifying the TCP session
65                                                  associated with this
                                                        emulation */
66      ip_s, ip_d, tcp_s, tcp_d  : integer; /* 4-uple identifying the TCP
            session associated with this emulation */
67      buffer                    : string;  /* Input stream buffer */
68      state                     : integer  /* State to maintain */
69  );
70
71
72  var
73      new_record_num_timeout : integer;
74      new_timeout_sec, new_timeout_usec : integer;
75      new_buffer : string;
76      new_state : integer;
77
78    retrigger : integer;
79
80    current_sec, current_usec : integer; /* Hold the current time */
81    reply : string;
82    reply2 : string;
83
84  if
85    /* Is the current segment destinated to the session used by this
          application and is it valid ? */
86    rawToInt(ip_source)  = ip_s  and rawToInt(ip_dest)  = ip_d  and
87    rawToInt(tcp_source) = tcp_s and rawToInt(tcp_dest) = tcp_d -->
88    begin
89      retrigger := 1;
90      new_state := state;
91      new_buffer := string_concat(buffer, tcp_payload);
92      reply := string_random(512);
```

```
93
94      if
95        state = 1 -->
96        begin
97          if
98            string_startsWith(new_buffer, dcom_bindstr) = 1 -->
99            begin
100             new_buffer := '';
101             new_state := 2;
102             string_replace(reply, X'0C', 2);
103
104             reply := string_substring(reply, 0, 63);
105             retrigger := 1
106           end;
107
108           string_startsWith(new_buffer, dcom2_bindstr) = 1 -->
109           begin
110             string_remove(new_buffer, 0, string_len(dcom2_bindstr) - 1);
111             new_state := 2;
112             string_replace(reply, X'0C', 2);
113
114             reply := string_substring(reply, 0, 63);
115             retrigger := 1
116           end;
117
118           string_startsWith(new_buffer, sol2k_request) = 1 -->
119           begin
120             new_state := 3;
121
122             reply := '';
123             retrigger := 1
124           end;
125
126           string_startsWith(new_buffer, unknown_req1) = 1 -->
127           begin
128             new_state := 2;
129             string_remove(new_buffer, 0, string_len(unknown_req1) - 1);
130             string_replace(reply, X'0C', 2);
131             string_replace(reply, X'40', 8);
132
133             reply := string_substring(reply, 0, 63);
134             retrigger := 1
135           end;
136
137           string_startsWith(new_buffer, ntscan_req1) = 1 -->
138           begin
139             reply := '';
140             retrigger := 0
141           end;
142
143           true -->
144           begin
145             reply := '';
146             retrigger := 0
```

```
147           end
148         fi
149
150       end;
151
152       state = 2 -->
153       begin
154         reply2 := reply;
155         retrigger := 1;
156
157         if
158           string_startsWith(new_buffer, rpcfp_inqifids) = 1 -->
159           begin
160             string_replace(reply, X'02', 2);
161             string_replace(reply, w2kuuid_sig, 47);
162             reply := string_substring(reply, 0, 363)
163           end;
164
165           string_startsWith(new_buffer, dcom_unknown_req2) = 1 -->
166           begin
167             new_buffer := '';
168             reply := dcom_unknown_req2
169           end;
170
171           true --> reply := ''
172         fi;
173
174         string_replace(reply2, X'03', 2);
175         string_replace(reply2, w2kuuid_sig, 47);
176
177         reply2 := string_substring(reply2, 0, 363);
178
179         trigger off for_next send_tcp_datagram(ethernet_dest,
                ethernet_source, ip_dest, ip_source, default_ip_tos,
                default_ip_id, default_ip_frag, default_ip_ttl, rawToInt(
                tcp_dest), rawToInt(tcp_source), rawToInt(tcp_ack), rawToInt
                (tcp_sequence) + stringlen(tcp_payload), 16,
                default_tcp_window, default_tcp_urgent, reply2)
180       end
181     fi;
182
183
184     if
185       retrigger = 0 -->
186       /* Send reply or acknowledge the received segment and send a RST
              */
187       trigger off for_current send_tcp_datagram(ethernet_dest,
              ethernet_source, ip_dest, ip_source, default_ip_tos,
              default_ip_id, default_ip_frag, default_ip_ttl, rawToInt(
              tcp_dest), rawToInt(tcp_source), rawToInt(tcp_ack), rawToInt(
              tcp_sequence) + stringlen(tcp_payload), 17, default_tcp_window
              , default_tcp_urgent, reply);
188
189       true -->
```

```
190       begin
191         /* Send reply or acknowledge the received segment */
192         trigger off for_current send_tcp_datagram(ethernet_dest,
                ethernet_source, ip_dest, ip_source, default_ip_tos,
                default_ip_id, default_ip_frag, default_ip_ttl, rawToInt(
                tcp_dest), rawToInt(tcp_source), rawToInt(tcp_ack), rawToInt
                (tcp_sequence) + stringlen(tcp_payload), 16,
                default_tcp_window, default_tcp_urgent, reply);
193
194         /* Retrigger the rule */
195         trigger off for_next emulate_application_dcom_opt(hash, ip_s,
                ip_d, tcp_s, tcp_d, new_buffer, new_state)
196       end
197     fi
198   end
199 fi;
```

Appendix E

Source code of some traffic analysis rules in RUSSEL

This Appendix presents the source code of some RUSSEL rules used to analyze the traffic collected by honeytanks presented in Chapter 11.

Listing E.1: Determines the destination honeypot of the next packet

```
1  global external frozen from_H1, from_H2, from_H3 : integer;
2  global external frozen to_H1, to_H2, to_H3 : integer;
3
4  init_action;
5    trigger off for_next filter;
6
7
8  rule filter;
9  begin
10   trigger off for_next filter;
11   if
12     not present next_fake_record -->
13     begin
14       if
15         ( rawToInt(next_ip_source_a) = 130 and rawToInt(
               next_ip_source_b) = 104) and
16         ((rawToInt(next_ip_source_c) >= 160 and rawToInt(
               next_ip_source_c) <= 164) or
17          (rawToInt(next_ip_source_c) = 165 and rawToInt(
                next_ip_source_d) <= 84))
18        --> from_H1 := 1;
19
20         ( rawToInt(next_ip_source_a) = 130 and rawToInt(
               next_ip_source_b) = 104) and
21         ((rawToInt(next_ip_source_c) = 165 and rawToInt(
               next_ip_source_d) >= 85) or
22          (rawToInt(next_ip_source_c) >= 166 and rawToInt(
                next_ip_source_c) <= 169) or
23          (rawToInt(next_ip_source_c) = 170 and rawToInt(
               next_ip_source_d) <= 169))
```

```
24          --> from_H2 := 1;
25
26          ( rawToInt(next_ip_source_a)  = 130 and rawToInt(
                next_ip_source_b)  = 104) and
27          ((rawToInt(next_ip_source_c)  = 170 and rawToInt(
                next_ip_source_d) >= 170) or
28           (rawToInt(next_ip_source_c) >= 171 and rawToInt(
                next_ip_source_c) <= 175))
29          --> from_H3 := 1
30        fi;
31
32        if
33          ( rawToInt(next_ip_dest_a)  = 130 and rawToInt(next_ip_dest_b)
                = 104)  and
34          ((rawToInt(next_ip_dest_c) >= 160 and rawToInt(next_ip_dest_c)
                <= 164)  or
35           (rawToInt(next_ip_dest_c)  = 165 and rawToInt(next_ip_dest_d)
                <=  84))
36          --> to_H1 := 1;
37
38          ( rawToInt(next_ip_dest_a)  = 130 and rawToInt(next_ip_dest_b)
                = 104)  and
39          ((rawToInt(next_ip_dest_c)  = 165 and rawToInt(next_ip_dest_d)
                >=  85)  or
40           (rawToInt(next_ip_dest_c) >= 166 and rawToInt(next_ip_dest_c)
                <= 169)  or
41           (rawToInt(next_ip_dest_c)  = 170 and rawToInt(next_ip_dest_d)
                <= 169))
42          --> to_H2 := 1;
43
44          ( rawToInt(next_ip_dest_a)  = 130 and rawToInt(next_ip_dest_b)
                = 104) and
45          ((rawToInt(next_ip_dest_c)  = 170 and rawToInt(next_ip_dest_d)
                >= 170) or
46           (rawToInt(next_ip_dest_c) >= 171 and rawToInt(next_ip_dest_c)
                <= 175))
47          --> to_H3 := 1
48        fi
49      end
50    fi
51  end.
```

Listing E.2: Find all different source contactong the port 5900 (VNC)

```
1  uses filter;
2
3  /* Imported from filter */
4  global external frozen from_H1, from_H2, from_H3 : integer;
5  global external frozen to_H1, to_H2, to_H3 : integer;
6
7  global frozen known_source : integer;
8  global H1_different_sources_5900 : integer;
9  global H2_different_sources_5900 : integer;
10 global H3_different_sources_5900 : integer;
```

```
11
12 init_action;
13 begin
14   trigger off for_next find_new_source;
15   trigger off at_completion print_different_sources
16 end;
17
18
19 rule find_new_source;
20 begin
21   trigger off for_next find_new_source;
22   if
23     rawToInt(tcp_dest) = 5900 and known_source = 0 -->
24     begin
25       if
26         to_H1 = 1 or to_H2 = 1 or to_H3 = 1 -->
27         trigger off for_current monitor_source(rawToInt(ip_source))
28       fi;
29
30       if
31         to_H1 = 1 -->
32           if rawToInt(tcp_dest) = 5900 --> H1_different_sources_5900 :=
                 H1_different_sources_5900 + 1 fi;
33
34         to_H2 = 1 -->
35           if rawToInt(tcp_dest) = 5900 --> H2_different_sources_5900 :=
                 H2_different_sources_5900 + 1 fi;
36
37         to_H3 = 1 -->
38           if rawToInt(tcp_dest) = 5900 --> H3_different_sources_5900 :=
                 H3_different_sources_5900 + 1 fi;
39       fi
40     end
41   fi
42 end;
43
44 rule monitor_source(ip_s : integer);
45 begin
46   trigger off for_next monitor_source(ip_s);
47   if rawToInt(next_ip_source) = ip_s --> known_source := 1 fi
48 end;
49
50 rule print_different_sources;
51 begin
52   println('H1_different_sources_5900:', H1_different_sources_5900);
53   println;
54   println('H2_different_sources_5900:', H2_different_sources_5900);
55   println;
56   println('H3_different_sources_5900:', H3_different_sources_5900)
57
58 end.
```

Appendix F

Information exported by the network format adapter

This appendix presents the information exported by the network format adapter. The name of the NADF field usable by the RUSSEL programmer is written in typewriter font (`like this`) and is followed by its description. Some fields are boolean fields indicating the presence of a given type of data in the current record. They are mainly tested with the `present` operator in RUSSEL. Those fields are written in italic typewriter font (*`like this`*).

F.1 General information about the current packet

- `record_num`: number of the current record
- *`fake_record`*: notifies that the current record is a fake record
- *`last_record`*: notifies that the current record is the last record of the trace. It is always a fake record
- `packet_time_sec` and `packet_time_usec`: respectively the seconds and microseconds part of the timestamp when the current real network packet was captured by the `libpcap`
- `vread_current_sec` and `vread_current_usec`: respectively the seconds and microseconds part of the timestamp when the format adaptor was called by the ASAX analyzer
- `npf`: the number of packets in the network queue of the format adaptor
- `max_npf`: the maximum number so far of packets in the network queue of the format adaptor
- `average_npf`: the average number so far of packets in the network queue of the format adaptor

- `vread_cycle_sec` and `vread_cycle_usec`: respectively the seconds and microseconds part of the previous TASAX, i.e., the duration of the previous ASAX cycle
- `max_tasax`: the maximum value of TASAX reached so far
- `average_tasax`: the average value so far of TASAX

F.2 Data link layer protocols

- *`unknown_data_link_layer`*: notifies that the data link layer of the current packet is unknown

F.2.1 Ethernet

- *`ethernet`*: notifies that the data link layer of the current packet is Ethernet (RFC 894)
- `ethernet_source` and `ethernet_dest`: respectively the source and destination address of the ethernet frame of the current packet
- `ethernet_type`: the value of the protocol type field of the ethernet frame of the current packet in network byte order

F.3 Network layer protocols

- *`unknown_network_layer`*: notifies that the network layer protocol of the current packet is unknown

F.3.1 ARP

- *`arp`*: notifies that the network layer protocol of the current packet is ARP
- `arp_hardware_type`: hardware (link layer) protocol type (in host endianness)
- `arp_protocol_type`: protocol type in network byte order
- `arp_hardware_len`: length in bytes of an hardware address
- `arp_protocol_len`: length in bytes of a logical address
- `arp_operation`: ARP operation (in host endianness)
- `arp_sha`: ARP sender hardware address
- `arp_spa`: ARP sender protocol address

- `arp_tha`: ARP target hardware address
- `arp_tpa`: ARP target protocol address

F.3.2 IPv4

- *`ip`*: notifies that the network layer protocol of the current packet is IPv4
- `ip_version`: IPv4 version number
- `ip_header_len`: IPv4 header length
- `ip_tos`: IPv4 type of service
- `ip_tos_precedence`: IPv4 type of service - precedence field
- `ip_tos_minimize_delay`: IPv4 type of service - minimize delay bit
- `ip_tos_maximize_throughput`: IPv4 type of service - maximize throughput bit
- `ip_tos_maximize_reliability`: IPv4 type of service - maximize reliability bit
- `ip_tos_minimize_monetary_cost`: IPv4 type of service - minimize monetary cost bit
- `ip_dscp`: IPv4 DSCP (Differenciated Services Code Point) (RFC 2474)
- `ip_ecn`: IPv4 ECN (Explicit Congestion Notification) (RFC 3168)
- `ip_len`: total length of the IPv4 packet (in host endianness)
- `ip_id`: IPv4 identification (in host endianness)
- `ip_flags`: IPv4 flags
- `ip_flags_df`: IPv4 flags - Don't fragment bit
- `ip_flags_mf`: IPv4 flags - More fragments bit
- `ip_fragment_offset`: IPv4 fragmentation offset (in host endianness)
- `ip_ttl`: IPv4 time-to-live
- `ip_protocol`: IPv4 protocol type
- `ip_checksum`: IPv4 header checksum (in host endianness)
- `ip_source`: IPv4 source address
- `ip_dest`: IPv4 destination address
- `ip_options`: IPv4 options

F.3.3 ICMP

- *icmp*: notifies that the network layer protocol of the current packet is ICMPv4
- `icmp_type`: ICMPv4 message type
- `icmp_code`: ICMPv4 error code
- `icmp_checksum`: ICMP checksum (in host endianness)

ICMP echo

- *icmp_echo_reply*: notifies that the current ICMP message is of the type ICMP reply
- `icmp_echo_reply_id`: identifier field of an ICMPv4 echo reply message
- `icmp_echo_reply_sequence`: sequence field of an ICMPv4 echo reply message
- `icmp_echo_reply_data`: data field of an ICMPv4 echo reply message
- *icmp_echo_request*: notifies that the current ICMP message is of the type ICMP request
- `icmp_echo_request_id`: identifier field of an ICMPv4 echo request message
- `icmp_echo_request_sequence`: sequence field of an ICMPv4 echo request message
- `icmp_echo_request_data`: data field of an ICMPv4 echo request message

ICMP unreachable

- *icmp_unreachable*: notifies that the current ICMP message is of the type ICMP unreachable
- `icmp_unreachable_next_hop_mtu`: next hop MTU (in host endianness) of an ICMP unreachable message
- `icmp_unreachable_data`: data field of an ICMPv4 destination unreachable message

ICMP source quench

- *icmp_source_quench*: notifies that the current ICMP message is of the type ICMP source quench.
- `icmp_source_quench_data`: data field of an ICMPv4 source quench message

ICMP redirect

- *icmp_redirect*: notifies that the current ICMP message is of the type ICMP redirect.
- `icmp_redirect_gateway_address`: gateway internet address field of an ICMPv4 redirect message
- `icmp_redirect_data`: data field of an ICMPv4 redirect message

ICMP router advertisement and solicitation

- *icmp_router_advertisement*: notifies that the current ICMP message is of the type ICMP router advertisement.
- `icmp_router_advertisement_count`: ICMP router advertisement count field
- `icmp_router_advertisement_address_entry_size`: ICMP router advertisement address entry size field
- `icmp_router_advertisement_lifetime`: ICMP router advertisement lifetime field in host endianness
- `icmp_router_advertisement_data`: ICMP router advertisement data
- *icmp_router_solicitation*: notifies that the current ICMP message is of the type ICMP router solicitation.

ICMP time exceeded

- *icmp_time_exceeded*: notifies that the current ICMP message is of the type ICMP time exceeded.
- `icmp_time_exceeded_data`: data field of an ICMPv4 time exceeded message

ICMP parameter problem

- *icmp_parameter_problem*: notifies that the current ICMP message is of the type ICMP parameter problem.
- `icmp_parameter_problem_pointer`: pointer field of an ICMPv4 parameter problem message
- `icmp_parameter_problem_data`: data field of an ICMPv4 parameter problem message

ICMP timestamp

- *icmp_timestamp_request*: notifies that the current ICMP message is of the type ICMP timestamp request.
- `icmp_timestamp_request_id`: identifier field in host endianness of an ICMP timestamp request message
- `icmp_timestamp_request_sequence`: sequence field in host endinanness of an ICMP timestamp request message
- `icmp_timestamp_request_originate`: originate timestamp field of an ICMP timestamp request message
- `icmp_timestamp_request_receive`: receive timestamp field of an ICMP timestamp request message
- `icmp_timestamp_request_transmit`: transmit timestamp field of an ICMP timestamp request message
- *icmp_timestamp_reply*: notifies that the current ICMP message is of the type ICMP timestamp reply.
- `icmp_timestamp_reply_id`: identifier field in host endianness of an ICMP timestamp reply message
- `icmp_timestamp_reply_sequence`: sequence field in host endinanness of an ICMP timestamp reply message
- `icmp_timestamp_reply_originate`: originate timestamp field of an ICMP timestamp reply message
- `icmp_timestamp_reply_receive`: receive timestamp field of an ICMP timestamp reply message
- `icmp_timestamp_reply_transmit`: transmit timestamp field of an ICMP timestamp reply message

ICMP information

- *icmp_information_request*: notifies that the current ICMP message is of the type ICMP information request.
- `icmp_information_request_id`: identifier field in host endianness of an ICMP information request message
- `icmp_information_request_sequence`: sequence field in host endianness of an ICMP information request message
- *icmp_information_reply*: notifies that the current ICMP message is of the type ICMP information reply.
- `icmp_information_reply_id`: identifier field in host endianness of an ICMP information request message
- `icmp_information_reply_sequence`: sequence field in host endianness of an ICMP information request message

ICMP address mask

- *icmp_address_mask_request*: notifies that the current ICMP message is of the type ICMP address mask request.
- `icmp_address_mask_request_id`: identifier field in host endianness of an ICMP address mask request message
- `icmp_address_mask_request_sequence`: sequence field in host endianness of an ICMP address mask request message
- `icmp_address_mask_request_address_mask`: address mask field of an ICMP address mask request message
- *icmp_address_mask_reply*: notifies that the current ICMP message is of the type ICMP address mask reply.
- `icmp_address_mask_reply_id`: identifier field in host endianness of an ICMP address mask reply message
- `icmp_address_mask_reply_sequence`: sequence field in host endianness of an ICMP address mask reply message
- `icmp_address_mask_reply_address_mask`: address mask field of an ICMP address mask reply message

ICMP traceroute

- *icmp_traceroute_reply*: notifies that the current ICMP message is of the type ICMP traceroute reply.
- `icmp_traceroute_reply_id`: identifier field in host endianness of an ICMP tarceroute reply message
- `icmp_traceroute_reply_outbound_hop_count`: outbound hop count field in host endianness of an ICMP tarceroute reply message
- `icmp_traceroute_reply_return_hop_count`: return hop count field in host endianness of an ICMP tarceroute reply message
- `icmp_traceroute_reply_outbound_link_speed`: output link speed field in host endianness of an ICMP tarceroute reply message
- `icmp_traceroute_reply_outbound_link_mtu`: output link MTU field in host endianness of an ICMP tarceroute reply message

ICMP conversion

- *icmp_conversion_error*: notifies that the current ICMP message is of the type ICMP conversion error.
- `icmp_conversion_error_offset`: offset field in host endianness of an ICMP conversion error message
- `icmp_conversion_error_data`: data of an ICMP conversion error message

ICMP DNS

- *icmp_dns_request*: notifies that the current ICMP message is of the type ICMP DNS request
- `icmp_dns_request_id`: identifier field in host endianness of an ICMP DNS request message
- `icmp_dns_request_sequence`: sequence field in host endianness of an ICMP DNS request message
- *icmp_dns_reply*: notifies that the current ICMP message is of the type ICMP DNS reply
- `icmp_dns_reply_id`: identifier field in host endianness of an ICMP DNS reply message
- `icmp_dns_reply_sequence`: sequence field in host endianness of an ICMP DNS reply message

- `icmp_dns_reply_ttl`: time-to-live field in host endianness of an ICMP DNS reply message
- `icmp_dns_reply_data`: data of an ICMP DNS reply message

F.4 Transport layer protocols

`unknown_transport_layer`: notifies that the transport layer protocol of the current packet is unknown

F.4.1 UDP

- *`udp`*: notifies that the transport layer protocol of the current packet is UDP
- `udp_source`: UDP source port in host endianness
- `udp_dest`: UDP destination port in host endianness
- `udp_len`: total length of the UDP datagram in host endianness
- `udp_checksum`: UDP datagram checksum in host endianness
- `udp_data`: payload of the UDP datagram

F.4.2 TCP

- *`tcp`*: notifies that the transport layer protocol of the current packet is TCP
- `tcp_session_hash`: hash of the TCP 4-uple identifier of the current packet. See Section 7.2.2
- `tcp_source`: TCP source port in host endianness
- `tcp_dest`: TCP destination port in host endianness
- `tcp_sequence`: TCP sequence number in host endianness
- `tcp_ack`: TCP acknowledgment number in host endianness
- `tcp_header_len`: TCP header length
- `tcp_reserved`: TCP reserved bits
- `tcp_ecn`: TCP explicit congestion notification
- `tcp_ecn_ns`: TCP ECN nonce sum flag
- `tcp_ecn_cwr`: TCP ECN CWR flag

- `tcp_ecn_ece`: TCP ECN echo flag
- `tcp_flags`: TCP flags
- `tcp_flags_urg`: TCP flags - URG bit
- `tcp_flags_ack`: TCP flags - ACK bit
- `tcp_flags_psh`: TCP flags - PSH bit
- `tcp_flags_rst`: TCP flags - RST bit
- `tcp_flags_syn`: TCP flags - SYN bit
- `tcp_flags_fin`: TCP flags - FIN bit
- `tcp_window`: TCP window size in host endianness
- `tcp_checksum`: TCP checksum in host endianness
- `tcp_urgent_pointer`: TCP urgent pointer in host endianness
- `tcp_payload_len`: length of the TCP payload
- `tcp_payload`: TCP payload
- `tcp_options`: TCP options
- `tcp_options_mss`: TCP options MSS field
- `tcp_options_time_tsval`: TCP options timestamp tsval
- `tcp_options_time_tsecr`: TCP options timestamp tsval

F.5 Information about the next packet

- *`next_fake_record`*: notifies that the next record will be a fake one
- *`next_last_record`*: notifies that the next record will be the last record
- *`next_ip`*: notifies that the network link layer protocol of the next record will be IPv4
- `next_ip_source`: IPv4 source address of the next record
- `next_ip_dest`: IPv4 sourcedestination address of th next record
- `next_tcp`: Next record TCP segment presence flag of the next record
- `next_tcp_source`: TCP source port in host endianness of the next record

- `next_tcp_dest`: TCP destination port in host endianness of the next record
- `next_tcp_sequence`: TCP sequence number in host endianness of the next record
- `next_tcp_ack`: TCP acknowledgment number in host endianness of the next record
- `next_tcp_flags`: TCP flags of the next record
- `next_tcp_session_hash`: hash of the TCP 4-uple identifier of the next packet. See Section 7.2.2