Creación de un IDS Hardware

Trabajo de Fin de Grado en Ingeniería Informática

Universidad Complutense de Madrid

Facultad de Informática

Curso académico 2018/2019

Autores: Félix Villar González y Jorge González Soria

Directores de proyecto: Marcos Sánchez-Élez Martín y Inmaculada Pardines Lence

Índice

Índice de figuras y tablas

Resumen

Internet aporta muchas ventajas y facilidades a usuarios y a empresas, en contraposición es una gran fuente de peligros siendo la principal entrada de ataques a una red. Esto ha supuesto que las empresas inviertan recursos en establecer y aplicar unas medidas y políticas de seguridad con el fin de protegerse de estos ataques. Una medida de seguridad puede ser el uso de un cortafuegos que controla y analiza toda la información que llega del exterior. Este dispositivo actúa, por lo tanto, como una primera línea de defensa. Como segunda línea de defensa se suele usar un Sistema de Detección de Intrusos (IDS). Su objetivo es la detección eficaz y rápida de posibles ataques para poder tratarlos lo antes posible.

En este proyecto se propone la creación de un IDS hardware, pues conviene migrar parte del trabajo software a hardware para que estos dispositivos dispongan de un sistema operativo más reducido que sea menos susceptible de presentar vulnerabilidades. La implementación de este sistema se realizará sobre una FPGA y se basa en la traducción de reglas a expresiones regulares.

Palabras clave: Sistema de detección de intrusos, IDS hardware, FPGA, Expresiones regulares, Sistema operativo, Vulnerabilidades.

Abstract

Internet provides several advantages and facilities to users and companies, in contrast it is a vast source of danger being the main entrance for any web attack. This has supposed that companies invest resources establishing and applying security measures and policies to protect themselves from these attacks. One of those security measures is often a firewall which controls and analyzes all the information that comes from outside. This device works as a first defense line. As a second defense line is usually used an Intrusion Detection System (IDS). Its objective is the fast and efficient detection of possible attacks for treating them as soon as possible.

In this project a hardware IDS is presented, because it is convenient to migrate part of the software work to hardware so these devices have a more reduced operative system that is less sensitive to present vulnerabilities. This system's implementation is going to be made over an FPGA and based on the rule to regular expression translation.

Keywords: Intrusion Detection System, hardware IDS, FPGA, Regular expression, Operative system, Vulnerabilities

Capítulo 1

Introducción

En este capítulo se exponen las motivaciones y objetivos que nos han llevado a desarrollar este proyecto, así como el plan de trabajo y la organización de esta memoria.

1.1 Motivación

La incesante conexión de nuevos dispositivos IoT a Internet, propiciando con ello la distribución de código dañino, constituye un elemento facilitador de los ciberataques [1]. Esto ha hecho que tratar este problema sea una gran preocupación para las empresas [2]. Estas invierten gran parte de su presupuesto en realizar un análisis de riesgos para definir métodos y medidas de seguridad con el fin de proteger la información que poseen; además de cumplir la legalidad vigente (Reglamento General de Protección de Datos (RGPD)).

Desde los primeros ciberataques en la década de los 90 las empresas son conscientes de la importancia de mantener activo el servicio que ofrecen a sus usuarios por la peligrosidad de la posible pérdida de datos, las pérdidas económicas y el daño a la imagen de la corporación.

Todo este proceso ha conducido a la aparición y desarrollo de sistemas aplicados a la ciberseguridad. Entre ellos podemos distinguir los cortafuegos, que establecen perímetros de seguridad y controlan todo el tráfico que puede pasar a la red, y los IDS, que detectan la presencia de un atacante que haya sido capaz de evadir el cortafuegos.

Los sistemas de detección de intrusos pueden ser de gran ayuda en la prevención de ataques y emisión de alertas cuando la red está bajo amenaza [2]. Tradicionalmente, las técnicas de detección de intrusos suelen agruparse en dos categorías, detección basada en firmas y detección basada en anomalías [3].

En la actualidad existen distintas investigaciones sobre cómo implementar de forma eficiente un IDS. Algunas buscan aumentar el rendimiento y la capacidad del sistema [4,5] mientras que otras giran en torno a buscar la reconfigurabilidad del sistema. Esta problemática ha sido abordada desde distintos puntos de vista [2,3,5]: desde el uso de *Machine Learning* hasta la implementación sobre múltiples *Field Programmable Gate Arrays* (FPGA).

Todas estas soluciones tienen un punto de encuentro en común: la búsqueda de un sistema de detección de intrusos reconfigurable, rápido y eficaz que sea capaz de analizar y detectar cualquier amenaza existente o desconocida dentro de la red.

Lo que se propone en este Trabajo de Fin de Grado es la implementación hardware de un IDS tipo NIDS basado en reglas. La principal ventaja de este sistema es que puede utilizar un sistema operativo reducido lo que lo convierte en un sistema más robusto [5, 6]. Además, esta implementación permite comparar las reglas en paralelo y aumentar la eficiencia del sistema.

La implementación hardware se va a realizar sobre una FPGA usando como punto de partida el Trabajo de Fin de Máster: Evaluación de Expresiones Regulares sobre Hardware Reconfigurable (Ignacio Martín Santamaría, 2010) [6] y el Trabajo de Fin de Grado que amplió a este: Mejora de la Evaluación de Expresiones Regulares Sobre Hardware Reconfigurable (Claudio Alejandro Muñoz Fernández, 2011) [7]. Todo ello basado en el trabajo de R. Sidhu y V. K. Prasanna [5] que nos servirá de base para la construcción de nuestro sistema.

1.2 Objetivos del trabajo

Los objetivos iniciales del proyecto son:

1. Comprender el trabajo previo y las reglas Snort.
2. Conseguir una traducción de reglas Snort a expresiones regulares.
3. Unificar el trabajo previo con nuestro desarrollo software.
4. Obtener una salida VHDL a partir de las reglas Snort.
5. Sintetizar VHDL en una FPGA para detectar tráfico de red malicioso.

Para llevar a cabo la implementación de un sistema de detección de intrusos hemos partido de una herramienta de reconocimiento de patrones desarrollada previamente [9,10]. Por tanto, en primer lugar, ha sido necesario realizar un trabajo de investigación acerca de los diferentes campos que abarca este proyecto para posteriormente comprender el funcionamiento del reconocedor de patrones, poder usarlo y validar su correcto funcionamiento.

Se tomaron como base los trabajos previos de otros autores [6,7] para comenzar con el estudio de los diferentes campos de investigación: las expresiones regulares (ER) [sección 3.1], las reglas Snort [sección 2.3.1], la notación postfija, y los conceptos necesarios sobre IDS [capítulo 2].

Con la ayuda del IDE Eclipse [8] y el uso de Java desarrollamos una herramienta de traducción de reglas Snort a expresiones regulares. Adaptando la ER a la entrada del reconocedor de patrones conseguimos completar y unificar la implementación de nuestro programa.

Finalizada la implementación del sistema se llevó a cabo la fase experimental para verificar el correcto funcionamiento de este y de las técnicas usadas para su desarrollo. Se hizo uso de Vivado [9] para implementar el sistema sobre una FPGA y para observar la simulación obtenida a partir de la salida VHDL y los *testbench*.

1.3 Plan de trabajo

Para alcanzar estos objetivos se han realizado las siguientes tareas:

1.1. Estudio e investigación de los trabajos previos de los que partimos y de varios artículos [5, 6, 7].
1.2. Estudio y comprensión de los campos de investigación que abarca este proyecto.
1.3. Distinción, identificación y división de las distintas opciones de una regla Snort en partes independientes bien diferenciadas.
1.4. Obtención de las ER de cada regla Snort.
1.5. Adaptación de la salida de nuestro programa a la entrada de la herramienta de reconocimiento de patrones [6].
1.6. Testing del sistema.
1.7. Análisis de los errores encontrados en los distintos programas.
1.8. Obtención de los ficheros VHDL para la configuración de la FPGA.
1.9. Aprendizaje básico del uso de FPGAs como Spartan3 o Virtex6.
1.10. Realización de pruebas sobre un entorno de simulación (Vivado) y sobre una FPGA.
1.11. Comprobación de los resultados obtenidos en la simulación respecto a salidas VHDL ya probadas.

Otras tareas no relacionadas con los objetivos del proyecto que hemos realizado son:

1.1. Estudio de tecnologías que permitan la ampliación de nuestro trabajo en un futuro.
1.2. Búsqueda de diversas funcionalidades para el software desarrollado.
1.3. Viabilidad del uso de este software en paralelo para incrementar el rendimiento.

El plan de trabajo está sujeto a posibles cambios. Durante el desarrollo de este proyecto ha sido necesario profundizar en los aspectos internos del software de reconocimiento de patrones. Del mismo modo se ha hecho uso de distintos recursos, como la máquina virtual Kali [10] y el programa de filtrado de paquetes Wireshark [11] para generar paquetes que permitan comprobar el correcto funcionamiento del sistema. También, de manera frecuente, se han realizado revisiones y validaciones de las partes ya finalizadas.

1.4 Organización de la memoria

Los capítulos que se enumeran a continuación siguen el orden establecido en el que se ha realizado el trabajo. Los contenidos han sido divididos de la manera más coherente en referencia al desarrollo software que se ha seguido. En cada capítulo se presenta el tema a tratar y la motivación de abordarlo.

Al ser un trabajo con gran carga de investigación se han realizado múltiples desarrollos, todos ellos plasmados en las diferentes secciones. Cabe destacar el capítulo dedicado

exclusivamente al diseño, la implementación y la arquitectura del Sistema de Detección de Intrusos debido a su complejidad y trascendencia en este proyecto.

A continuación, se describe brevemente el contenido de los siguientes capítulos:

En el **capítulo 2** se hace un estudio acerca de los sistemas de detección de intrusos existentes y sus tipos y de la importancia de proteger la red. Además, se definen conceptos recurrentes a lo largo de este documento.

En el **capítulo 3** se explica la herramienta de reconocimiento de patrones y se estudian las expresiones regulares.

En el **capítulo 4** se presenta en detalle el diseño, la implementación y la arquitectura del sistema de detección de intrusos.

En el **capítulo 5** se evalúan los resultados experimentales obtenidos y se discute acerca de posibles tecnologías, resaltando las que han sido finalmente usadas.

En el **capítulo 6** se destacan las principales conclusiones derivadas de este proyecto y se plantea el posible trabajo futuro.

Chapter 1

Introduction

In this chapter is exposed the motivations and objectives that lead us to develop this project, as well as the work plan and structure of the memory.

1.1 Motivation

The unceasing IoT devices connection to Internet, promoting dangerous code distribution constitute an enabler for cyberattacks[1]. This has produced that dealing with this problem is a major concern for companies [2]. They invest great percentage of their budget in making a risk analysis to define security methods and measures to protect the information that they possess; besides satisfy the current laws (General Data Protection Regulation(GDPR)).

Since first cyberattacks on the nineties companies are aware of the importance of keeping active the service they offer to their customers because of the danger from the possible data and economic lost and the image harm of the company.

All this process has lead to the emergence and development of systems applied to cybersecurity. Among them we can distinguish firewalls, that establish a security perimeter and control all the traffic that can flow on the net, and IDS, that detect the attacker's presence that has been able to evade the firewall.

Intrusion detection systems can be really helpful in the attack prevention and emission of alerts when the net is under threat [2]. Traditionally, intrusion detection techniques are usually grouped in two categories, signature based detection and anomaly based detection [3].

Nowadays there are different investigations about how to implement efficiently an IDS. Some try to increase system performance and capacity [4,5] while others want to achieve system reconfigurability. This issue has been approached from different points of view [2,3,5]: from using Machine Learning to the implementation over multiple Field Programmable Gate Arrays (FPGA).

All these solutions have a common meeting point: the search of a reconfigurable, fast and effective intrusion detection system that is able to analyze and detect every threat in the net.

What is proposed in this End-of-Degree Project is a hardware implementation of an NIDS type IDS based on rules. The main advantage of this system is that it could use a reduced operating system that make it more robust[5,6]. Also, this implementation allows to compare rules in parallel and increase the system's efficiency.

The hardware implementation is going to be done on a FPGA using the End-of-Master: Evaluación de Expresiones Regulares sobre Hardware Reconfigurable (Ignacio Martín Santamaría, 2010) [6] and the End-of-Degree Project that extend it:Mejora de la Evaluación de Expresiones Regulares Sobre Hardware Reconfigurable (Claudio Alejandro Muñoz Fernández, 2011) [7]. All of this is based on the work from R. Sidhu y V. K. Prasanna [5] that is going to be the base for the development of our system.

1.2 Work objectives

The initial project's objectives are:

1. Understand the previous work and the Snort rules.
2. Obtain a translation from Snort rules to regular expressions.
3. Unify the previous work with our software development.
4. Obtain a VHDL output based on the snort Rules.
5. Synthesize VHDL on a FPGA to detect malicious traffic.

To carry out the implementation of an intrusion detection system we have started from a pattern matching tool previously developed [9,10]. Therefor, it has been needed to perform an investigation work about the differents fields that our project covers for later on understand the pattern matching's functioning, be able to use it and validate its correct performance.

As a base it was taken the previous work from other authors [6,7] to start with the study of the different investigation sectors: the regular expressions (RE) [section 3.1], Snort rules [section 2.3.1], postfix notation, and the needed concepts abouts IDS [chapter 2].

With the use of Eclipse IDE [8] and Java we develop a translation tool from Snort rules to regular expressions. Adjusting the RE to the pattern matching tool's input we achieve to complete and unify the implementation of our program.

System's implementation completed, it was made an experimental phase to verify the correct running of the program and the techniques used on its development. it was used Vivado [9] to implement the system on a FPGA and to observe the simulation obtained from the VHDL output and the testbenchs.

1.3 Workplan

To reach our objectives the following task have been performed:

1. Study and research of previous works that we start from and several articles [5,6,7].
2. Study and comprehension of the investigation fields that this project covers.
3. Distinction, identification and classification of the different Snort rule options into independent, well differentiated parts.
4. Obtention of the RE for each Snort rule.
5. Adaptation of our program's output to the input of the pattern matching tool[6].

6. System testing.
7. Analysis of the errors found in the different programs.
8. Obtention of the VHDL files for the FPGA configuration.
9. Basic usage learning of FPGAs like Spartan3 or Virtex6.
10. Testing on a simulated environment (Vivado) and on a FPGA.
11. Result validation obtained from the simulation regarding VHDL outputs already tested.

Other tasks not related with the project objectives that we perform:

1. Technology study that allow the ampliation of our work in the future.
2. Search of diverse functionalities for the developed software.
3. Technical viability of use this software in parallel to increase performance.

The work plan is subject to change. During the development of this project it has been necessary to deepen in the intern aspects of the pattern machine software. Similarly has been made use of different resources, as the virtual machine Kali[10] and the package filtering program Wireshark[11] to generate packages that allow to test the correct system work. Also, frequently, we made revisions and validations of the already done parts.

1.4 Document structure

The chapters listed below follow the established order in which this work was performed. Contents have been divided in the most consistent manner in reference to the software development that has been followed. In each chapter the topic to be discussed and the motivation to work on it are presented.
Being a work with great research load, multiple developments have been made, all of them embodied in different sections. It is worth to mention the chapter dedicated exclusively to the design, implementation and architecture of the Intrusion Detection System due to its complexity and significance in this project.

The content of the following chapters is briefly described below:

In **Chapter 2** a study is made about existing intrusion detection systems and their types and the importance of protecting the network.

Chapter 3 explains the pattern recognition tool and regular expressions are studied.

Chapter 4 presents in detail the design, implementation and architecture of this intrusion detection system.

Chapter 5 evaluates the experimental results obtained and discusses possible technologies, highlighting those that have finally been used.

Chapter 6 highlights the main conclusions derived from this project and discusses possible future work.

Capítulo 2

Sistema de detección de intrusos

Para la realización de este trabajo se ha hecho una profunda investigación sobre los ciberataques y los métodos de defensa, en particular de los sistemas IDS. Las dos líneas de defensa fundamentales que existen se exponen en la sección 2.1 y en la sección 2.2 se ahonda en la primera línea de defensa, el cortafuegos. En la sección 2.3 explicamos en profundidad los IDS, el tema principal de este trabajo y en la sección 2.4 mostramos algunos ejemplos populares de estos sistemas.

2.1 Introducción

Las medidas y políticas de seguridad que se establecen para proteger a usuarios y empresas son un elemento clave para resguardarse de posibles ataques y evitar vulnerabilidades en redes y sistemas. Con ello, a lo largo de la historia se ha creado una gran variedad de métodos para proteger la información, desde el más clásico como la criptografía hasta algunos más actuales como el cortafuegos, IDS y finalmente el Sistema de Prevención de Intrusos (IPS).

Mientras que el cortafuegos está considerado la primera línea de defensa de cualquier sistema, los IDS e IPS son considerados, la segunda línea de defensa contra los ataques externos [13].

2.2 Cortafuegos

Un cortafuegos es un dispositivo de seguridad de la red que monitoriza el tráfico entrante y saliente y decide si debe permitir o bloquear un tráfico específico en función de un conjunto de restricciones de seguridad definidas [12].

Como se observa en la Figura 2.1, los cortafuegos establecen una barrera entre las redes internas seguras, controladas y fiables y las redes externas como Internet. Su principal función es denegar el acceso de una red a otra. La mayoría de las empresas usan cortafuegos para prohibir el acceso desde internet a sus redes internas. También pueden ser usados para restringir el acceso desde un segmento de la red interna a otro segmento interno de la propia red [12].

Figura 2.1: Estructura de una red con cortafuegos

Un cortafuegos puede ser un dispositivo hardware especializado, un servidor ejecutando un software o ambos en conjunto. En la actualidad los más usados son los cortafuegos software.

Un cortafuegos tiene una política de seguridad definida y granular que dicta a qué servicios está permitido acceder, qué direcciones IP y rangos están restringidos y qué puertos pueden ser accedidos. Suele instalarse en los cuellos de botella donde toda la información pueda ser inspeccionada y monitorizan los paquetes entrantes y salientes de la red que protege, descartan paquetes o los redireccionan dependiendo de su configuración [14].

Los paquetes son filtrados en base a su dirección IP origen, dirección IP destino o el número de puerto usado, el tipo de paquete y protocolo, la información de la cabecera y la secuencia de bits, entre otros. La mayor parte de las veces, las empresas utilizan un cortafuegos para generar una zona desmilitarizada (DMZ), la cual es un segmento de la red localizada entre la parte protegida y la desprotegida [12].

La DMZ normalmente contiene servidores web, mails y DNS; muchas DMZ tienen también un sistema IDS para intentar detectar comportamiento malicioso.

2.3 ¿Qué es un IDS?

Desde los años 60 se empezaron a introducir prácticas de auditoría para la inspección de datos y paquetes, pero no fue hasta 1984 cuando Dorothy Denning y Peter Neumann desarrollaron la idea del primer sistema de detección de intrusos, el prototipo IDES (Sistema Especialista de Detección de Intrusión) [15]. Este sistema se basa en la idea de que un patrón de comportamiento difiere entre un usuario normal y un intruso.

Los sistemas de detección de intrusos han evolucionado mucho con el paso del tiempo, actualmente se definen como un programa de detección de accesos no autorizados a un computador o a una red. Esta detección la realizan mediante patrones o con técnicas heurísticas [12].

Durante todo este tiempo han surgido nuevos sistemas como los Sistema de Prevención de Intrusiones (IPS) y modificaciones de sistemas ya existentes como los cortafuegos, que merecen la pena diferenciar.

En la Figura 2.2 podemos observar cómo se estructura una red con un IPS y un cortafuegos.

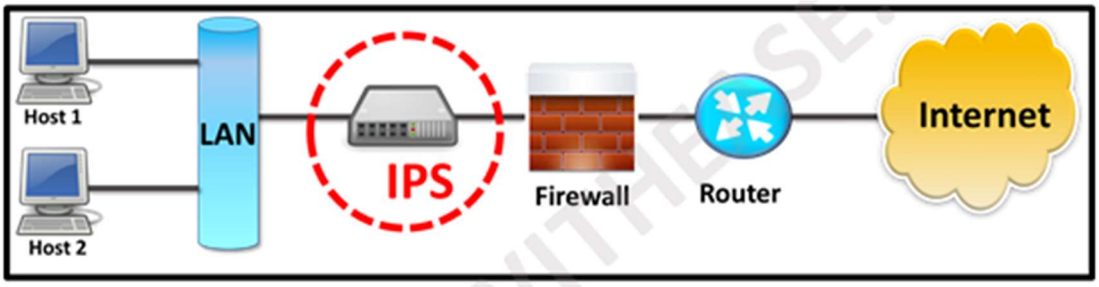


Figura 2.2: Estructura de una red con IPS

Un IDS difiere de un cortafuegos en que este último detecta amenazas y las rechaza para evitar que ocurran intrusiones, mientras que un IDS solo las detecta y genera una alarma. Un cortafuegos limita el acceso entre redes, pero no determina que un ataque pueda estar ocurriendo internamente en la red mientras que un IDS detecta ataques dentro del sistema y evalúa una intrusión cuando esta tiene lugar.

Un sistema de detección de intrusos es un sistema hardware o software que, monitorizando el sistema o la red, puede detectar actividades maliciosas como accesos no autorizados a un dispositivo o una red y generar una alerta o un reporte con los datos para que el ataque pueda ser gestionado posteriormente [12]. Podemos observar la estructura de una red con IDS en la Figura 2.3.

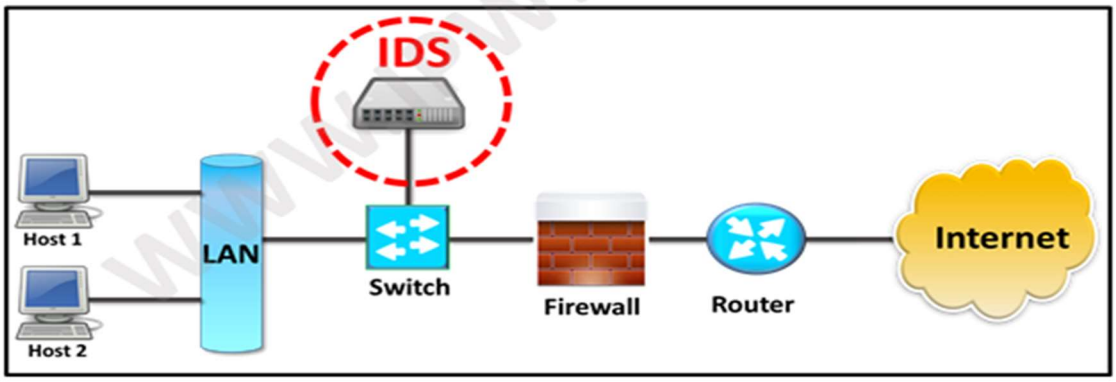


Figura 2.3: Estructura de una red con un IDS

Los IDS software están más extendidos que los IDS hardware, son viables porque detectan ataques de forma eficiente y además son flexibles y rápidos.

Respecto a los métodos de detección existen dos tipos de IDS frecuentemente usados, los basados en firmas y los basados en anomalías estadísticas [2, 12].

El problema principal de los IDS basados en firmas es su incapacidad de detectar amenazas desconocidas, ya que un IDS necesita un conocimiento base de todos los ataques llamados firmas, que serán comparados con el tráfico proveniente de la red para detectar amenazas.

Dependiendo de su alcance se diferencian dos tipos de IDS: *Host-based IDS (HIDS) y Network-IDS (NIDS)*.

Los HIDS monitorizan el tráfico únicamente en el dispositivo específico, para detectar intentos de adueñarse del mismo, en vez de realizar un análisis de todo el tráfico existente. Puede ser instalado en equipos individuales o servidores, detecta comportamientos extraños y avisa de posibles ataques. Al detectar el ataque se evitan acciones como el borrado de ficheros del sistema, la reconfiguración de opciones importantes, en general, cualquier acción que pueda poner el sistema en riesgo. Suelen ser instalados en servidores web críticos, no en todos los dispositivos de la red [12].

Los NIDS capturan todo el tráfico en la red en un punto estratégico y detectan tráfico inusual mediante sensores virtuales. Pueden ser un ordenador con el software necesario instalado o dispositivos dedicados con sus tarjetas de interfaz de red (NIC) configuradas en modo promiscuo. Las NIC con configuración estándar vigilan el tráfico *unicast* dirigido a su propio sistema, tráfico dirigido a todos los dispositivos del sistema (broadcast) y a veces tráfico *multicast*, pero si su configuración está en modo promiscuo capturan todo el tráfico de la red [12].

Podemos observar de forma esquemática las diferencias en la estructura de red entre dispositivos NIDS y HIDS en la Figura 2.4**.**

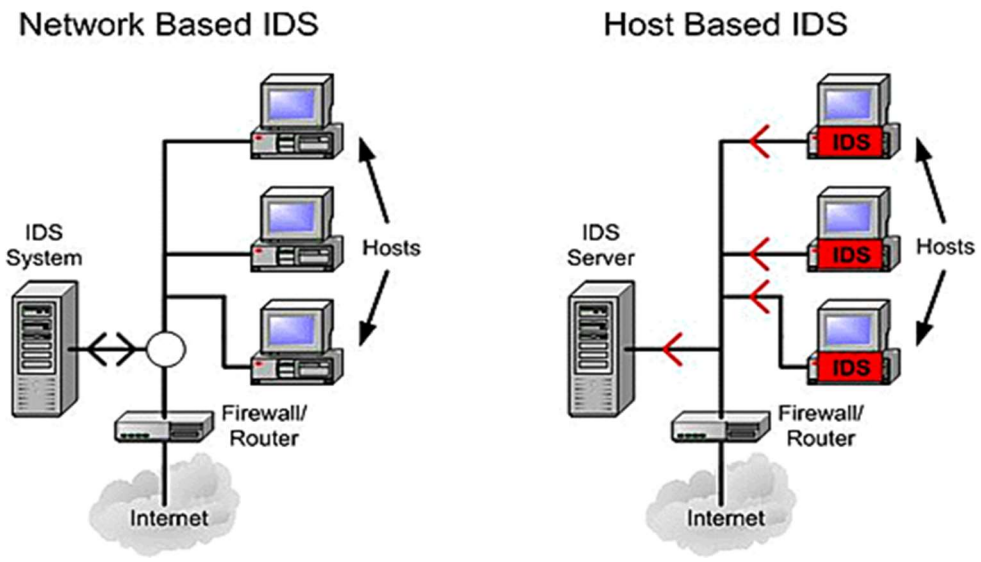


Figura 2.4: Estructura de una red con un NIDS VS HIDS

Según su funcionamiento los NIDS se pueden clasificar en *Knowledge-Based IDS, State-Based IDS, Behavior-Based IDS y Rule-Based IDS*.

Los *Knowledge-Based IDS*, también conocidos como basados en firmas, utilizan una base de datos de ataques conocidos cuyo contenido se compara con el tráfico de la red. Buscan patrones que coincidan con las firmas almacenadas en la base de datos y alertan sobre un posible ataque o modifican la configuración del cortafuegos [3]. Cualquier acción que no sea reconocida como un ataque será aceptada por este tipo de IDS.

Los *State-Based IDS* son una variante de los Knowledge-Based IDS que estudian los cambios de variables o "estados" importantes del sistema. Vigilan un flujo de actividades en vez de paquetes de red, aunque siguen basándose en firmas ya existentes en las bases de datos [12].

Los *Behavior-Based IDS* estudian el tráfico de red para aprender cómo son los paquetes estándar dentro de ella. Informan cuando una actividad o evento no coincide con el comportamiento normal de la red y por este motivo requieren mucho gasto computacional para el aprendizaje. Según el método de aprendizaje que estos IDS utilizan se pueden clasificar en tres tipos:

Los basados en anomalías estadísticas, que utilizan *machine learning* para crear perfiles de los estándares normalizados de tráfico de la red; los basados en anomalías en protocolos, que aprenden del estándar usado en protocolos concretos para encontrar ataques dentro de los paquetes del propio protocolo [12]; y finalmente, los basados en tráfico, que vigilan actividades inusuales en el tráfico de red.

Los Rule-Based IDS, son similares a los sistemas basados en firmas, utilizan un sistema experto que trabaja con reglas programadas de tipo IF/THEN. Este sistema experto permite ciertas características de inteligencia artificial cercana a los behavior-based IDS. En contraposición, la creciente complejidad de las reglas aumenta la demanda del software o hardware requerido y al ser un sistema basado en reglas no puede detectar nuevos ataques.

Todo IDS debe funcionar continuamente sin supervisión, capaz de sobrevivir a fallos, como una caída del sistema o perturbaciones, e imponiendo una sobrecarga mínima al sistema monitorizado con una tasa ínfima de falsos negativos [12]. Además, tiene que ser adaptable frente al funcionamiento de los distintos sistemas y los continuos cambios realizados al añadir nuevas funcionalidades.

Un IDS forma parte de la infraestructura de estrategia global de defensa de una organización. Puede detectar intrusiones desconocidas e imprevistas y dificulta al atacante la eliminación de sus huellas digitales [12].

En general, son independientes de los sistemas operativos consiguiendo un menor coste de implementación y mantenimiento. Tienen una alta tasa de falsas alarmas. No son un sustituto para un cortafuegos ni para una auditoría de seguridad. Puede ser necesario un reentrenamiento en aquellos IDS no basados en reglas si el comportamiento cambia con el tiempo. Durante este reentrenamiento el IDS es

vulnerable a recibir paquetes maliciosos de la red que pueda considerar y aprender cómo aceptados.

Para este proyecto vamos a desarrollar un NIDS basado en firmas (*Knowledge-Based Network-IDS)*.

2.4 Knowledge-Based Network IDS

En esta sección profundizamos en algunos de los *Knowledge-Based Network-IDS* más populares en la actualidad, Snort y Suricata, explicamos su funcionamiento, sus características y analizamos las principales diferencias entre ellos. Además, exponemos similitudes entre estos dos sistemas y otros de este mismo tipo.

2.4.1 Snort

Snort es un Knowledge-Based-Network IDS basado en preprocesadores y reglas creado por Martin Roesch bajo una licencia GPL (*open source*) [16]. Implementa un motor de detección de ataques y barrido de puertos que permite registrar, alertar y responder ante cualquier anomalía, analizando el tráfico de paquetes en tiempo real y llevando un histórico de estos ataques. En la actualidad, es el sistema de detección de intrusos más usado del planeta [17].

La principal operación que lleva a cabo Snort es examinar todo el tráfico de la red y registrar los eventos anómalos. Es el IDS más extendido en la actualidad porque, además de ser software libre, ofrece al usuario gran control sobre la configuración de las reglas [4].

Como se observa en la Figura 2.5, en la estructura de Snort se distinguen distintas partes. El decodificador toma los paquetes y los prepara para ser preprocesados o enviados al motor de detección. Los preprocesadores buscan anomalías en las cabeceras, modifican y arreglan los datos para que sean analizados por el motor de detección, el cual es el encargado de detectar si existe un intento de intrusión en los paquetes analizados utilizando el conjunto de reglas Snort para buscar patrones. Si se detecta algún intento de acceso no autorizado, se realiza la acción configurada por la regla correspondiente, generando una alerta, que será usada por los plugins de salida para almacenarla o enviarla según se haya especificado.

Todo el sistema Snort funciona gracias a sensores, dispositivos conectados a la red que se encargan de capturar los paquetes y analizarlos por si resultan ser maliciosos. Se suelen instalar en el punto de acceso a redes internas o puntos críticos de esta.

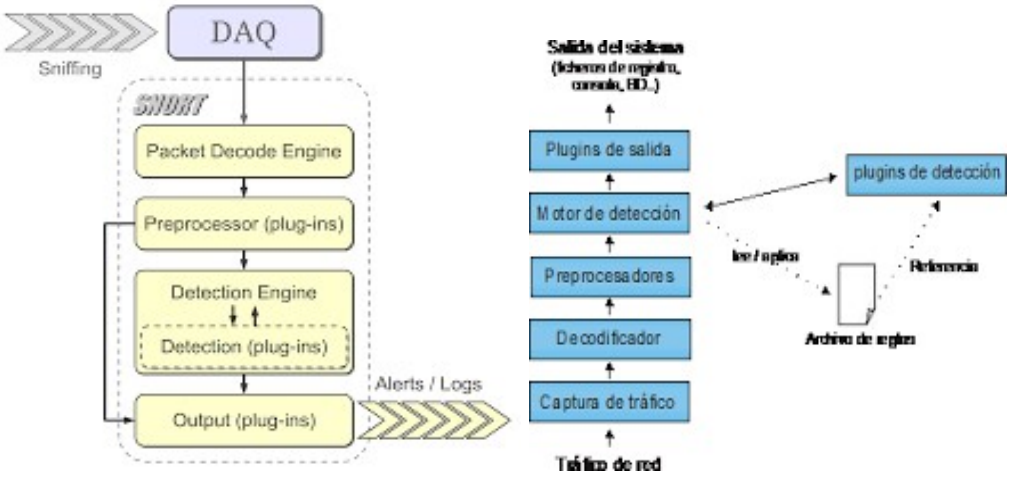


Figura 2.5: Estructura IDS Snort

Snort usa un lenguaje simple y ligero para la sintaxis de reglas. Estas reglas están divididas en dos secciones lógicas, la cabecera de la regla y las opciones. La Figura 2.6 muestra un ejemplo de regla Snort, en la que se puede apreciar las dos partes que la forman. Esta regla emite un mensaje de error cuando detecta un paquete TCP [17].

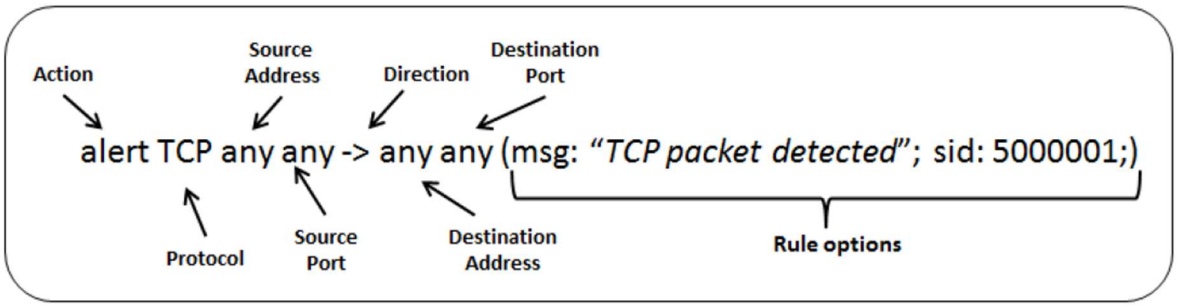


Figura 2.6: Ejemplo de regla Snort [17]

Snort puede funcionar en tres modos: modo *sniffer*, modo de registro de paquetes y modo IDS.

El modo *sniffer* monitoriza todos los paquetes de red en tiempo real, el modo de registro de paquetes almacena toda la actividad en un informe para un posterior análisis y el modo IDS controla el tráfico de la red y lo analiza frente a un conjunto de reglas definidas.

Snort es comúnmente usado para análisis de tráfico en tiempo real, detección de ataques maliciosos, escaneo de puertos, registro de paquetes y control de desbordamientos de *buffer*.

2.4.2 Suricata

Suricata es un motor de software libre desarrollado por la *Open Information Security Foundation* (OISF) que puede realizar actividades y funcionar como un IDS o un IPS [16].

Entre las características de Suricata destacan: el multihilo, el soporte GPU (CUDA), las estadísticas de rendimiento, la detección de protocolos automáticos, las comprobaciones MD5, la posibilidad de utilizar scripts en LUA, un *IP matching* rápido junto a una lista de IPs de soporte, geolocalización por IP, control de reputación por IP y distintos reportes de estadísticas y peticiones web.

Como muestra la Figura 2.7**,** Suricata tiene 4 módulos de subproceso unido al multihilo que permiten ejecutar varios de estos de forma simultánea. Estos módulos son la captura de paquetes, el decodificador, la detección y comparación de firmas y el procesamiento de evento y salidas.

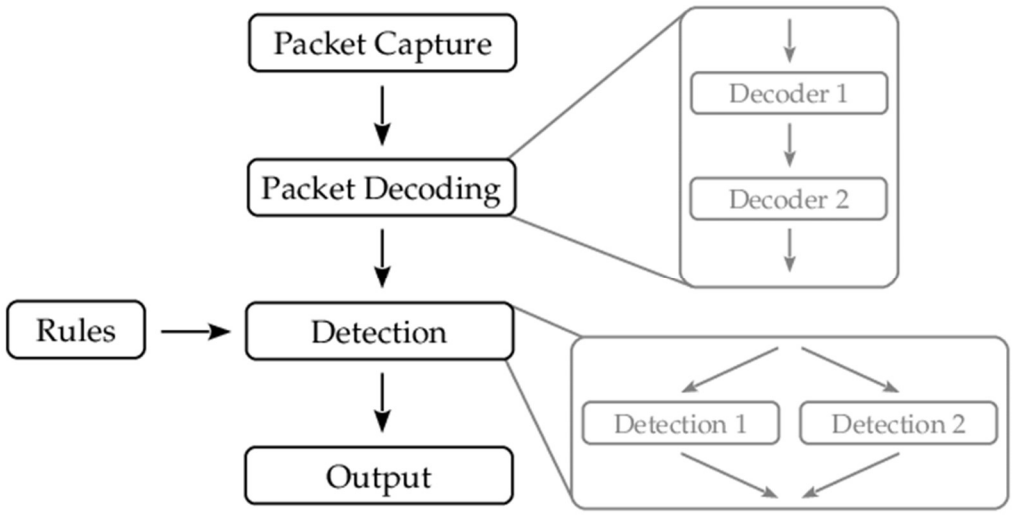


Figura 2.7: Arquitectura del IDS Suricata

2.4.3¿Características comunes de los Knowledge-Based Network IDS?

Todas las reglas usadas en Snort, Suricata y otros NIDS basados en firmas, se componen de dos partes principales, la cabecera y las opciones.

La cabecera (*Rule Header*) contiene el protocolo de paquetes a analizar, las IPs y puertos específicos a monitorizar tanto de origen como destino, el sentido de circulación de los paquetes y la acción a realizar si se encuentra un paquete coincidente con la firma. Por ejemplo: Alert tcp 192.168.1.0 any -> any 80

Las opciones (*Rule options*) contienen el mensaje que mostrará la alerta generada y la información necesaria para poder decidir si el paquete es malicioso.

Dentro de este campo se distinguen cuatro tipos:

- General o metadata: son opciones referentes a la regla, no se usan en la fase de detección, proporcionan información sobre la regla al administrador.

- *Payload*: son opciones de búsqueda de firmas (patrones) en el área de datos del paquete. Existen numerosos campos dentro de este tipo, el campo *content* es uno de los campos más importantes, busca patrones concretos en los paquetes de datos y tiene diversos modificadores, como por ejemplo, *nocase* o *within* que añaden funcionalidades a esta opción. Cabe destacar la importancia que tiene el campo *content* en nuestro proyecto.

- *Non-payload*: son opciones de búsqueda de patrones fuera del área de datos del paquete, es decir, en la cabecera, en la cola o en sus propiedades. Como ejemplo: *fragoffset, flags, ttl, id, dsize, seq, ack*.

- *Post-detection*: son opciones de ejecución tras la activación de una regla, normalmente para almacenar información sobre el paquete. Algunas de estas opciones son: *logto, session, tag, replace*.

Capítulo 3

Expresiones regulares y herramienta de reconocimiento de patrones

En este capítulo exponemos el estudio que hemos realizado sobre las expresiones regulares y el conocimiento necesario sobre la herramienta de reconocimiento de patrones que vamos a usar para el desarrollo de este trabajo. En la sección 3.1 se explicará la teoría relativa a las expresiones regulares que será de utilidad en capítulos posteriores. En la sección 3.2 se ahondará en la herramienta heredada y se explicarán sus limitaciones.

En este capítulo se definirán conceptos que aparecerán de manera recurrente en el resto del documento.

3.1 Estudio de las expresiones Regulares

Una expresión regular es un conjunto de símbolos y caracteres que describe una serie de cadenas sin enumerar sus elementos [6]. Normalmente se utilizan para describir de forma concisa un conjunto, sin tener que enumerar todos sus elementos.

Su efectividad a la hora de describir conjuntos regulares hace que sean utilizadas en varios dominios científicos, entre ellos la Informática y las Telecomunicaciones.

3.1.1 Introducción

Los orígenes de las expresiones regulares se cree que residen en la teoría de autómatas y en la teoría de lenguaje formal. Estas teorías son las encargadas del estudio de los modelos de computación y los autómatas. El matemático Stephen Cole Kleene definió modelos usando una notación matemática llamada conjuntos regulares [6].

Años después Ken Thompson usó la notación de Kleene para la implementación de un editor cuya función era encontrar cadenas o subcadenas en un texto, lo que ha provocado que en la actualidad exista el comando de búsqueda “grep”, basado en este editor. Este comando fue incorporado a Unix.

Las expresiones regulares han sido usadas a lo largo de la historia en gran variedad de entornos y para resolver innumerables problemas lo que produjo que existieran muchas notaciones distintas. Este problema fue abordado por el estándar POSIX 1003.2 del IEEE que intentó dar una solución creando una notación unificada y homogénea (REs) [6].

3.1.2 Sintaxis y ejemplos

Las expresiones regulares son una manera común de expresar patrones de emparejamiento de cadenas de caracteres (*strings*). Los elementos mínimos de una expresión regular son los caracteres a emparejar [4].

Las expresiones regulares unen estos patrones de emparejamiento de *strings* junto con operadores que nos permiten expresar concatenación, alternancia, repetición e innumerables funcionalidades dependiendo del ámbito y contexto en el que nos encontremos.

La sintaxis básica de estas expresiones regulares es:

- **El punto ".":** Símbolo que se usa como un carácter cualquiera.
- **Los paréntesis "()"**: Definen una agrupación de caracteres.
- **Los corchetes "[]":** Definen una agrupación de caracteres en la cual se busca un carácter de ese grupo. Se suele usar junto con **el guión "-"** para especificar el rango de los grupos. Por ejemplo:

  La expresión regular [A-Z1-9a-z] reconoce un patrón formado por una letra mayúscula cualquiera, un número del 1 al 9 y una letra minúscula.

- **Las llaves "{}"**: Las llaves definen cuántas veces se repite un elemento. Por ejemplo:

  La expresión regular a B {2} reconocerá dos letras B seguidas y B{2,4} reconocerá de 2 a 4 Bs.

- **El más "+":** Este símbolo indica que el conjunto que le precede tiene que ser encontrado al menos una vez. Por ejemplo:

  "(tatara)+nieto" puede ser tataranieto, una sola vez o más veces, como (tatara)(tatara)tataranieto.

- **La interrogación "?":** Este símbolo indica que el conjunto que le precede puede encontrarse una sola vez o no encontrarse. Por ejemplo:

  "(tatara)?nieto", reconoce tanto nieto como tataranieto.

- **El por "*":** Este símbolo indica que el conjunto que le precede puede aparecer ninguna, una o más veces. Por ejemplo:

  "(tatara)*nieto", reconoce tanto nieto, como tataranieto, como (tatara)(tatara)tataranieto.

- **El pipe o OR lógico "|":** Este símbolo indica la disyunción, es una OR lógica.
- **La exclamación "!":** Este símbolo indica la negación, un inversor lógico (NOT).

- **Los símbolos "^" y "$":** Marcan el inicio y fin, respectivamente, de una expresión regular, no se usan en todos los lenguajes.

Su sintaxis también soporta la notación hexadecimal usada para expresar caracteres **ASCII** [7].

En la Figura 3.1 se puede ver un ejemplo de una expresión regular. En ella se explica el uso de cada uno de los distintos signos.

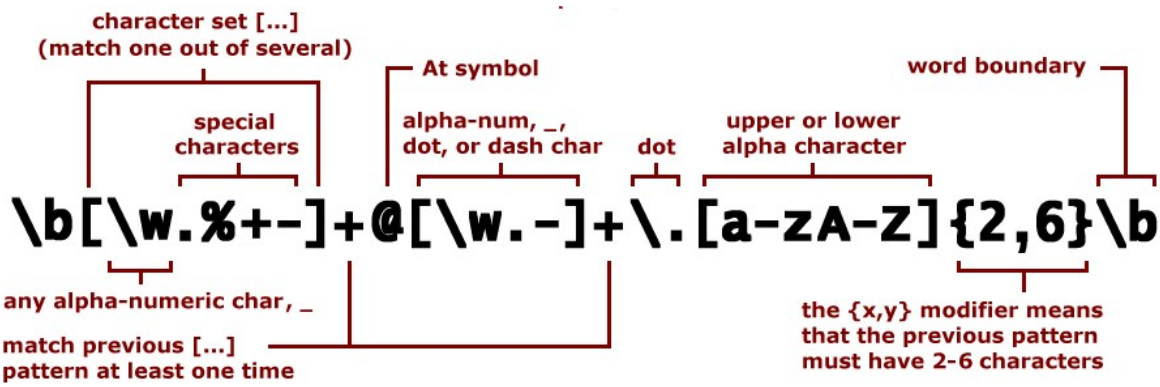


Figura 3.1: Ejemplo y explicación de una expresión regular

Se pueden consultar más ejemplos en el artículo de R. Sidhu y V.K. Prasanna [5].

3.1.3 Uso de ERs en el proyecto

En este proyecto era necesario el uso de expresiones regulares para la conversión de reglas. El uso de expresiones regulares propicia una mejora en el rendimiento del sistema, simplificando los cómputos por parte de la FPGA respecto a las reglas sin procesar.

Esta conversión favorece además el procesamiento por parte del sistema, realizando por un lado la transformación a ER y por otro el procesamiento de estas expresiones y los paquetes para que el IDS sea funcional.

La principal ventaja de usar expresiones regulares es su simplicidad y versatilidad debido a que su sintaxis permite infinidad de funciones, como hemos visto en la sección 3.1.2.

En este proyecto se ha llevado a cabo una conversión a expresión regular para poder adaptar la herramienta de reconocimiento de patrones a los requisitos del sistema que queríamos, un IDS funcional implementado sobre una FPGA.

Un inconveniente del uso de esta tecnología en el proyecto es la adaptabilidad, siendo en general una característica positiva, pero en este proyecto ha supuesto de forma negativa un gasto de recursos. La utilización de ERs ha implicado un gran esfuerzo en

la alineación del sistema reconocedor de patrones y nuestro sistema para conseguir implementar el IDS.

3.2 Antecedentes

Hemos tenido que realizar un estudio exhaustivo de la documentación previa proporcionada por los tutores. Esta se compone de un TFG [5] y un TFM [7] toda ella basada en el paper de R. Sidhu y V.K. Prasanna [6].

Esta documentación nos ha marcado la base del proyecto y una línea a seguir en las tecnologías a emplear. El proyecto se compone en términos generales de una herramienta de reconocimiento de patrones con una mejora en la evaluación de las expresiones regulares reutilizando *"engines"*, partes de hardware que evalúan una misma expresión regular (ER), aplicado sobre hardware reconfigurable, en este caso una FPGA [5].

3.2.1 Reconocedor de patrones

Esta herramienta fue desarrollada por Ignacio Martín Santamaría [7] basándose en la propuesta de implementación que expusieron por primera vez R. Sidhu y V.K. Prasanna [6].

Como se observa en la Figura 3.2, el sistema se compone de dos bloques: el generador de código que genera código VHDL a partir de las expresiones regulares que recibe, y el reconocedor de expresiones regulares que analiza el hardware descrito por el código VHDL [7].

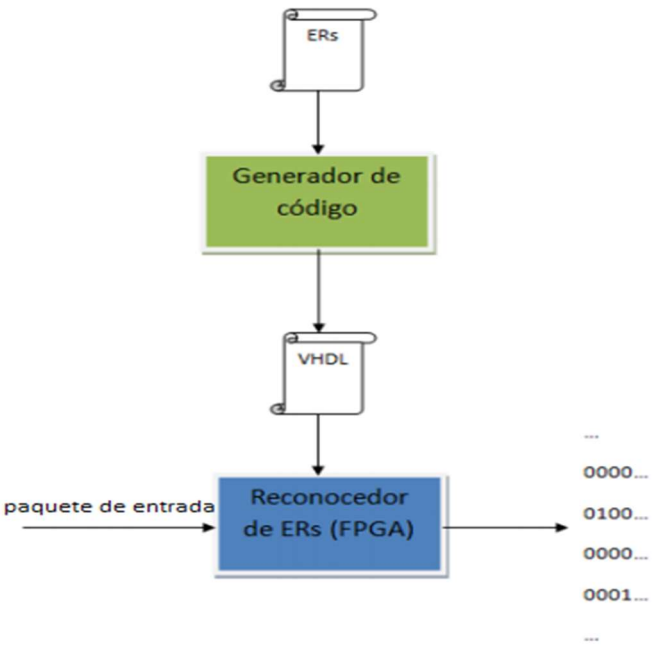


Figura 3.2: Esquema del funcionamiento de la arquitectura

Toda la estructura e implementación fue basada en [19] y [20], por lo que la lectura de estos estudios facilitó la comprensión de esta herramienta.

Todo este trabajo fue mejorado años después por Claudio Alejandro Muñoz Fernández [5], que consigue un incremento significativo de la velocidad y del porcentaje de utilización de la FPGA. Además, incorpora la notación postfija que permite maximizar el número de unidades lógicas procesadas por la placa.

3.2.2 Restricciones

Este reconocedor de patrones cuenta con una serie de limitaciones descritas en [5] basadas en la implementación y arquitectura que se siguió durante su desarrollo. Estas limitaciones determinan la mejora y reutilización de la FPGA, y el coste en tiempo que hace el sistema del reconocimiento de expresiones regulares.

Hemos considerado las siguientes limitaciones en la reutilización de expresiones regulares ya procesadas (véase la Figura 3.3):

- El preprocesamiento interno reconoce la expresión "ftp" al aplicar notación postfija como “ft·p·", esto hace que al intentar reconocer “utp” no se pueda reutilizar la subexpresión “tp” anteriormente reconocida.
- Si el sistema ya hubiera reconocido las expresiones “ftk” y “app”, procesadas con notación postfija como “ft·k·” y “ap·p·”; y quisiera reconocer la expresión “ftp” (forma postfija “ft·p·”), se podría reutilizar la cadena “ft” pero nunca la “p” pues la subexpresión “·p·” solo podrá ser reconocida tras “ap”.

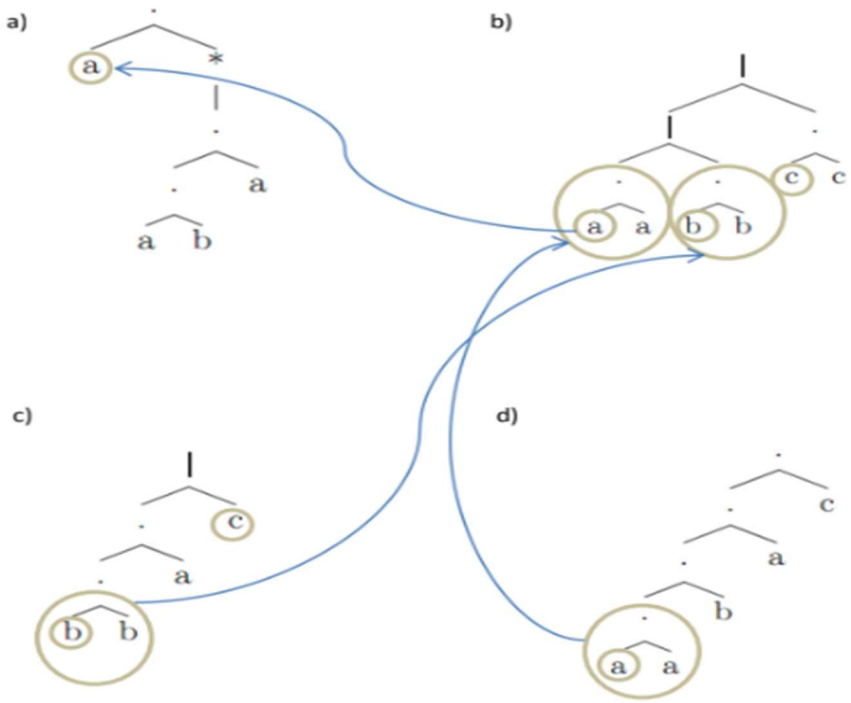


Figura 3.3: Esquema de reutilización de *engines*

- ❖ Contemplamos cualquier limitación de eficiencia derivada de las anteriormente descritas.

Al aplicar esta herramienta de reconocimiento de patrones para la implementación de un IDS hardware usando reglas Snort nos hemos encontrado con otras restricciones relacionadas con la traducción de estas reglas.

Debido a la traducción que se realiza de expresiones regulares en los ficheros VHDL [7] hay determinados caracteres usados de manera frecuente en las reglas Snort que pueden ser reconocidos por esta herramienta como caracteres, pero no con la funcionalidad de las propias expresiones regulares. Esto limita la traducción de reglas a expresiones regulares eliminando parte de su funcionalidad. Entre ellos se encuentran: ".", "?", "{}" y "!".

3.2.3 FPGA

Las expresiones regulares permiten un reconocimiento de patrones con el que implementar un nuevo sistema IDS, pero para obtener una implementación en hardware rápida y sencilla de reconfigurar fue indispensable el uso de FPGAs.

Una FPGA (Field Programmable Gate Array) es un circuito integrado que puede ser programado con código HDL para simular circuitos digitales y reproducir sus funciones. Para conseguir dicho propósito las FPGAs están formadas por distintos tipos de bloques lógicos programables entre los que se encuentra una lógica de enrutamiento, lo que permite la interconexión de estos. Adicionalmente, los bloques encargados de la entrada y la salida también pueden ser programados. Toda esta lógica combinacional de las celdas puede ser implementada físicamente en una tabla de consulta (LUT)[21].

La FPGA se define mediante un programa implementado por el usuario. Dependiendo de la FPGA el programa se guarda de manera permanente (ROM) o semipermanente (RAM). Esta capacidad de ser programable permite al usuario acceso a complejos diseños integrados evitando los costes de ingeniería asociados a estos [6].

Para generar la interconexión de bloques lógicos dentro de la FPGA se emplea un software especial, el cual traduce los diseños del usuario o el código de descripción hardware, colocando y conectando según una configuración predeterminada o personalizada por el usuario [22].

Para implementar nuestro sistema en este proyecto hemos usado una FPGA. Hemos hecho uso de un reconocedor de patrones implementado en Java que genera dinámicamente ficheros de configuración VHDL. Estos ficheros definen el comportamiento que debe realizar la FPGA, en este caso, reconocer con las reglas ya introducidas al IDS si un paquete de red es malicioso o no.

En cada ejecución del sistema configuramos la FPGA para que usando las reglas introducidas reconozca distintos patrones en diferentes paquetes. De esta manera aprovechamos las ventajas de funcionalidad que proporcionan las FPGAs, en especial, la reconfigurabilidad.

Capítulo 4

Arquitectura del sistema

En este capítulo se explican en detalle los pasos que se han seguido en el diseño y la implementación del sistema, así como las distintas partes en las que hemos dividido este sistema para su implementación y su explicación en este documento.

En la sección 4.1 se explica el diseño de la arquitectura global y de cada uno de los módulos que lo conforman y en la sección 4.2 la implementación llevada a cabo a partir del diseño anteriormente explicado.

4.1 Diseño de la arquitectura del IDS Hardware

Se presentan en esta sección los procesos de diseño de la arquitectura del sistema agrupados en distintos módulos, con el fin de facilitar la comprensión de la implementación del sistema, así como su desarrollo.

Antes de explicar cada uno de los diferentes módulos, nos vamos a centrar en la descripción del sistema global y los objetivos que queremos cumplir como se explicó en el Capítulo 1.

El diseño de la arquitectura global es de gran importancia ya que su estructura afecta a la eficiencia y al rendimiento del programa (Figura 4.1).

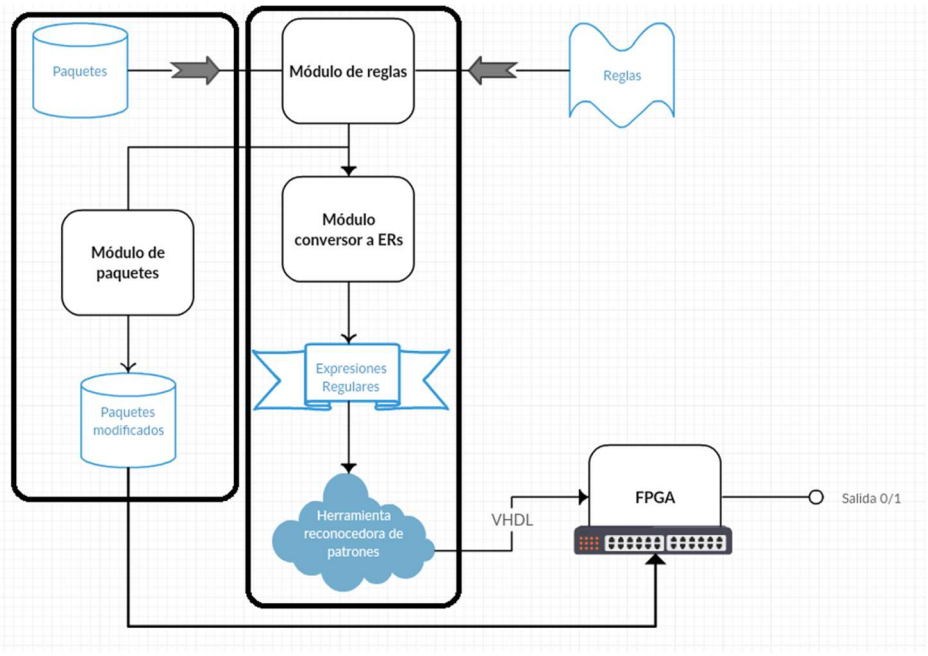


Figura 4.1: Arquitectura del IDS

El diseño se basa en una división por funcionalidad, cuya arquitectura se puede ver en la Figura 4.1. Se pueden apreciar dos partes claramente diferenciadas. La parte de los paquetes que se ejecuta siempre y la parte de las reglas que se ejecuta solo cuando queremos añadir o modificar las reglas del IDS.

Las reglas y los paquetes entran al Módulo de reglas. Por un lado, las reglas son procesadas y divididas en partes de cara a la futura transformación en expresiones regulares, y, por otro lado, los paquetes son almacenados para enviarlos al Módulo de paquetes. Aunque no sufren ninguna alteración, las reglas introducidas marcan factores como por donde cortar o en qué parte dentro del paquete se encuentra el patrón a reconocer, por ello hemos considerado importante que tanto reglas como paquetes entren a este módulo.

El Módulo de reglas produce dos salidas en paralelo, las reglas ya filtradas en dirección del Módulo conversor a ERs y los paquetes de datos seleccionados para cada regla al Módulo de paquetes.

El Módulo de paquetes recorta, modifica y adapta el paquete de datos y se lo envía como entrada a la FPGA.

El Módulo conversor a ERs devolverá un fichero de texto con todas las ERs indicando cuál pertenece a cada regla. Este fichero será analizado por la herramienta de reconocimiento de patrones que genera unos ficheros VHDL de configuración de la FPGA, o en su defecto producirá la salida para poder simular los resultados de nuestro sistema.

Por último, la FPGA producirá una salida 0 o 1, a modo de alarma, por cada uno de los ataques a detectar. En el caso de la simulación mostrará la misma salida usando la herramienta Vivado [9].

4.1.1 Diseño del Módulo de reglas

Este módulo es el encargado de la recepción de las reglas, estas pueden ser de tipo Snort o Suricata. La versión final del proyecto admite todo tipo de reglas en su completitud, desde las reglas más sencillas a las más complejas. Cualquier otro tipo de reglas que no sean las dos nombradas anteriormente, deberían ser procesadas por este módulo de la misma manera, ya que todas siguen la misma estructura (veáse sección 2.4).

Este módulo se encarga de separar cada regla en los distintos campos y opciones definidos por el programa siguiendo la sintaxis de las reglas tal y como se explicó en la sección 2.4.

Nuestro programa es capaz de aceptar como entrada una regla de forma individual o un conjunto de reglas. De forma individual recibe una cadena de caracteres mientras que, si recibe un conjunto toma las reglas de un fichero de texto, en el que cada regla está separada por ";". No existe límite numérico de reglas en el fichero de texto, el programa es capaz de leer y almacenar todas las reglas que procese.

RandomSnort.java | prueba2parecidas.txt | logRegex.txt | pruebaSnortV1.txt

```
HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS ( msg:"MALWARE-CNC Win.Trojan.KopiLuwak variant outbound request d
HOME_NET 2589 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR - Dagger_1.4.0"; flow:to_client;established; cont
$EXTERNAL_NET any -> $HOME_NET 7597 ( msg:"MALWARE-BACKDOOR QAZ Worm Client Login access"; flow:to_server;es
$EXTERNAL_NET any -> $HOME_NET 12345:12346 ( msg:"MALWARE-BACKDOOR netbus getinfo"; flow:to_server;establish
$HOME_NET 20034 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR NetBus Pro 2.0 connection established"; flow:to
$HOME_NET any -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR Infector.1.x"; flow:established;to_client; conten
$HOME_NET 666 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR SatansBackdoor.2.0.Beta"; flow:to_client;establis
$HOME_NET 6789 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR Doly 2.0 access"; flow:established;to_client; co
$EXTERNAL_NET 1000:1300 -> $HOME_NET 146 ( msg:"MALWARE-BACKDOOR Infector 1.6 Client to Server Connection Re
$HOME_NET 31785 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR HackAttack 1.20 Connect"; flow:established;to_c
$EXTERNAL_NET any -> $HOME_NET 21 ( msg:"PROTOCOL-FTP ADMw0rm ftp login attempt"; flow:to_server;established
$HOME_NET 30100:30102 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR NetSphere access"; flow:established;to_cl
$HOME_NET 6969 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR GateCrasher"; flow:established;to_client; conten
$HOME_NET 5401:5402 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR BackConstruction 2.1 Connection"; flow:esta
$EXTERNAL_NET any -> $HOME_NET 666 ( msg:"MALWARE-BACKDOOR BackConstruction 2.1 Client FTP Open Request"; fl
$HOME_NET 666 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR BackConstruction 2.1 Server FTP Open Reply"; flow
$EXTERNAL_NET 3344 -> $HOME_NET 3345 ( msg:"MALWARE-BACKDOOR Matrix 2.0 Client connect"; flow:to_server; con
$EXTERNAL_NET 3345 -> $HOME_NET 3344 ( msg:"MALWARE-BACKDOOR Matrix 2.0 Server access"; flow:to_server; cont
$HOME_NET 5714 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR WinCrash 1.0 Server Active"; flow:stateless; fla
$EXTERNAL_NET any -> $HOME_NET 79 ( msg:"MALWARE-BACKDOOR CDK"; flow:to_server;established; content:"ypi0ca"
```

Figura 4.2: Fichero de entrada de ejemplo

En este diseño hemos decidido considerar 2 tipos de reglas de entrada, las reglas tipo Protocolo y las tipo *Content*.

El tipo Protocolo se centra en el contenido de los campos de la cabecera del paquete mientras que el tipo *Content* en el contenido del campo datos.

4.1.1.1 Reglas tipo Protocolo

Dentro de este grupo podemos identificar subgrupos que se corresponden con los diferentes protocolos que admiten las reglas Snort y Suricata en la cabecera del paquete. Estos tipos son: Tipo IP, Tipo TCP, Tipo UDP y Tipo ICMP.

El tipo IP contiene todas las reglas con el campo protocolo IP y los paquetes entrantes referentes a este mismo protocolo (Figura 4.3). Cabe destacar y separar este tipo por la importancia que tienen los paquetes IP en la capa de red, así como la existencia de ataques con estas características en este protocolo.

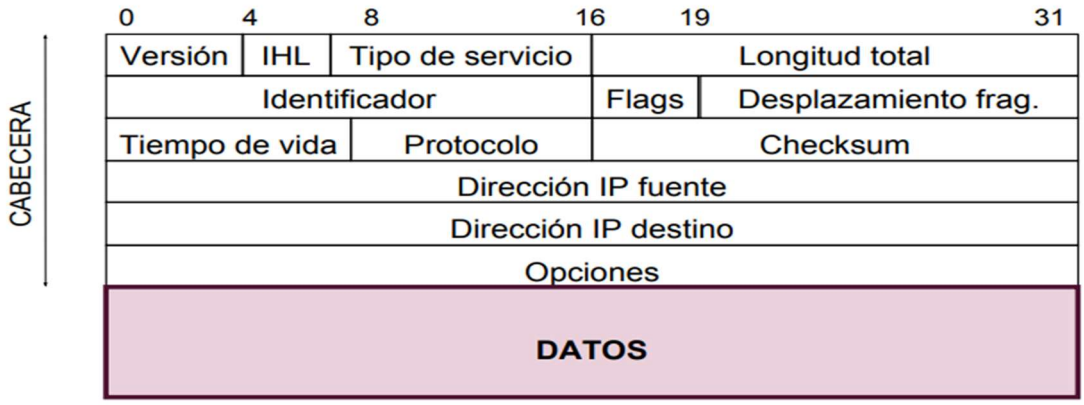


Figura 4.3: Formato de un datagrama IP

El tipo TCP incluye todas las reglas con el campo protocolo TCP. Este tipo tiene mucha importancia en nuestro programa dado que TCP es el protocolo de transporte más importante y por ello es uno de los protocolos que más ataques sufre. El paquete y datagrama son reconocibles por sus características propias (Figura 4.4).

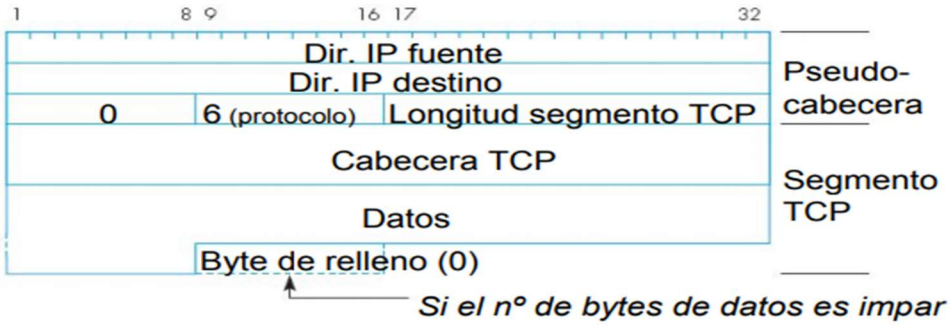


Figura 4.4: Formato de un segmento TCP

El tipo UDP contiene todas las reglas y paquetes con el campo protocolo UDP. Este protocolo es de transporte, sin conexión y sin control de errores. En la Figura 4.5 podemos observar el formato de sus paquetes.

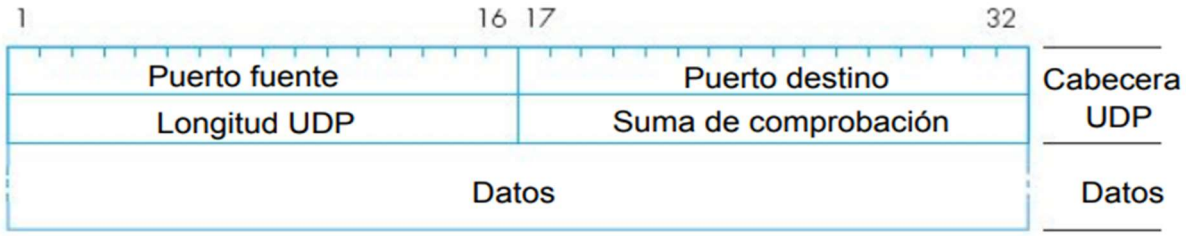


Figura 4.5: Formato de un Datagrama UDP

El tipo ICMP incluye todas las reglas con el campo protocolo ICMP (Figura 4.6). Se corresponde con el protocolo de la capa de red del mismo nombre. Es el encargado del intercambio de mensajes de control de red y de mensajes de error en el caso de que se produzca algún fallo en la comunicación.

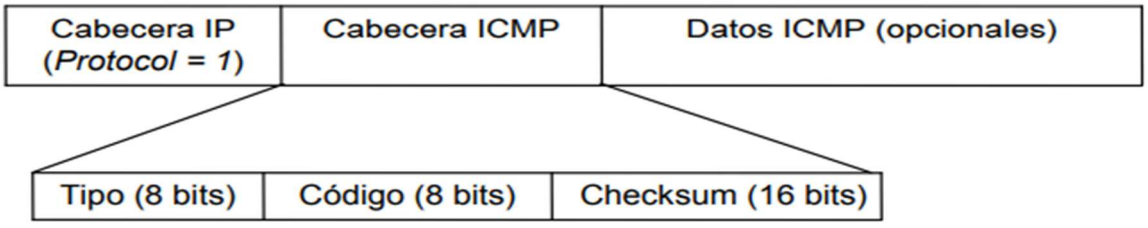


Figura 4.6: Formato de un mensaje ICMP

4.1.1.2 Reglas tipo *Content*

Hemos decidido separar este tipo en dos subgrupos, el referente al campo *Content* básico y el grupo de reglas de tipo *Content* con opciones de recortado de paquetes.

Tipo *Content*: Reglas en las que la identificación de los paquetes maliciosos depende del campo *content* y de sus opciones como, por ejemplo, *nocase*. Es el tipo más importante en este trabajo, en el estudio de reglas, entre ellas las reglas Snort, el 87% de estas están relacionadas con el campo *content* [2]. En nuestro proyecto será el foco principal tanto en reglas como en paquetes.

Tipo *Content* con opciones de recortado de paquetes: Reglas en las que la identificación de un ataque o un paquete malicioso depende de las opciones del campo *content*, como el *offset* y *depth*. Son una ampliación del tipo anterior y tienen mucha importancia por la misma razón que el tipo *content*.

Tras la recepción de la regla o fichero de entrada nuestro programa es capaz de reconocerlas, almacenarlas y enviárselas al Módulo conversor a ERs.

Este programa puede ser usado con cualquier otro software que requiera de la separación en las distintas partes y opciones de una regla. Esto permite la diferenciación clara de todos los campos de una regla y la división de la regla eligiendo las opciones deseadas para un futuro reconocedor de patrones o software similar.

Esta funcionalidad hace que este programa sea un software adaptable y reutilizable. El programa está escrito en Java, esto es así por conveniencia, pues queríamos continuar con la línea que seguían los trabajos previos también desarrollados en Java.

El método para reconocer una regla consiste en realizar un filtrado que separa la regla en cabecera (Rule Header) y opciones (Rule options). Una vez realizada esta primera división aplicamos unos filtros para separar las diferentes opciones de forma individual con sus campos (Figura 4.7). Hemos realizado diversas mejoras a este método, entre ellas la eliminación de las comillas de algunos campos de opciones como "*content*" o "*msg*" donde no es importante su conservación.

Para reconocer una regla de manera individual se aplica directamente el método descrito anteriormente, mientras que para el reconocimiento de un conjunto de reglas en un fichero de texto se necesita un preprocesamiento, que separa el conjunto de reglas en reglas individuales y las almacena en una lista para poder aplicar el descrito a cada una.

Tras la aplicación del método de filtrado la lista resultante es enviada al Módulo conversor de ERs.

Este módulo sufrió una modificación y mejora en las fases finales de desarrollo del proyecto, ya que las reglas Snort con campo *content* en hexadecimal separado por *pipes* "|" producían un error debido a una restricción de la herramienta de reconocimiento de patrones (veáse sección 3.2.2). Este error fue subsanado, y de esta manera validamos que nuestro programa puede recoger cualquier regla como entrada y procesarla.

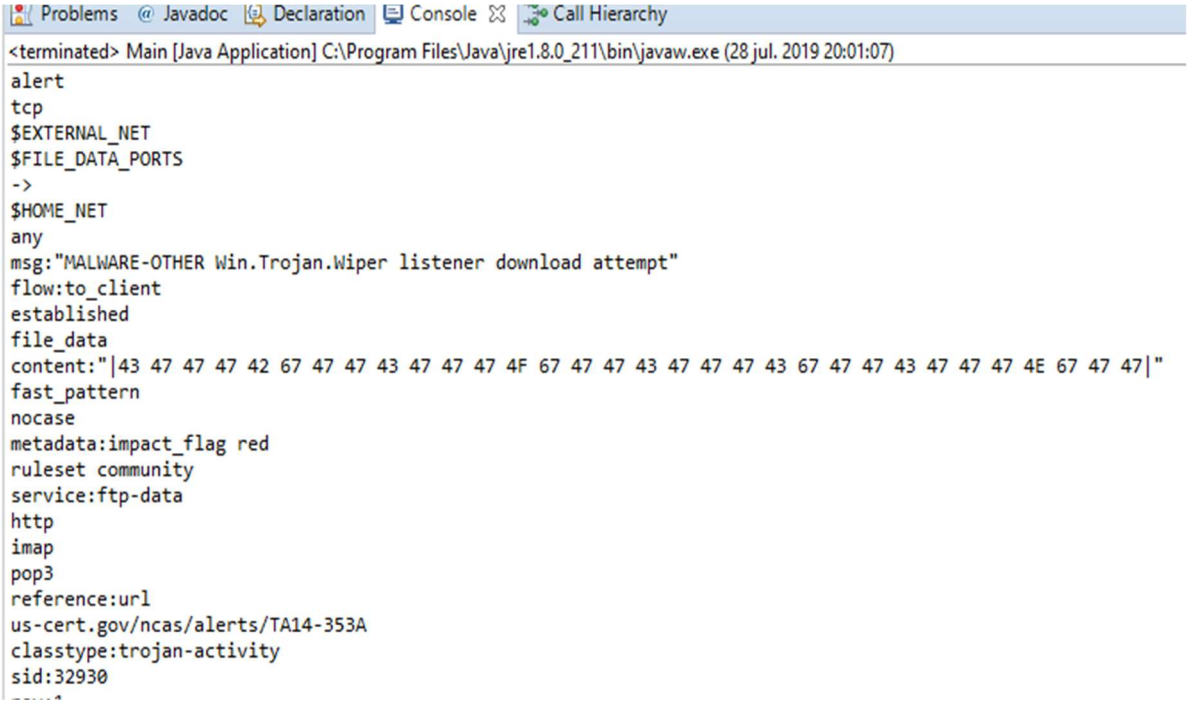


Figura 4.7: Ejemplo de fichero de salida

4.1.2 Diseño del Módulo conversor a ERs

Este módulo es el encargado de transformar la lista que le llega desde el Módulo de reglas en una expresión regular asociada a cada regla Snort. Esta transformación sigue las normas de sintaxis de expresiones regulares explicadas en la sección 3.1.

Nuestro programa es capaz de convertir cualquier regla Snort en expresión regular. En un principio la conversión era estricta a la sintaxis de las ERs, tomando elementos como el inicio y fin de regla ("^" y "$" respectivamente) y utilizando la totalidad de los caracteres de las reglas Snort.

Por ejemplo: ^((! "GET").)*$ sería la expresión regular asociada a una regla de tipo *content*, que busque reconocer una petición *"GET"* en los paquetes de datos.

Durante el desarrollo tuvimos que modificar y actualizar este módulo con el fin de poder unificarlo con la herramienta de reconocimiento de patrones. Esta modificación consistió en cambiar la lógica con la que traducimos la lista inicial a expresiones regulares, ya que no todas eran válidas en el proyecto (véase sección 3.2.2).

Tras todo el proceso de desarrollo la salida producida por este módulo es un fichero de texto con las expresiones regulares que usaremos en la herramienta de reconocimiento de patrones, tal y como se puede observar en la Figura 4.8.

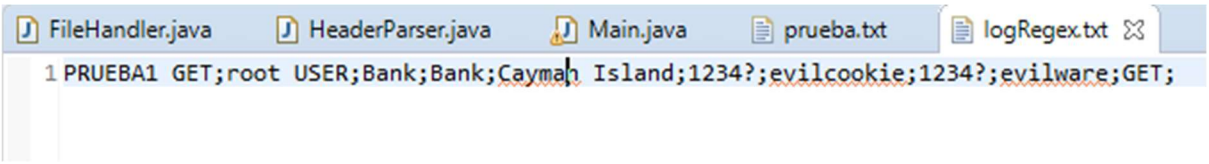


Figura 4.8: Ejemplo de salida para un fichero PRUEBA1 de 9 reglas Snort analizando el campo *content*

4.1.3 Diseño del Módulo de paquetes

Este módulo es el encargado de capturar el paquete de datos original, duplicarlo y modificarlo. La modificación se realiza en base a las opciones que las reglas tengan en el campo *Content.* Se mantiene el paquete de datos original para poder seguir reconociendo otras reglas Snort de otro tipo (véase sección 4.1.1) y a su vez se generan más paquetes con las modificaciones necesarias para poder procesar todas las reglas que se introduzcan.

En un principio no se meditó sobre la necesidad de este módulo, pues la idea inicial no requería de un preprocesamiento previo de estos paquetes.Una vez se descubrieron las restricciones del trabajo anterior respecto a las expresiones regulares (véase sección 3.2.2) se decidió crear este módulo e implementarlo.

Este módulo recibe como entrada un archivo con el paquete de datos en el que se buscará el patrón a detectar por las reglas Snort.

La función principal del programa es adaptar y facilitar el trabajo de la búsqueda de patrones con reglas Snort, para ello se separan las distintas partes del datagrama, como cabecera y datos, y se eliminan aquellas partes que no son útiles para el proyecto.

La lógica de este módulo es generar copias de varios segmentos de los paquetes de prueba independientes al original para simular ciertas opciones de las reglas que solo buscan patrones en estos segmentos específicos. Al basarse esta acción en las opciones de las reglas, pensamos que deberíamos conocer cada opción de cada regla para poder realizar de manera apropiada los recortes y modificaciones de los paquetes. Dependiendo de cada patrón, hacemos un recorte, por eso este módulo es dependiente del Módulo de reglas, pero se ejecuta de forma simultánea al Módulo conversor a expresiones regulares.

Las salidas de este módulo son el paquete original y tantas copias modificadas de este como recortes sean necesarios para reconocer las múltiples reglas de entrada (Figura 4.9).

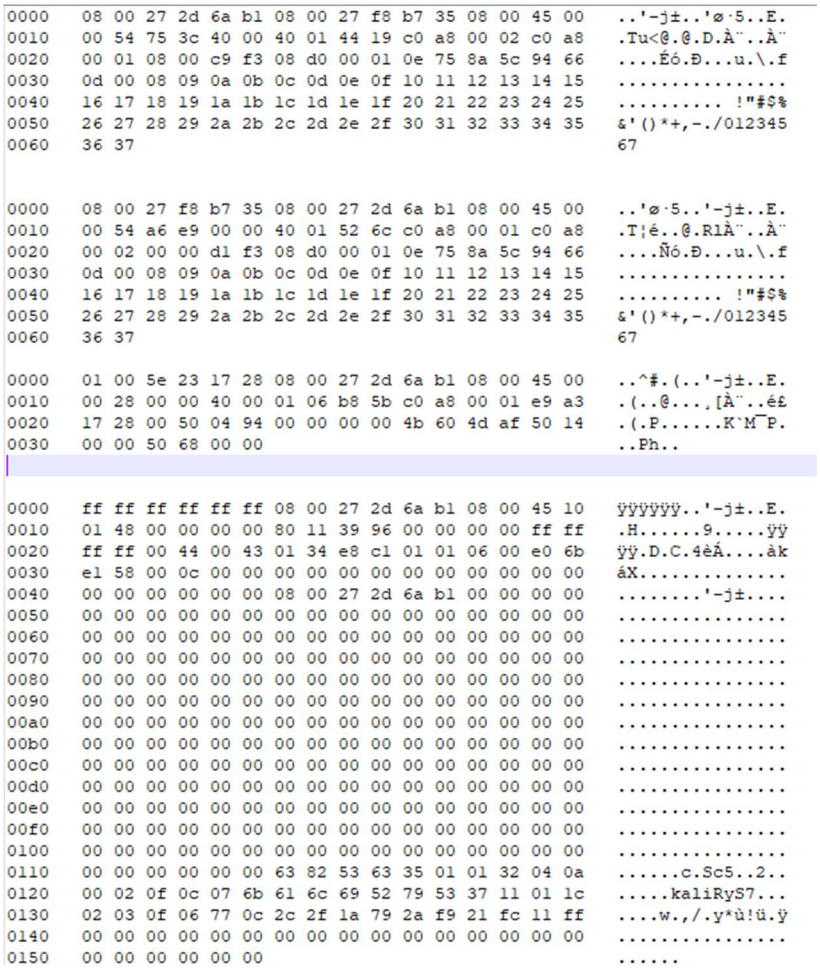


Figura 4.9: Ejemplo de paquete de datos

4.2 Implementación del IDS Hardware

En esta sección se describe con alto nivel de detalle el diseño de cada módulo del sistema final, así como las modificaciones que hemos ido introduciendo para mejorar distintos aspectos de la implementación.

El desarrollo se ha hecho de la manera más flexible posible para poder realizar la implementación de forma sencilla, siempre teniendo en mente los diseños de los módulos (véase sección 4.1). Además, se ha sometido a un constante proceso de validación gracias al uso de los *testbech* de la propia herramienta de reconocimiento de patrones.

En las siguientes subsecciones se presentan los detalles técnicos de mayor relevancia de la implementación.

4.2.1 Implementación del Módulo de reglas

Este módulo es el módulo inicial del sistema, tal y como hemos explicado en el diseño. Comienza leyendo un fichero de texto que contiene las reglas. A medida que las lee, las procesa y las almacena en un *string*.

Tras esto, comienza el proceso de filtrado o separación. Empezamos separando la cabecera (*header)* de las opciones (*options)* usando la apertura de paréntesis, *"(", como* punto de división. Esta separación se realiza pasando cada regla por un bucle que la divide y almacena en un mapa de listas (una lista por cada regla, y un string por cada opción) (Figura 5.10).

El índice del bucle se inicializa a 15 porque es el mínimo de caracteres que puede ocupar una cabecera en una regla y de esta manera se reduce el número de iteraciones del bucle.

Una vez separados, procesamos la cabecera dividiéndola por espacios, separando de esta manera cada uno de los parámetros, tales como: acción, protocolo, direcciones y puertos, etc.

El procesamiento del segmento Opciones es más complejo, divide sus campos separándolos por punto y coma, ";", consiguiendo una separación por campos, cada uno de ellos con sus opciones internas todas juntas. Esta división se realiza de *string* a *sub-strings.*

En la cabecera el último elemento es un elemento vacío y en el segmento Opciones es un ")", por ello se eliminan ambos.

```
public static List<List<String>> parseo(String nombre) throws IOException {
    List<List<String>> finalParse = new ArrayList<List<String>>();
    File archivo = null;
    archivo = new File (nombre);
    FileReader fileread = new FileReader(archivo);
    BufferedReader buffer = new BufferedReader(fileread);
    String line = buffer.readLine();
    boolean found = false;
    int i = 15;
    while(line!= null) {
        line = line.trim();
        found =false;
        i = 15;
        while(!found) {
            if(line.charAt(i) == '(')
            {
                found = true;
            }
            else
                i++;
        }
        List<String> sal = new ArrayList<String>();
        String salida0= line.substring(0, i);
        String salida1 = line.substring(i+1);
        sal.addAll(Arrays.asList(salida0.split(" ")));
        sal.addAll(Arrays.asList(salida1.split(";")));
        sal.remove(sal.size()-1);
        finalParse.add(sal);
        line = buffer.readLine();
    }
    buffer.close();
    return finalParse;
}
```

Figura 4.10: Código Módulo de reglas

Estos datos ya separados son fragmentados usando una interfaz llamada *Options* que, dependiendo del tipo de regla y ataque buscado, separa y recoge la opción y el protocolo correspondiente. En el caso de opciones con sub-opciones, como *content*, este fragmentado se realiza respetando las sub-opciones y, en el caso de ser útiles, procesándolas.

Por ejemplo, una regla con campo *Content* y dentro de este la opción *depth* será procesada y dividida de igual manera que el resto. En caso de que existan más opciones se mantendrían y la opción *depth* que afecta a los paquetes sería procesada.

La interfaz *Options* es extendida por distintas clases, una por sistema (protocolo, *content, depth, offset*). Se utiliza un *parser* convencional para identificar cuál es cada opción (Figura 4.11).

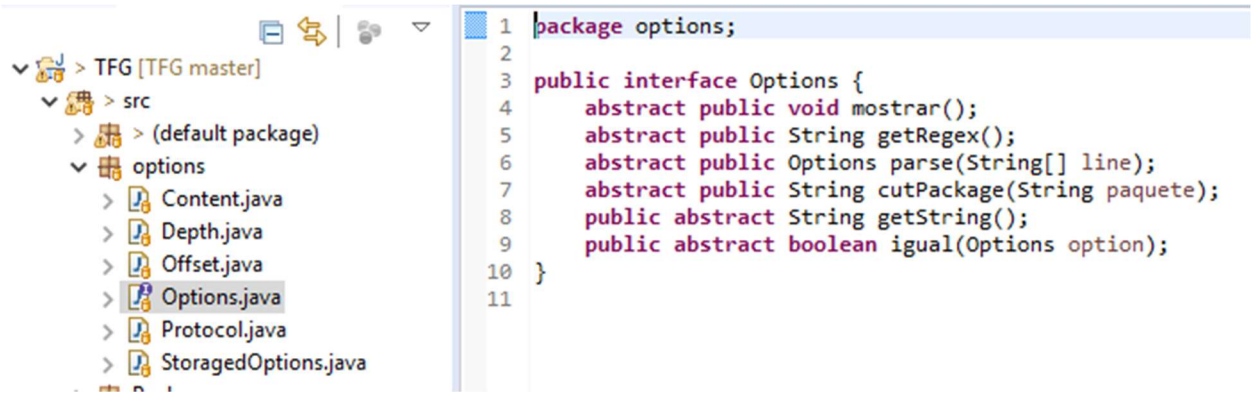


Figura 4.11: Estructura de la Interfaz *Options* y sus clases asociadas

Cada opción reconocida de una regla es guardada en una lista junto con el resto de las opciones de la misma regla.

A este sistema, para realizar pruebas de rendimiento basadas en tiempos (véase capítulo 5), se le añadió un sistema de comparación, en el que se compara cada regla que vaya a ser añadida con las ya almacenadas, si resulta que una regla es igual a cualquiera de las anteriores se descarta.

4.2.2 Implementación del Módulo conversor a ERs

Tras procesar todas las reglas del archivo inicial y almacenarlas en un mapa de listas de opciones, este módulo se encarga de coger cada lista de opciones y transformarla en una expresión regular.

Para este proceso, se usa un método abstracto getRegex() de la interfaz *Options,* que está implementado en cada una de las clases que extienden a esta interfaz. Usa los métodos de transformación a expresión regular explicados en la sección de diseño.

Este método transforma por separado cada opción de la lista en una expresión regular que se almacena en un *string*. Este *string* contendrá la expresión regular resultante de analizar la regla Snort con todas sus opciones.

En el caso del tipo Protocolo se busca el símbolo correspondiente al protocolo en hexadecimal, pues es así como está contenido en el paquete (Figura 4.12).

```
@Override
public String getRegex() {
    // TODO Auto-generated method stub
    switch (this.tipo) {
    case "TCP":
        byte[] s = DatatypeConverter.parseHexBinary("06");
        return new String(s);
    case "ICMP":
        byte[] icmp = DatatypeConverter.parseHexBinary("01");
        return new String(icmp);
    case "IP":
        break;
    case "UDP":
        byte[] udp = DatatypeConverter.parseHexBinary("11");
        return new String(udp);
    default:
        return "";
    }
    return tipo;
}
```

Figura 4.12: Ejemplo de getRegex para el tipo Protocolo

Este módulo transforma todas las listas de opciones formando una expresión regular por cada regla Snort original. Todas estas expresiones regulares se separan entre ellas por punto y coma, ";", siguiendo el estándar de entrada de la herramienta de reconocimiento de patrones heredada del trabajo previo.

Finalmente, el *string* que contiene todas las expresiones regulares ya transformadas se escribe en un fichero para su envío al conversor a VHDL.

4.2.3 Implementación del Módulo de paquetes

Este módulo está separado del resto y su función es adaptar el paquete para el correcto funcionamiento de las reglas con sus distintas opciones del campo *content* que detectan o buscan elementos en posiciones definidas de este.

Los paquetes son paquetes reales de datos o datagramas extraídos del tráfico de una red, se reciben desde un archivo de texto, en formato hexadecimal, divididos por un salto de línea, un punto y coma y otro salto de línea.

Al haber recogido todas las opciones de las reglas iniciales en un mapa de listas de opciones era necesario recorrerlo para poder trabajar con todas las posibles opciones que caracterizan el paquete y determinan su posterior recorte.

Para poder realizar los recortes añadimos la función *cutPackage()* a la interfaz *Options* para que, dependiendo del tipo de opción, devolviera el paquete recortado.

CutPackage() duplica los bytes del paquete según indica la opción del campo *content* de cada regla. En este método hacemos uso de la función *substring* ya existente en la librería de Java para facilitarnos el recorte del paquete de datos.

Tras recortar el paquete de todas las formas que las opciones de las reglas necesitan, el resultado son tantos paquetes como opciones modifican a este. Estos paquetes son almacenados en un fichero separados por punto y coma, ";" (véase Figura 4.13).

```
File archivoPackage = new File("logPackage.txt");
BufferedWriter buffer = new BufferedWriter(new FileWriter(archivoPackage));
//Version sin hilo
if(!hilo)
    for(int i = 0; i < paquetes.size();i++) {
        for(int j=0;j<listadelistadeopciones.size();j++) {
            String aux = paquetes.get(i);
            buffer.write(aux+"\n");
            for(int z = 0; z < listadelistadeopciones.get(j).size();z++) {
                String recorte = listadelistadeopciones.get(j).get(z).cutPackage(aux);
                if(recorte!= null) {
                    buffer.write(recorte+"\n;\n");
                }
            }
        }
    }
//Version con hilo
else {
    ParserPaquetes thread = new ParserPaquetes(listadelistadeopciones);
    thread.start();
}
buffer.close();
```

Figura 4.13: Método principal de recorte de paquetes

La explicación de por qué almacenar todos los diferentes recortes que se producen, junto con el paquete original es la necesidad de evitar errores relativos a reconocer una regla que requería recorte en el paquete original o no reconocer una regla porque el paquete está recortado.

Al enviar todos los paquetes, cada regla podrá ser reconocida con su paquete correspondiente, evitando errores en el reconocimiento. Este fichero, que contendrá todos los paquetes, será el que se enviará a la FPGA.

Se implementó un multihilo para poder realizar la escritura de reglas simultáneamente al recorte de paquetes. Esto resultó muy útil de cara a las pruebas sobre la FPGA (véase capítulo 5).

Se puede observar el código asociado a este proyecto en: https://github.com/Fvalley/TFG.

Capítulo 5

Resultados experimentales

En este capítulo exponemos las pruebas que hemos realizado sobre nuestro sistema, así como los resultados obtenidos.

En la sección 5.1 se explican las tecnologías usadas y descartadas para el desarrollo del proyecto, en la sección 5.2 se profundiza en la obtención de los paquetes que se utilizaron para realizar las pruebas y en la sección 5.3 se exponen los resultados experimentales obtenidos.

5.1 Tecnologías

Para la realización de este trabajo se han valorado distintas herramientas y tecnologías, se ha decidido la utilización de aquellas cuyas características se adaptan mejor a nuestro proyecto. A continuación, se expone en una tabla las tecnologías usadas y descartadas exponiendo los motivos que nos han llevado a tomar esta decisión.

Java	Hemos elegido este lenguaje de programación debido a la amplia variedad de librerías y su fácil uso en proyectos orientados a objetos. Además, el trabajo previo estaba desarrollado en este lenguaje y la integración de ambos trabajos podría resultar más compacta.
Regex	El uso de expresiones regulares fue de carácter obligatorio, dado que el trabajo previo estaba realizado para recoger como entrada del sistema estas expresiones (véase capítulo 3).
IDE Eclipse	Decidimos usar Eclipse como entorno de desarrollo porque es sólido, útil y cómodo. También, porque los integrantes ya habíamos trabajado con este entorno de desarrollo. Asimismo, Eclipse destaca por su versatilidad y compatibilidad. Todo esto facilita el desarrollo y la unificación con la herramienta de reconocimiento de patrones y el desarrollo en VHDL y las FPGAs.
Git	Tecnología usada para el control de versiones del programa. Los integrantes ya la habíamos utilizado como herramienta en otros ámbitos durante la carrera por lo que nos pareció una opción sencilla de usar. Resulta una herramienta muy útil para este tipo de proyectos software.

Wireshark	Decidimos usar Wireshark dado que es el sistema de análisis de protocolos de red más extendido del mundo. En particular lo hemos usado para capturar los paquetes de datos con los que probar el funcionamiento de nuestro programa y usarlo junto con las reglas Snort en la detección de paquetes maliciosos. Los integrantes ya lo habíamos utilizado en diversas asignaturas del grado y nos parecía una alternativa *open source* muy interesante.
Vivado (Xilinx)	Vivado Design Suite es una *suite* software para el desarrollo y análisis de diseños HDL. Para realizar las pruebas de unión de nuestro proyecto con la herramienta de reconocimiento de patrones utilizamos este entorno dada la experiencia previa y la recomendación de los tutores.
Kali-Linux	Utilizamos la máquina virtual Kali para generar paquetes con los que hacer las pruebas. Se han simulado ataques, utilizando para ello las distintas herramientas que vienen instaladas por defecto en Kali, entre ellas Metasploit.
Drive y Onedrive	Tecnologías usadas para el control de versiones de la memoria y numerosos archivos científicos a examinar durante el proyecto. Son herramientas fáciles de usar y estábamos familiarizados con ellas.
Notación Postfija	La notación polaca inversa o notación postfija es un método de introducción de datos alternativo. Comprender este sistema algebraico fue necesario para encontrar fallos y poder realizar la unificación de nuestro desarrollo con la herramienta de reconocimiento de patrones (véase sección 3.2).
C y C++	Descartamos estos lenguajes por la nula o complicada aplicación de la orientación a objetos, además de las dificultades que podrían suscitar de cara a la unificación con el reconocedor de patrones.
Python	Descartamos esta tecnología por su tipado dinámico, que podría ser contraproducente en este proyecto generando errores si no se controla adecuadamente. Aparte, Java ofrece una buena portabilidad que permite simplificar el trabajo cooperativo, además de facilitar la unión con el trabajo previo (véase capítulo 3).
C#	Descartamos este lenguaje de programación por la inexperiencia de los participantes con su uso. Es similar a Java y podría haber sido un gran candidato para el desarrollo de nuestra herramienta.

5.2 Obtención de paquetes de prueba

Para la realización de pruebas y la obtención de resultados experimentales hemos utilizado paquetes simulando distintos tipos de ataque. Se trata de paquetes de datos relativos a los protocolos usados por nuestro sistema, descritos en la sección 4.1.1.1.

Las herramientas usadas en este proceso han sido la máquina virtual Kali-Linux, la herramienta de análisis y filtrado de tráfico de red, Wireshark y la herramienta Hping3.

Para simular los ataques necesitábamos una máquina atacante y una máquina víctima, ambas en una misma red. Por ello, creamos una red local entre dos máquinas Kali, simulando un posible ataque real en la red.

Recogimos paquetes en ambas máquinas usando Wireshark, tanto los paquetes recibidos por la víctima como los enviados por el atacante. Estos paquetes fueron almacenados en un fichero de texto, posteriormente separados por ";" para su procesamiento por el Módulo paquetes (véase sección 4.1.3).

Se han realizado 3 tipos de ataque: ICMP *flooding* con *spoof*, TCP SYN *flooding* y un ataque a protocolo UDP.

ICMP *flooding* con *spoof*

Es un ataque de denegación de servicio, DoS. Usa direcciones IP generadas de manera aleatoria como emisores de mensajes ICMP *echo requests* con el fin de saturar la máquina de la víctima. El receptor o víctima intenta responder a todos los mensajes de control enviados por el atacante con mensajes *echo reply* pero al ser una cantidad inmensa, la máquina deja de funcionar.

El comando usado por el atacante es:

hping3 --icmp --flood –-rand-source *dirección-víctima*

Si quisiéramos falsear la ip del atacante sin generar IPs aleatorias se podría usar la opción -a *dirección-falsa.*

TCP SYN *flooding*

Es un ataque DoS. Envía mensajes de protocolo TCP y el flag SYN activado de manera que la máquina receptora se piensa que está iniciando una conexión, pero el *handshake* nunca se llega a completar. Esto provoca que la máquina víctima deje tantas conexiones abiertas que saturan el sistema. Este tipo de ataques suelen estar dirigidos contra una máquina, un punto de acceso o un cortafuegos.

El comando usado por el atacante es:

hping3 -V -c 1000 -d 100 -S --flood -p --rand-source *dirección-víctima*

Las opciones -c y -d permiten configurar el número de paquetes enviados y su tamaño, respectivamente.

UDP

El ataque al protocolo UDP más común es por *reflection flood*, pero al no tener distintos servidores en los que reflejar y debido a que la extracción de paquetes debía ser un proceso simple se decidió usar un ataque UDP *flooding* normal.

El comando usado por el atacante es:

hping3 --udp --flood *dirección-víctima*

Otras alternativas

Se valoró la posibilidad de realizar un ataque DNS *spoofing* aunque no se llegó a realizar finalmente debido a que la cantidad de paquetes recogidos era suficiente para la realización de las pruebas y la obtención de los resultados experimentales.

Para las pruebas con reglas de tipo *Content* se crearon paquetes sintéticos. Por ejemplo, introdujimos en el paquete el patrón “GET” a reconocer por una o varias de las reglas introducidas en la FPGA.

5.3 Resultados experimentales

En esta sección se presentan los resultados experimentales obtenidos al implementar el sistema IDS propuesto. Se han realizado las mediadas utilizando diferentes métodos, técnicas, simulaciones; además de numerosas y variadas colecciones de datos.

En un principio las pruebas fueron realizadas utilizando reglas simples y sintéticas, creadas por nosotros siguiendo la sintaxis de Snort. Una vez que finalizamos todo el desarrollo, para validar el funcionamiento del sistema, extrajimos un paquete de reglas completo de la página oficial de Snort [23], unas 4000 reglas (Figura 5.1), y lo utilizamos para probar la eficacia de nuestro IDS.

```
3  # This file contains rules that were created by Sourcefire, Inc. and other third parties
4  # (the "GPL Rules") that are distributed under the GNU General Public License (GPL),
5  # v2. The GPL Rules created by Sourcefire are owned by Sourcefire, Inc., and the GPL
6  # Rules not created by Sourcefire are owned by their respective owners. Please see
7  # the AUTHORS file included in the community package for a list of third party owners and their
8  # respective copyrights.
9  #
10 # This file does not contain any Sourcefire VRT Certified Rules; the VRT Certified
11 # Rules are distributed by Sourcefire separately under the VRT Certified Rules License
12 # Agreement (v 2.0)
13 #
14 #-----------------
15 # COMMUNITY RULES
16 #-----------------
17
18 # alert tcp $HOME_NET 2589 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR - Dagger_1.4.0"; flow:to_client,established; content:
19 # alert tcp $EXTERNAL_NET any -> $HOME_NET 7597 ( msg:"MALWARE-BACKDOOR QAZ Worm Client Login access"; flow:to_server,establ
20 # alert tcp $EXTERNAL_NET any -> $HOME_NET 12345:12346 ( msg:"MALWARE-BACKDOOR netbus getinfo"; flow:to_server,established; c
21 # alert tcp $HOME_NET 20034 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR NetBus Pro 2.0 connection established"; flow:to_clie
22 # alert tcp $HOME_NET any -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR Infector.1.x"; flow:established,to_client; content:"W
23 # alert tcp $HOME_NET 666 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR SatansBackdoor.2.0.Beta"; flow:to_client,established;
24 # alert tcp $HOME_NET 6789 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR Doly 2.0 access"; flow:established,to_client; conten
25 # alert tcp $EXTERNAL_NET 1000:1300 -> $HOME_NET 146 ( msg:"MALWARE-BACKDOOR Infector 1.6 Client to Server Connection Reques
26 # alert tcp $HOME_NET 31785 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR HackAttack 1.20 Connect"; flow:established,to_clien
27 # alert tcp $EXTERNAL_NET any -> $HOME_NET 21 ( msg:"PROTOCOL-FTP ADMw0rm ftp login attempt"; flow:to_server,established; co
28 # alert tcp $HOME_NET 30100:30102 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR NetSphere access"; flow:established,to_client;
29 # alert tcp $HOME_NET 6969 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR GateCrasher"; flow:established,to_client; content:"Ga
30 # alert tcp $HOME_NET 5401:5402 -> $EXTERNAL_NET any ( msg:"MALWARE-BACKDOOR BackConstruction 2.1 Connection"; flow:establish
```

Figura 5.1: Paquete de reglas Snort

Debido a las restricciones del sistema heredado (véase sección 3.2.2) tuvimos que eliminar una serie de reglas de este paquete ya que contenían caracteres en hexadecimal que al ser convertidos a ASCII y, posteriormente, a expresión regular eran caracteres no válidos para nuestro sistema.

Por otro lado, para la realización de pruebas y obtención de resultados obtuvimos paquetes de datos que fueron capturados con la herramienta Wireshark, simulando en la máquina Kali los ataques que deseábamos detectar. Además, se crearon paquetes sintéticos y más simples para demostrar el correcto funcionamiento del sistema y evitar problemas de restricciones y falsos positivos.

Estas pruebas se han podido llevar a cabo siguiendo varios métodos gracias a la implementación realizada (véase cap. 4). Usando solo una regla y solo un paquete, múltiples reglas con un solo paquete, una sola regla con múltiples paquetes y múltiples reglas con múltiples paquetes. Además de la alternancia entre monohilo y multihilo y el uso de una comparación para las repeticiones de reglas similares en paquetes grandes.

La arquitectura del sistema está implementada por módulos (véase capítulo 5) de tal manera que las reglas son procesadas por el programa la primera vez que se usa el IDS y cada vez que se quieran modificar o reemplazar reglas. En cambio, los paquetes son duplicados y fragmentados en cada ejecución. Por este motivo, para analizar la eficiencia del sistema, el tiempo relacionado con la partición y duplicado de paquetes es importante. Hemos medido el tiempo de introducir un paquete de datos y cotejarlo con un número variable de reglas en cinco ejecuciones diferentes (Tabla 5.1).

Tabla 5.1: Tiempos de ejecución (ms) de fragmentar y duplicar paquetes

1 regla	2 reglas parecidas	2 reglas distintas	10 reglas	100 reglas	1000 reglas
1330	2,905	17,852	34,289	84,431	193,323
1655	1,701	18,888	23,876	41,137	131,096
1275	1,631	16,916	17,632	32,542	120,801
1171	2,314	13,277	17,433	34,338	144,500
1824	2,043	9,180	34,964	43,225	140,611

Como se puede observar en la figura, un sistema con 1000 reglas es capaz de procesar un paquete en un tiempo relativamente pequeño. Los tiempos son incrementales, y cabe resaltar que con el incremento de reglas se produce también un aumento en la cantidad de veces que hay que fragmentar un paquete, alcanzándose tiempos de ejecución mayores.

A continuación, se exponen dos tablas, la Tabla 5.2 y la Tabla 5.3, en las que se compara el rendimiento en milisegundos de nuestro programa tras las diferentes pruebas realizadas y el rendimiento del sistema IDS completo. Se ha usado multihilo, explicado en la sección 4.2.2, y la comparación descrita en la sección 4.2.1 como alternativas de implementación con posible impacto en el rendimiento.

Tabla 5.2: Tiempos de ejecución (ms) de nuestro programa

Multihilo	Comparación	1 regla	2 reglas parecidas	2 reglas distintas	10 reglas	100 reglas	1000 reglas
NO	NO	232,403	357,245	389,115	430,190	784,415	3.037,813
	SI	-----	241,464	283,727	309,170	879,132	2.791,585
SÍ	NO	227,952	258,919	211,985	242,354	389,970	2.570,294
	SI	-----	100,257	270,888	241,962	672,930	2.151,958

Como se puede observar en los resultados obtenidos, para una regla la mejora usando el multihilo no es significativa. Pero, en cuanto aplicamos esta técnica al uso de dos o más reglas la diferencia de tiempos es muy elevada, siendo posible reducir el tiempo de ejecución a la mitad.

Podemos comprobar que el uso de la comparación, explicada en la sección 4.2.1, es útil cuando hay semejanza entre reglas, aunque no se consiguen grandes resultados. Cabe destacar que este método penaliza mucho el rendimiento en la ejecución de conjuntos de datos no demasiado grandes en los que pueda no existir demasiada semejanza entre reglas, como es el caso de la prueba de 100 reglas.

El rendimiento con 1000 reglas es coherente, la comparación es útil, pero siguen sin conseguirse grandes incrementos en el rendimiento, aunque en conjunto usando multihilo la eficiencia es buena.

Como se puede observar en la Tabla 5.3 los tiempos globales del sistema son coherentes con los tiempos de ejecución de nuestro programa. Cabe destacar la eficiencia del multihilo afectando a los tiempos del sistema completo. También podemos observar que cuando la cantidad de reglas a procesar por la herramienta de reconocimiento de patrones es grande, como en el caso de la ejecución de 1000 reglas, los resultados escalan debido a la gran cantidad de procesamiento de ERs que esta tiene que realizar.

Tabla 5.3: Tiempos de ejecución (ms) globales del sistema

Multihilo	Comparación	1 regla	2 reglas parecidas	2 reglas distintas	10 reglas	100 reglas	1000 reglas
NO	NO	436,191	767,445	873,226	454,883	4.921,463	110.349,166
	SI	-----	390,521	750,779	580,238	4.748,204	79.256,007
SÍ	NO	394,511	618,251	827,342	352,345	2.291,740	131.258,840
	SI	-----	231,552	148,734	388,450	5.741,950	121.597,719

Hemos realizado simulaciones en FPGA usando la herramienta Vivado [9]. La FPGA empleada para la simulación fue una Virtex-7 XC7VX485TFFG1157-1. Estas simulaciones las hemos realizado con 1, 2, 10, 100 y 1000 reglas, siguiendo el esquema usado en las pruebas realizadas con tiempos.

A continuación, mostramos para las diferentes simulaciones los resultados de la implementación en relación con el área de la FPGA ocupada.

Todas las LUTs usadas en la FPGA en cada prueba son de tipo lógico por lo que el número de LUTs y el número de CLBs coincide.

1 Regla

Como podemos observar en la Figura 5.2 el área utilizada de la FPGA es muy pequeña.

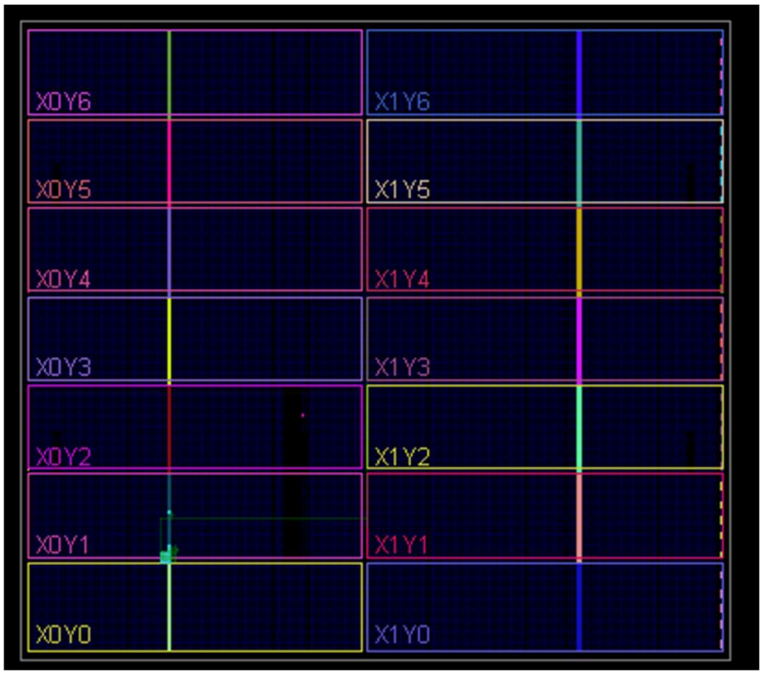


Figura 5.2: Esquema de la utilización de la FPGA para 1 regla

La implementación de nuestro sistema introduciendo una regla utiliza 9 LUTs de las 204000 disponibles. El aprovechamiento de LUTs es muy bueno, de las 9 utilizadas, 6 están usadas en su totalidad (Figura 5.3). En la Figura 5.4 podemos distinguir claramente el área utilizada y las conexiones que se establecen cuando se realiza la implementación de nuestro IDS en la FPGA utilizando una sola regla.

```
Slice Logic Utilization:
  Number of Slice Registers:                 6 out of 408,000    1%
    Number used as Flip Flops:               6
    Number used as Latches:                  0
    Number used as Latch-thrus:              0
    Number used as AND/OR logics:            0
  Number of Slice LUTs:                      9 out of 204,000    1%
    Number used as logic:                    9 out of 204,000    1%
      Number using O6 output only:           8
      Number using O5 output only:           0
      Number using O5 and O6:                1
      Number used as ROM:                    0
    Number used as Memory:                   0 out of  70,200    0%
    Number used exclusively as route-thrus:  0

Slice Logic Distribution:
  Number of occupied Slices:                 6 out of  51,000    1%
  Number of LUT Flip Flop pairs used:        9
    Number with an unused Flip Flop:         3 out of       9   33%
    Number with an unused LUT:               0 out of       9    0%
    Number of fully used LUT-FF pairs:       6 out of       9   66%
    Number of unique control sets:           1
    Number of slice register sites lost
      to control set restrictions:           2 out of 408,000    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                    11 out of     600    1%
```

Figura 5.3: Uso de recursos para 1 regla

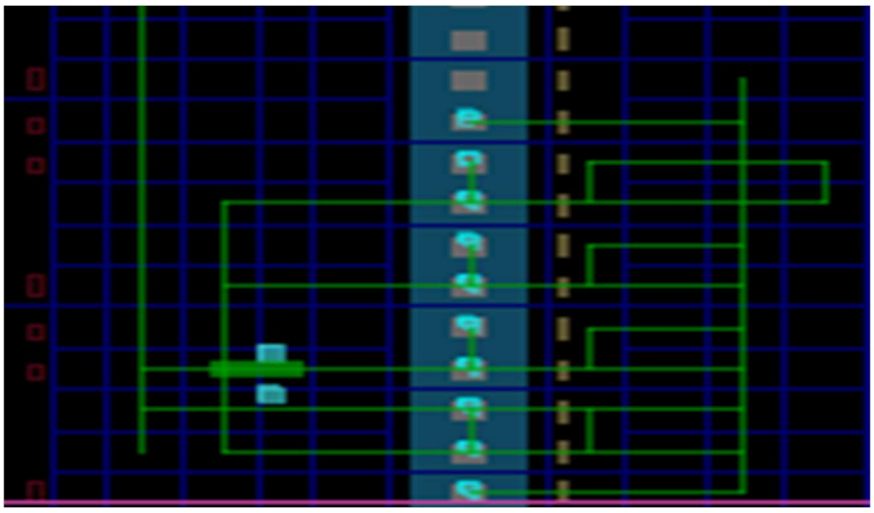


Figura 5.4: Esquema ampliado de la utilización de la FPGA para 1 regla

2 Reglas

A simple vista no se observa ningún incremento en el área utilizada, tal y como muestra la Figura 5.5.

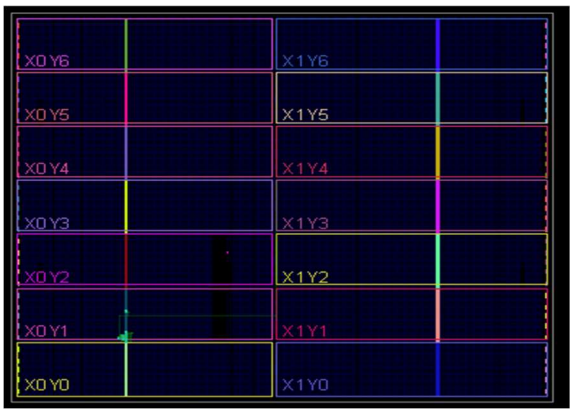


Figura 5.5: Esquema de la utilización de la FPGA para 2 reglas

Llama la atención el reducido uso de recursos de la FPGA que en la mayoría de componentes es un 1%. Utilizando dos reglas distintas se usan 33 LUTs y se incrementa el uso de pares LUT-FF completos hasta un 93% (Figura 5.6).

```
Slice Logic Utilization:
  Number of Slice Registers:                    31 out of 408,000     1%
    Number used as Flip Flops:                  31
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                         33 out of 204,000     1%
    Number used as logic:                       33 out of 204,000     1%
      Number using O6 output only:              31
      Number using O5 output only:               0
      Number using O5 and O6:                    2
      Number used as ROM:                        0
    Number used as Memory:                       0 out of  70,200     0%
    Number used exclusively as route-thrus:      0

Slice Logic Distribution:
  Number of occupied Slices:                    31 out of  51,000     1%
  Number of LUT Flip Flop pairs used:           33
    Number with an unused Flip Flop:             2 out of      33     6%
    Number with an unused LUT:                   0 out of      33     0%
    Number of fully used LUT-FF pairs:          31 out of      33    93%
    Number of unique control sets:               1
    Number of slice register sites lost
      to control set restrictions:               1 out of 408,000     1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                        11 out of     600     1%
```

Figura 5.7: Esquema ampliado de la utilización de la FPGA para 2 reglas

10 Reglas

Si se observan las Figuras 5.8 y 5.10 los cambios en el área tampoco son significativos, aunque se aprecia un mayor número de interconexiones y uso de área de la placa.

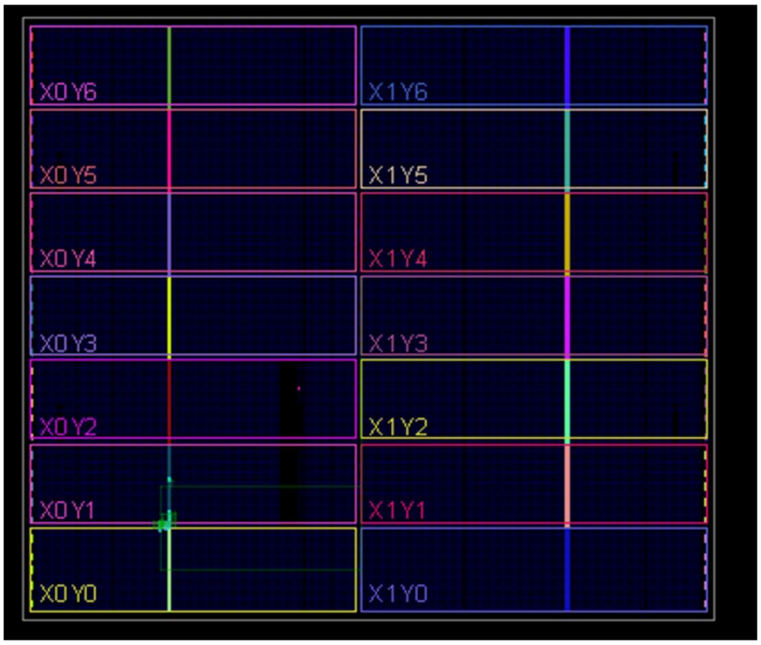


Figura 5.8: Esquema de la utilización de la FPGA para 10 reglas

El área de ocupación de nuestro sistema si usamos 10 reglas aumenta hasta 63 LUTs de las 204000 disponibles. En este caso el porcentaje de uso de pares LUT-FF completos es del 88% (Figura 5.9). La eficiencia en el uso de pares LUT-FF en las pruebas realizadas es debida a la óptima implementación del reconocedor de patrones. Los bloques básicos, los que reconocen un carácter, están compuestos por un biestable acompañado de una lógica muy sencilla, lo cual se ajusta al par LUT-FF [6].

```
Design Summary
--------------
Number of errors:      0
Number of warnings:   13
Slice Logic Utilization:
  Number of Slice Registers:                    63 out of 408,000    1%
    Number used as Flip Flops:                  63
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                         67 out of 204,000    1%
    Number used as logic:                       67 out of 204,000    1%
      Number using O6 output only:              62
      Number using O5 output only:               0
      Number using O5 and O6:                    5
      Number used as ROM:                        0
    Number used as Memory:                       0 out of  70,200    0%
    Number used exclusively as route-thrus:      0

Slice Logic Distribution:
  Number of occupied Slices:                    37 out of  51,000    1%
  Number of LUT Flip Flop pairs used:           67
    Number with an unused Flip Flop:             8 out of      67   11%
    Number with an unused LUT:                   0 out of      67    0%
    Number of fully used LUT-FF pairs:          59 out of      67   88%
    Number of unique control sets:               1
    Number of slice register sites lost
      to control set restrictions:               1 out of 408,000    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                        11 out of     600    1%
```

Figura 5.9: Uso de recursos para 10 regla

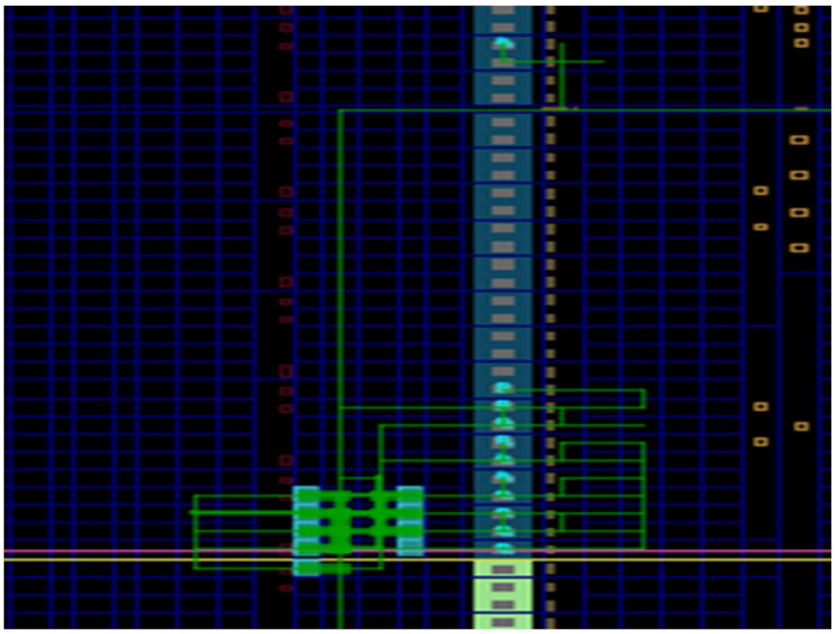


Figura 5.10: Esquema ampliado de la utilización de la FPGA para 10 reglas

100 Reglas

Al observar la Figura 5.11 comparándola con los resultados obtenidos en las pruebas anteriormente descritas, podemos comprobar que el área utilizada aumenta; sin embargo, este incremento no es significativo en relación con la capacidad total de la FPGA.

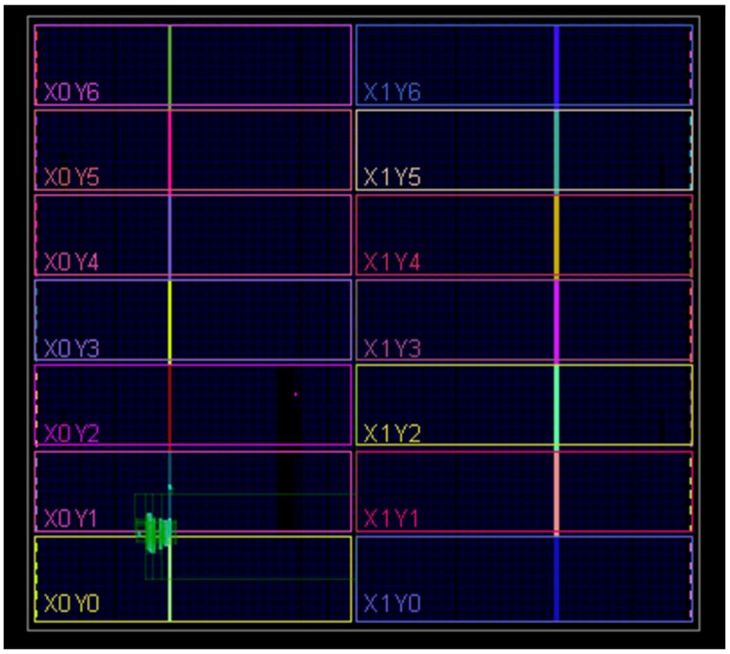


Figura 5.11: Esquema de la utilización de la FPGA para 100 reglas

Con 100 reglas observamos que el uso de LUTs se dispara hasta 562, pero sigue siendo una cantidad pequeña en comparación con las 204000 disponibles. El porcentaje de pares LUT-FF completos usados sigue siendo alto (Figura 5.12).

```
Design Summary
--------------
Number of errors:      0
Number of warnings:   13
Slice Logic Utilization:
  Number of Slice Registers:                 528 out of 408,000    1%
    Number used as Flip Flops:               528
    Number used as Latches:                    0
    Number used as Latch-thrus:                0
    Number used as AND/OR logics:              0
  Number of Slice LUTs:                      562 out of 204,000    1%
    Number used as logic:                    562 out of 204,000    1%
      Number using O6 output only:           552
      Number using O5 output only:             0
      Number using O5 and O6:                 10
      Number used as ROM:                      0
    Number used as Memory:                     0 out of  70,200    0%
    Number used exclusively as route-thrus:    0

Slice Logic Distribution:
  Number of occupied Slices:                 332 out of  51,000    1%
  Number of LUT Flip Flop pairs used:        562
    Number with an unused Flip Flop:          34 out of     562    6%
    Number with an unused LUT:                 0 out of     562    0%
    Number of fully used LUT-FF pairs:       528 out of     562   93%
    Number of unique control sets:             1
    Number of slice register sites lost
      to control set restrictions:             0 out of 408,000    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                      11 out of     600    1%
```

Figura 5.12: Uso de recursos para 100 reglas

En la Figura 5.13 se puede apreciar un gran incremento en el número de interconexiones y una mayor densidad de elementos utilizados.

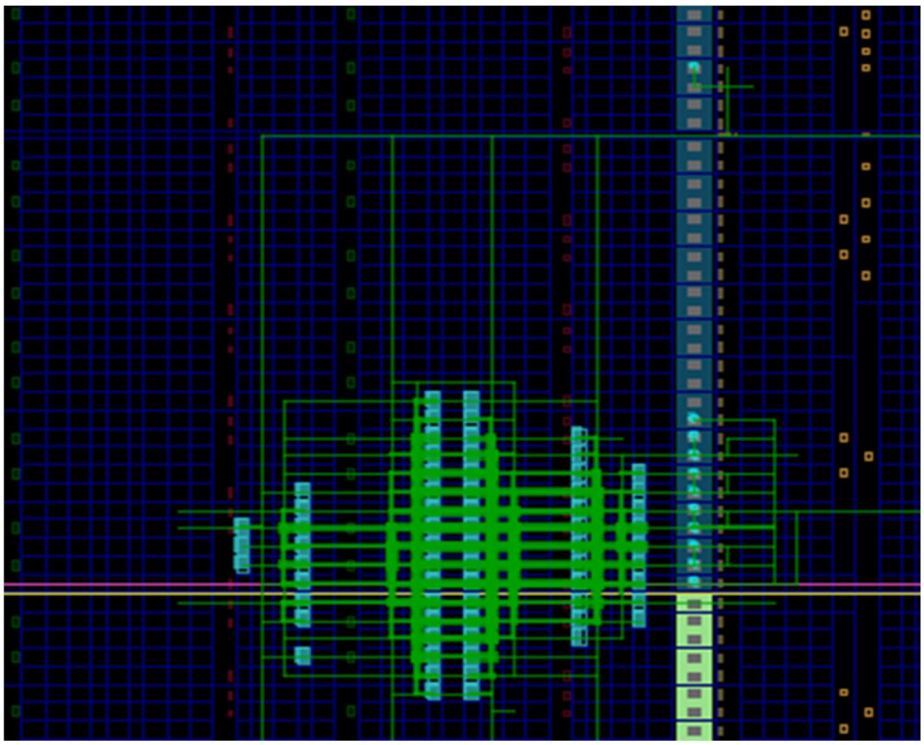


Figura 5.13: Esquema ampliado de la utilización de la FPGA para 100 reglas

1000 Reglas

Como se puede apreciar en la Figura 5.14 la ocupación de la placa al ejecutar 1000 reglas se incrementa notablemente, abarcando casi la mitad de un sector de la FPGA..

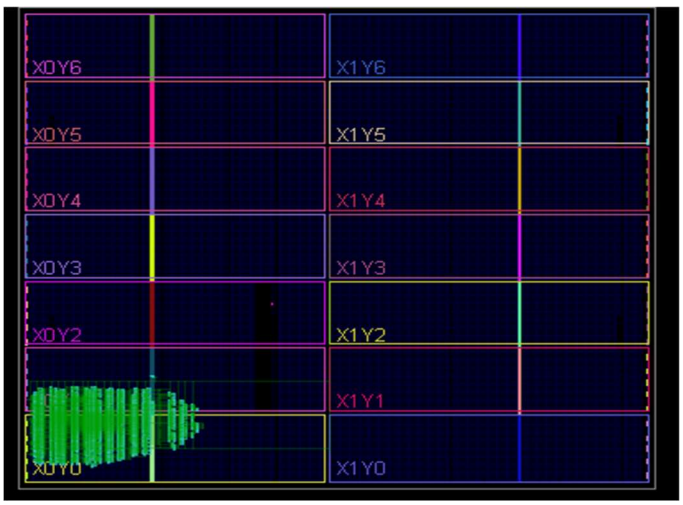


Figura 5.14: Esquema de la utilización de la FPGA para 1000 reglas

Si se observan las tasas de ocupación de los recursos se aprecia que el uso de LUTs aumenta hasta 4942, lo que supone el 2% de la placa (Figura 5.15). El resto de parámetros siguen manteniendo niveles similares a los obtenidos en las otras pruebas realizadas.

```
Slice Logic Utilization:
  Number of Slice Registers:                 4,456 out of 408,000    1%
    Number used as Flip Flops:               4,456
    Number used as Latches:                      0
    Number used as Latch-thrus:                  0
    Number used as AND/OR logics:                0
  Number of Slice LUTs:                      4,942 out of 204,000    2%
    Number used as logic:                    4,942 out of 204,000    2%
      Number using O6 output only:           4,815
      Number using O5 output only:               0
      Number using O5 and O6:                  127
      Number used as ROM:                        0
    Number used as Memory:                       0 out of  70,200    0%
    Number used exclusively as route-thrus:      0

Slice Logic Distribution:
  Number of occupied Slices:                 1,966 out of  51,000    3%
  Number of LUT Flip Flop pairs used:        4,942
    Number with an unused Flip Flop:           486 out of   4,942    9%
    Number with an unused LUT:                   0 out of   4,942    0%
    Number of fully used LUT-FF pairs:       4,456 out of   4,942   90%
    Number of unique control sets:               1
    Number of slice register sites lost
      to control set restrictions:               0 out of 408,000    0%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                      11 out of     600    1%
```

Figura 5.15: Uso de recursos para 1000 reglas

En cuanto a la distribución del área, la tendencia es la misma, creciente, con una red de conexiones amplia como se puede apreciar en la Figura 5.16**.**

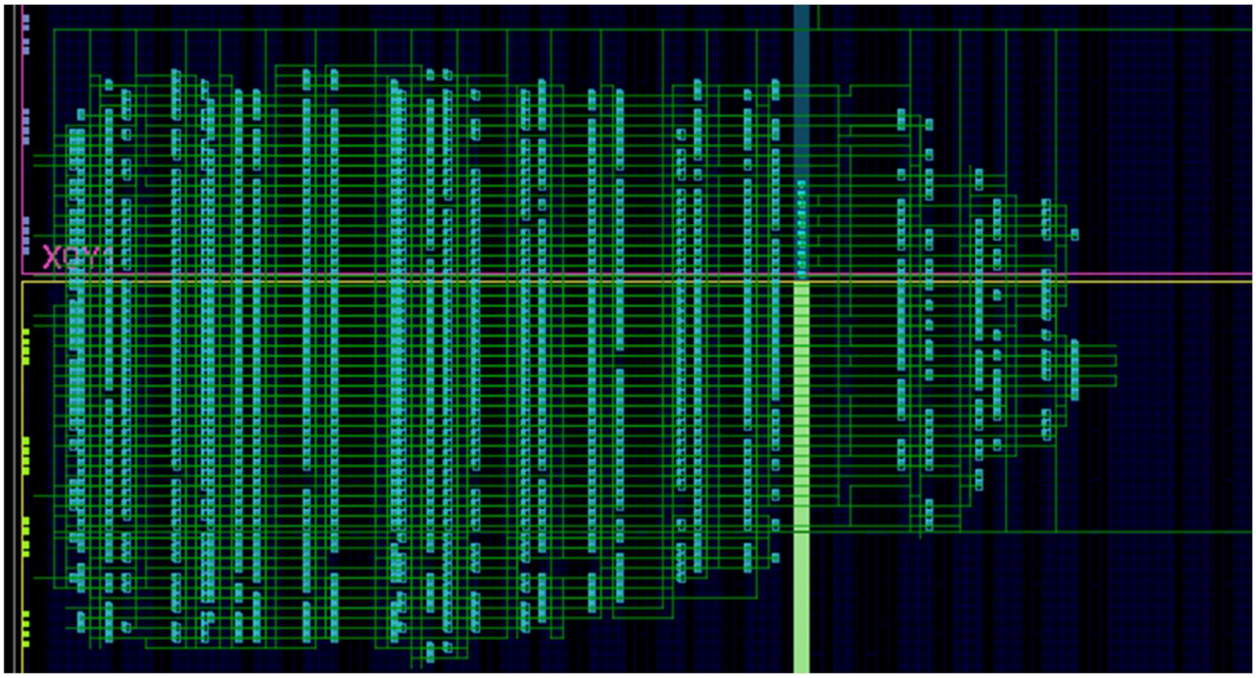


Figura 5.16: Esquema ampliado de la utilización de la FPGA para 1000 reglas

Otras 1000 reglas distintas

Decidimos realizar otra prueba con reglas completamente distintas para comprobar la consistencia de los datos anteriormente mostrados y ver las posibles diferencias que se podrían dar.

Como se observa en la Figura 5.17 y en la Figura 5.18, los resultados son muy parecidos. El área utilizada de la FPGA es similar y la red de conexiones, aunque se produce en otro sector de la FPGA, en cuanto a tamaño es igual.

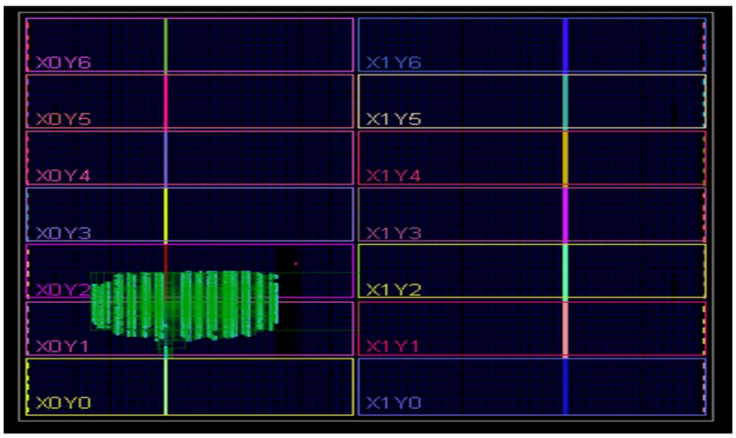


Figura 5.17: Esquema de la utilización de la FPGA para 1000 reglas distintas

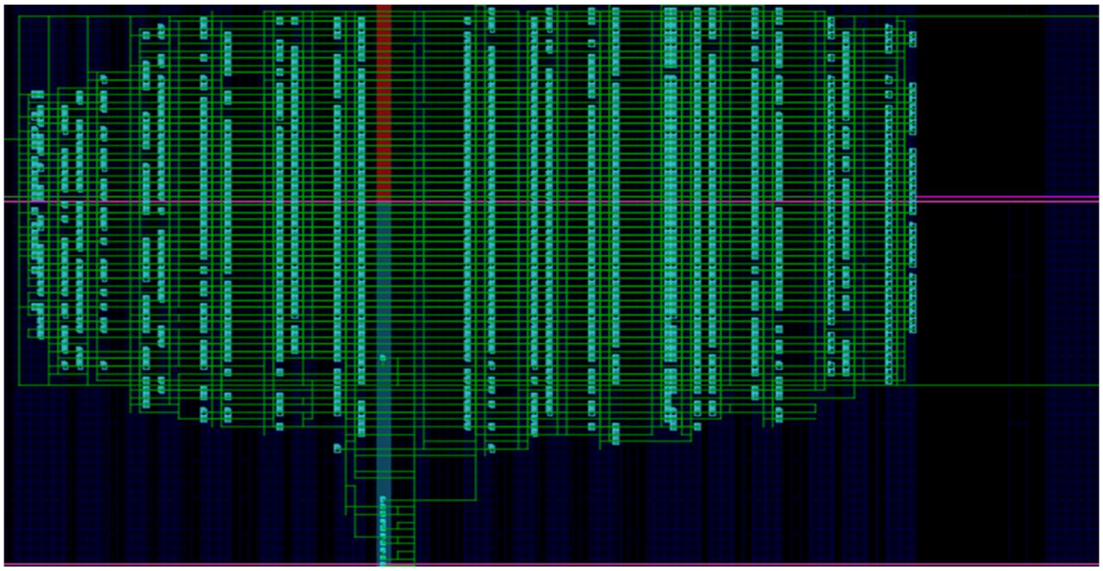


Figura 5.18: Esquema ampliado de la utilización de la FPGA para 1000 reglas distintas

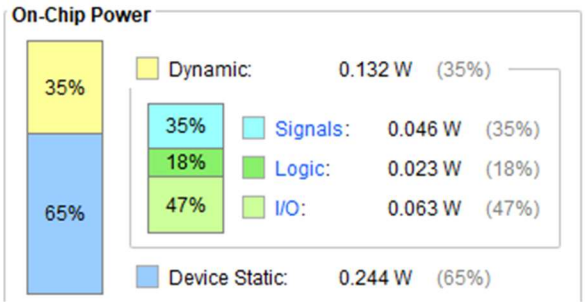
Se mantiene la tendencia en el uso de recursos, el número de LUTs usadas es similar a la otra prueba con 1000 reglas (Figura 5.19).

```
Slice Logic Utilization:
  Number of Slice Registers:                4,444 out of 408,000    1%
    Number used as Flip Flops:              4,444
    Number used as Latches:                     0
    Number used as Latch-thrus:                 0
    Number used as AND/OR logics:               0
  Number of Slice LUTs:                     4,949 out of 204,000    2%
    Number used as logic:                   4,949 out of 204,000    2%
      Number using O6 output only:          4,822
      Number using O5 output only:              0
      Number using O5 and O6:                 127
      Number used as ROM:                       0
    Number used as Memory:                      0 out of  70,200    0%
    Number used exclusively as route-thrus:     0

Slice Logic Distribution:
  Number of occupied Slices:                2,104 out of  51,000    4%
  Number of LUT Flip Flop pairs used:       4,949
    Number with an unused Flip Flop:          505 out of   4,949   10%
    Number with an unused LUT:                  0 out of   4,949    0%
    Number of fully used LUT-FF pairs:      4,444 out of   4,949   89%
    Number of unique control sets:              1
    Number of slice register sites lost
      to control set restrictions:              4 out of 408,000    1%

  A LUT Flip Flop pair for this architecture represents one LUT paired with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                       11 out of     600    1%
```

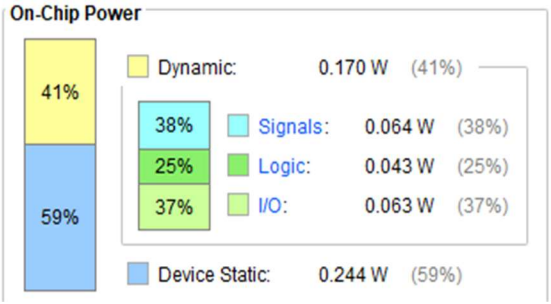
Figura 5.19: Uso de recursos para 1000 reglas distintas

En la Figura 5.20 podemos observar el consumo de potencia en cada una de las pruebas que se han realizado utilizando un número distinto de reglas. La primera corresponde con 1 regla, la segunda con 2 y así respectivamente.

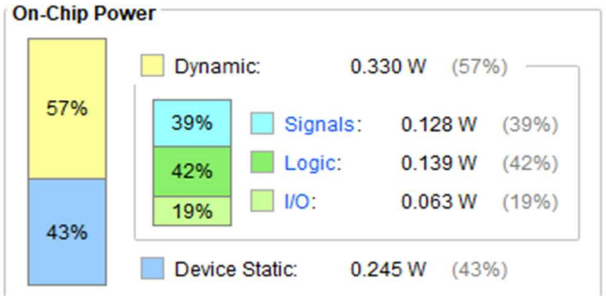
El consumo de I/O se mantiene estático en todas las pruebas menos en la de 1000 reglas. La mayor diferencia en cuanto al consumo de potencia se aprecia en las señales y los componentes lógicos. A medida que se incrementa el tamaño del sistema aumenta la lógica utilizada, no superando el consumo máximo de las pruebas realizadas el valor de 7.432 W.



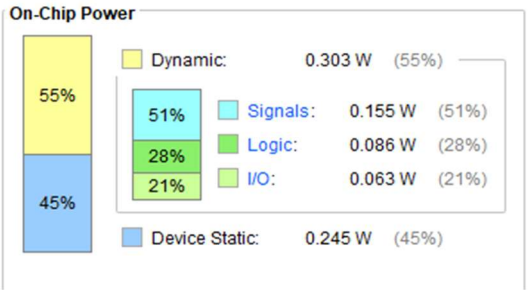
1 regla



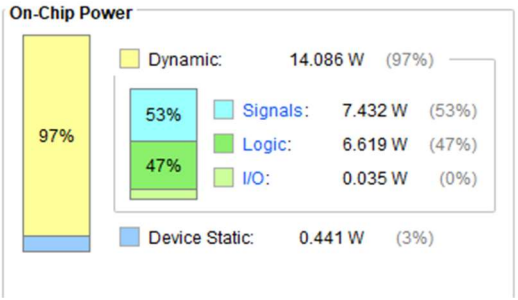
2 reglas



10 reglas



100 reglas



1000 reglas

Figura 5.20: Consumo de potencia de la FPGA

Capítulo 6

Conclusiones y Trabajo futuro

6.1 Conclusiones

En este Trabajo de Fin de Grado proponemos la implementación hardware de un Sistema de Detección de Intrusos de tipo NIDS basado en reglas. Para ello, partimos de una herramienta de reconocimiento de patrones diseñada para generar código VHDL que se sintetizará en una FPGA. Nuestro objetivo, es que el código hardware generado por esta herramienta implemente un IDS fiable y eficiente, añadiendo para ello los módulos de preprocesamiento que sean necesarios.

Durante el desarrollo de este trabajo hemos estudiado el funcionamiento y las características de los distintos tipos de Sistemas de Detección de Intrusos disponibles en la actualidad. También hemos tenido que examinar y comprender el formato de las reglas Snort o Suricata y de los paquetes de red asociados a los distintos protocolos, con el fin de poder reconocer ataques que se aprovechen de las vulnerabilidades de los mismos.

Uno de los trabajos más costosos ha sido el estudio de las expresiones regulares y de la notación polaca inversa. Este tipo de expresiones son la entrada del reconocedor de patrones, por lo que hemos tenido que transformar las reglas Snort o Suricata a expresiones regulares. Esto nos ha supuesto mucho esfuerzo debido a la gran diferencia entre ambas sintaxis.

Como hemos dicho, hemos desarrollado una serie de módulos de preprocesamiento que integramos con la herramienta de reconocimiento de patrones para conseguir un código hardware que implemente un IDS funcional sobre una FPGA. Estos módulos tienen dos funciones; por un lado, fragmentar y duplicar los paquetes de red para generar la entrada de datos del IDS hardware; por otro lado, convertir las reglas Snort o Suricata en expresiones regulares que se utilizan como entrada del reconocedor de patrones. El código asociado al módulo que realiza esta última función solo se tendrá que ejecutar cada vez que cambien las reglas, mientras que el que trabaja con los paquetes de red se tiene que ejecutar siempre.

Una vez implementado el IDS sobre la FPGA a partir del código VHDL generado por el sistema desarrollado en este TFG, hemos realizado distintas pruebas para probar tanto su correcto funcionamiento como su eficiencia, midiendo esta última en términos de tiempo de ejecución, uso de recursos y área de ocupación de la FPGA. En las distintas pruebas realizadas se ha variado el número de reglas que se utilizan como entrada del sistema IDS, con el fin de analizar la escalabilidad del sistema. Del análisis de los resultados obtenidos podemos concluir que la única limitación en el número de reglas que se pueden utilizar para implementar el IDS está impuesta por el área disponible en

la FPGA. Las pruebas se han realizado con la herramienta Vivado utilizando una placa Virtex-7. Esta herramienta permite estimar los tiempos de ejecución y el consumo de potencia y medir el área de ocupación de la FPGA.

Hemos medido dos tipos de tiempos, para distinguir entre el tiempo de ejecución del módulo que convierte las reglas en expresiones regulares, cuyo código se ejecuta únicamente cuando se modifican o añaden nuevas reglas, y el tiempo de ejecución del sistema IDS completo, que incluye la fragmentación de los paquetes y la búsqueda de patrones presentes en las reglas dentro de estos fragmentos.

Los últimos resultados obtenidos una vez realizada la implementación completa del Sistema de Detección de Intrusos muestran la viabilidad y rendimiento de nuestro sistema.

Como último apunte, nos gustaría destacara que para la realización de este trabajo han sido fundamentales los conocimientos adquiridos en las asignaturas de Redes y Redes y Seguridad estudiadas durante nuestra titulación de grado.

6.2 Trabajo futuro

La inclusión de un módulo transformador de reglas a expresiones regulares en el sistema heredado abre la puerta a numerosas optimizaciones. Durante el desarrollo se han encontrado puntos de mejora a explotar para una funcionalidad más completa y fluida de todo el sistema, entre los que podemos destacar:

Mejora del conversor de expresiones regulares a VHDL: un cambio de diseño para el soporte de métodos de expresiones regulares que actualmente no están consideradas en el sistema (véase sección 3.2.2). Destaca especialmente el funcionamiento de la negación "!" por su uso en numerosas reglas importantes en sistemas IDS y las llaves "{}", para reducir el número de falsos positivos pues las reglas solo buscarían en partes concretas del paquete y se podría evitar el uso del actual módulo de paquetes (véase sección 4.1.4).

***Buffer* software:** Para desarrollar el funcionamiento de acciones de las reglas, proponemos la creación de un *buffer* software en el que se almacenarán las acciones de las reglas. Con ello podríamos conseguir que el IDS al reconocer un patrón enviase la alerta a este *buffer* que contendría las acciones a realizar definidas por las reglas introducidas con anterioridad.

Unión con Wireshark: En lo relativo a la automatización, sería muy útil la conexión del sistema con la salida del programa wireshark para facilitar la captura de paquetes y

generar *testbenchs* automáticos para Vivado basados en estos paquetes. Esto crearía un sistema en tiempo real, un IDS completo y funcional.

Uso de varias placas FPGA: Siguiendo la búsqueda de la mejora de rendimiento, sería interesante evaluar cómo se comporta el sistema sobre varias FPGAs funcionando en paralelo.

Ampliación de tipos: Aunque ya estén implementados los campos más importantes de las reglas en cuanto a ataques se refiere, todo el desarrollo de las distintas opciones de Snort y Suricata sería una mejora para la detección de ataques más complejos.

Mejora del sistema: Realizar una mejora interna del sistema actual de forma que las opciones que son auxiliares de otras se conviertan en atributos, con lo que se consigue reducir el tamaño de la memoria y facilitar el funcionamiento del IDS.

Chapter 6

Conclusions and Future work

6.1 Conclusions

This Final Degree project has proposed the hardware implementation of an NIDS type IDS based on rules focused on search and test the efficiency and viability of this system.

We have reached the objectives exposed in chapter 1, obtaining a system composed by our program and the inherited system, the pattern matching tool.

During this project's development it was possible to deepen in the functioning and the properties of the different intrusion detection system currently available. Assure the efficiency of the Hardware IDS versus the Software IDS may suppose a challenge, for that it was done a study of this systems described in chapter 2.

The program we design and implement should meet the minimum requirements of being able to be used with the pattern matching tool already implemented[7]. This supposed an extensive study of regular expressions and the reverse polish notation besides the use of a FPGA to simulate the behavior of the global system.

Over multiple design and implementation iterations on our program result to the final version described in chapter 5, with which the tests explained in chapter 6 were made. The test were done for the obtention of experimental results in performance, use of resources and occupation areas.

All the test were done with a variable set of rules as input for the intrusion detection system in order to be able to show time variation, resources and FPGA occupation in relation with the increase in the amount of input data.

Performance tests evaluate our program written in Java and global efficiency of all the system measured in nanoseconds (ns).

For the resource tests we used the simulation tool Vivado [9], that allow to measure times over the system and on the board, also let us estimate the CPU, LUT and FPGA occupation area use. The chosen board was a Virtex-7. The main advantage that this tool has offered is the capacity of comparing between the results over different tests.

The last obtained results with a complete implementation of the intrusion detection system lay the ground about the viability of this systems.

6.2 Future work

The inclusion of a rule to regular expression transformer module on the inherited system opens the door to several optimizations. During development were found improvement points to exploit for a more complete and fluid functionality of all the system, among them we highlight:

Regular expression to VHDL converter improvement: a design change for regular expression methods support that now are not considered on the system (see section 3.2.2). Specially stands out the negation "!" functioning by its use in several important rules on IDS systems and keys "{}" to reduce the number of false positives because the rules only search for concretes package parts and could avoid the use of the actual package module (see section 5.1.4).

Software buffer: To develop the action rule functions, we propose the creation of a software buffer where the rule actions will be stored. With it we could make that the IDS when recognises a pattern could send the alert to this buffer that contains the action defined by the rules previously introduced to perform.

Wireshark union: In terms of automatization, it would be really helpful the conexion between our system and the output of wireshark to ease the package capture and generate automatic testbenchs for Vivado based on those packages. This would create a real-time system, a complete and functional IDS.

Use of several FPGA boards: Following the research of improving performance, it would be interesting to evaluate several FPGAs in parallel and study this improvement.

Type ampliation: Although the most important rule fields about attacks are already implemented, all the development of the different snort and suricata options would be an improvement for the detection of more complex attacks.

System improvement: Make an intern improvement of the actual system to make the auxiliar options to become attributes of the main ones, that would reduce memory size and ease the IDS performance.

Capítulo 7

División del trabajo

En este capítulo describiremos el trabajo que ha desempeñado cada miembro del equipo.

Para llevar este proyecto a cabo se ha realizado un trabajo colaborativo prácticamente en su totalidad entre ambos integrantes durante todo el proceso de investigación, desarrollo, implementación y documentación del proyecto. El trabajo se ha repartido de forma equitativa, trabajando prácticamente durante la mayor parte del tiempo. En este capítulo expondremos qué tareas ha desempeñado cada miembro.

7.1. Jorge González Soria

Al principio del proyecto Jorge aportó ideas sobre la implementación y la forma de llevarla a cabo.

Previamente, realizó una investigación sobre las bases de las expresiones regulares y las reglas Snort para poder comprender el alcance del proyecto. Estudió el trabajo previo junto con la herramienta de reconocimiento de patrones para poder comprender su funcionamiento y poder modelar una conexión eficiente entre este sistema y nuestro proyecto.

Tras toda la investigación previa que se realizó en conjunto con Félix, Jorge empezó a pensar el diseño del sistema y comenzó la implementación del Módulo de reglas.

Jorge diseñó parte del funcionamiento del Módulo de reglas y el conversor de reglas a expresión regular y se encargó de la parte principal de la implementación del Módulo de reglas.

Tras el desarrollo, generó las pruebas, *schematics* y los *testbench* para las simulaciones en Vivado y las mediciones de tiempos implementadas en Java para poder probar la eficiencia del sistema.

Para el desarrollo del proyecto, Jorge estudió de los artículos y la documentación referente a nuestro proyecto, los sistemas IDS y las expresiones regulares.

Junto con Félix, Jorge ha trabajado en la finalización del sistema y las diferentes mejoras que se han ido realizando (véase sección 4.1). Además, llevó a cabo una investigación de la viabilidad de las futuras ampliaciones que puede tener nuestro proyecto.

Y, por último, Jorge ha redactado de forma equitativa la memoria, trabajando más concretamente en los diferentes capítulos y secciones en las que ha colaborado.

7.2. Félix Villar González

Desde el comienzo del proyecto, Félix aportó ideas que ayudaron a la concepción del proyecto en su fase inicial. También realizó una investigación sobre las expresiones regulares y las reglas Snort y estudió el trabajo previo y la herramienta de reconocimiento de patrones.

Tras esta investigación que realizó en conjunto con Jorge, Félix planteó el sistema del módulo de reglas y parte del mecanismo para la transformación de regla Snort a expresión regular.

Desarrolló a su vez el funcionamiento del módulo de paquetes, la lógica que tenía que tener y su conexión con el resto del sistema.

Una vez validó, junto con Jorge, la lógica de este módulo realizó su diseño e implementación.

Para la realización del proyecto, utilizando los artículos recibidos por los tutores junto a su propia búsqueda, comprendió las bases de los sistemas IDS, las FPGA y las expresiones regulares. Aplicando esto, decidió qué opciones de las reglas eran posibles y útiles para ser convertidas a expresiones regulares.

Tras esto, realizó un análisis de los resultados obtenidos tras la ejecución de las pruebas y extrajo sus conclusiones.

Junto con Jorge, Félix ha trabajado en la finalización del sistema y las diferentes mejoras que se han ido realizando (véase sección 4.1).

Y, por último, Félix trabajó en la memoria, escribiendo los apartados en los cuales él había estado más involucrado y otros correspondientes a capítulos comunes que fueron divididos equitativamente entre los dos. Además, realizó una investigación sobre las futuras aplicaciones de nuestro sistema.

Bibliografía

[1] CCN-CERT. 2019. Ciberamenazas y Tendencias. CNI. Retrieve from https://www.ccn-cert.cni.es/informes/informes-ccn-cert-publicos.html

[2] Mahdi H. Moghaddam, Ricardo A. Calix, *Network Intrusion Detection Using a Hardware-Based Restricted Coulomb Energy Algorithm on a Cognitive Processor.*

[3] An FPGA-based Network Intrusion DetectionArchitecture. Abhishek Das *, Student Member, IEEE, David Nguyen, Student Member, IEEE, Joseph Zambreno, StudentMember, IEEE, Gokhan Memik, Member, IEEE and Alok Choudhary, Fellow, IEEE

[4] Assisting Network Intrusion Detection with Reconfigurable Hardware. B. L. Hutchings and R. Franklin and D. CarverDepartment of Electrical and Computer Engineering,Brigham Young University, Provo, UT 84602

[5] R. Sidhu and V. K. Prasanna, "Fast Regular Expression Matching Using FPGAs," The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), Rohnert Park, CA, USA, 2001, pp. 227-238.

[6] Evaluación de Expresiones Regulares sobre Hardware Reconfigurable. Ignacio Martín Santamaría, Sistemas Informáticos, Facultad de Informática, Universidad Complutense de Madrid.

[7] Mejora de la Evaluación de Expresiones Regulares Sobre Hardware Reconfigurable. Claudio Alejandro Muñoz Fernández, Sistemas Informáticos, Facultad de Informática, Universidad Complutense de Madrid.

[8] IDE Eclipse *Official Website* « https://www.eclipse.org».

[9] Vivado *Official Website* «https://www.xilinx.com/support.html#documentation ».

[10] Kali *Official Website* «https://www.kali.org/kali-linux-documentation/»

[11] *Wireshark Official Website* «https://www.wireshark.org/docs/»

[12] Shon Harris, Fernando Maymí, CISSP®, *All-in-One, Exam Guide,Seventh Edition*.

[13] Tamer AbuHmed, Abedelaziz Mohaisen, and DaeHun Nyang, *Deep Packet Inspection for Intrusion Detection Systems*: *A Survey. Information Security Research Laboratory,* Inha University, Incheon, Korea

[14] E. Villa, A. Zidaritz, M.D. Varga, G.Eschelbeck, M.K. Jones y M.J. McArdle, Active Firewall System and Methodology>>, Network Associates, Inc.,Santa Clara, CA (US)

[15] D. Denning, P.G. Neumann, « Requirements and Model for IDES – A Real-Time Intrusion Detection Expert System>>

[16] Martin Roesch. Snort -lightweight intrusion detection for networks. In 13th Systems Administration Conference, LISA '99, Seattle, WA, November 1999. www. usenix. orgl events/lisa99/full-papers/roesch/roeschJttmV

[17] Using Network Packet Generators and Snort Rules for Teaching Denial of Service Attacks Dr. Zouheir Trabelsi,Latifa Alketbi,United Arab Emirates University

[18] https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Suricata

[19] B.C. Brodie, R.K. Cytron, D.E. Taylor "A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching" (ISCA‟06)

[20] J. Divashree, H. Rajashekar , Kuruvilla Varghese "Dynamically Reconfigurable Regular Expression Matching Architecture", 2008 International Conference on Application-Specific Systems, Architectures and Processors

[21] http://proxacutor.free.fr/index.html

[22] http://www.tutorial-reports.com/computer-science/fpga

[23] Snort oficial Website <<https://www.snort.org/downloads/>>