Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Programa de Maestría en Computación

Reglas *OpenFlow*: Mejorando el Algoritmo de Detección de Interacciones

Propuesta sometida bajo consideración de la Escuela de Computación para optar por el grado de *Magister Scientiae* en Ciencias de la Computación

Oscar Mario Vásquez Leitón

Estudiante

Rodrigo Bogarín Navarro

Tutor

Cartago Enero, 2015

DEDICATORIA

Les dedico este trabajo a mi madre, mi hermano y mi novia, quienes fueron las personas que me ayudaron y alentaron durante todos estos años de estudio y arduo trabajo.

AGRADECIMIENTOS

Primero que todo, quiero agradecer a mi madre por haberme apoyado durante todo el proceso, y a mi padre por haberme enseñado desde pequeño que la diferencia entre los sueños y los logros está en el esfuerzo y la dedicación.

También quiero agradecer a mi novia, que durante los dos años y medio de estudio me apoyó y instó a continuar adelante, a pesar de que esto significaba que casi no iba a tener tiempo libre.

Quiero agradecer a mis profesores tutores, el doctor Carlos González, por ayudarme durante el proceso de elección del tema y elaboración del anteproyecto, y el máster Rodrigo Bogarín por su guía y ayuda a lo largo de la investigación, sus consejos fueron muy valiosos para lograr un trabajo de buena calidad.

Finalmente quiero agradecer a los señores Jesús Ulate y Marco Salazar, quienes me ayudaron incondicionalmente con las revisiones del anteproyecto y la tesis, ayuda sin la cual no hubiera podido completar el trabajo a tiempo.

ÍNDICE GENERAL

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ABREVIATURAS

FTE: Entrada de una tabla de flujo, del inglés *Flow Table Entry*.

TCAM: memoria de propósito específico que permite valores ternarios (0, 1 y X), del inglés *Ternary Content Addressable Memory*.

RESUMEN

En los últimos años se ha popularizado la programación dinámica de las redes, conocida como redes definidas por software. El protocolo de comunicaciones *OpenFlow* se ha convertido en uno de los más importantes dentro de las redes definidas por software,

Actualmente, la mayoría de las configuraciones se realiza manualmente mediante reglas que definen el comportamiento en los nodos de las redes, lo cual produce que la configuración sea un proceso expuesto a errores, dada la posible interacción no deseada que puede ocurrir entre las reglas. Bifulco y Schneider (2013), propusieron una definición para la interacción entre estas reglas y un algoritmo para detectarlas; pero indicaron que lo consideraban apto para aplicaciones en tiempo real con pocas reglas o para el tiempo de desarrollo.

En este trabajo se busca presentar una mejora en el rendimiento del algoritmo de detección de interacciones entre reglas *OpenFlow,* utilizando estructuras de datos bien conocidas para la reducción en la cantidad de operaciones necesarias, así como un procesamiento perezoso (análogo a la inicialización perezosa) de las comparaciones entre las reglas.

Los experimentos realizados mostraron que la mejora obtenida en el procesamiento de las reglas depende de la composición del conjunto de reglas, pero de forma general se obtuvo una mejora del 41%.

Adicionalmente, se pudo determinar que los cambios realizados en el algoritmo se presentan un enfoque alternativo que permite pensar en la utilización del algoritmo dentro de un *switch* y no sólo por parte de los desarrolladores cuando se están creando las aplicaciones como fue propuesto originalmente.

Palabras claves: *OpenFlow*; Algoritmos; Interacciones entre reglas, Redes definidas por software;

ABSTRACT

In recent years dynamic programming of networks has gained popularity, this is known as software defined networks. The communications protocol OpenFlow has become one of the most important amongst the software defined networks.

Nowadays most of the OpenFlow configurations are made by applications, and these applications are manually created by manipulating rules that define network behavior. This can be an error prone process, causing unwanted interactions between the rules. Bifulco and Scheider (2013) proposed a formal definition for the interactions between rules and a detection algorithm, but stated that the algorithm was suited only for OpenFlow applications with just a few hundred of rules or during development.

This work presents an improvement of the interactions detection algorithm performance, using well-known data structures to reduce the amount of required operations and a lazy comparison between rules (analogous to lazy initialization).

The experiments shown the achieved improvement depends on the composition of the rule set, but the overall improvement was of 41%, and it was determined that the changes in the algorithm represent an alternative approach that suggests the utilization of the algorithm inside an OpenFlow switch and not just for small applications or during development as originally conceived.

Keywords: OpenFlow; Algorithm; Rule interactions; Software defined networks.

www.tec.ac.cr

TEC | Tecnológico de Costa Rica
Programa de Maestría en Computación
Teléfono: 2550-2402

APROBACIÓN DE LA TESIS

"Reglas OpenFlow: Mejorando el Algoritmo de Detección de Interacciones"

TRIBUNAL EXAMINADOR

M.Sc. Rodrigo Bogarín Navarro
Profesor Asesor

Ph. D. José Enrique Araya Monge
Profesor Lector

M.Sc. Enrique Coen Alfaro
Profesional Externo

Dr. Roberto Cortés Morales
Coordinador del Programa
de Maestría en Computación

Enero, 2015

INTRODUCCIÓN

Durante varias décadas, las redes de computadoras han sido programadas estáticamente y se han creado muchas herramientas para facilitarlo; pero en los últimos años se ha ido popularizando la programación dinámica de las redes, conocido como redes definidas por software.

El protocolo de comunicaciones *OpenFlow* se ha convertido en uno de los más importantes dentro de las redes definidas por software, gracias a la funcionalidad que brinda para configurar y modificar el flujo de tráfico en las redes. Actualmente, la mayoría de las configuraciones se realizan por medio de aplicaciones, las cuales se desarrollan manualmente mediante reglas que definen el comportamiento de las redes. Esto conlleva que la configuración sea un proceso propenso a producir errores, debido a la posible interacción no deseada que puede ocurrir entre las reglas.

Bifulco y Schneider (2013) propusieron una definición para la interacción entre estas reglas y un algoritmo para detectarlas, pero indicaron que lo consideraban apto para aplicaciones en tiempo real con pocas reglas o bien, para el tiempo de desarrollo. Esta investigación se basa en este trabajo citado y propone un algoritmo alternativo, con el fin de adecuarlo a aplicaciones en tiempo real con una cantidad mucho mayor de reglas.

La programación de redes por medio del protocolo *OpenFlow* está en una etapa temprana de desarrollo, por lo que todavía existen muchas áreas que se están comenzando a explorar. Uno de los pasos en este desarrollo consiste enevaluar los problemas encontrados anteriormente y sus soluciones correspondientes, así como analizar la factibilidad en los problemas actuales. La investigación presentada sigue esta línea de trabajo, donde se analiza el estado del arte de la programación de las reglas de *firewall*. Estas reglas presentan un problema similar al de las reglas *OpenFlow*, pero con un domino considerablemente más pequeño. Por otra parte también se analiza el estado del arte sobre la programación de las reglas *OpenFlow*.

El algoritmo alternativo presentado, se basa en optimizaciones realizadas con algoritmos de detección de interacciones en reglas de *firewall*, basta señalar que en los experimentos realizados se demuestra la mejora lograda en el rendimiento, con respecto al algoritmo propuesto por Bifulco y Schneider (2013).

A continuación, se presenta un resumen contextual, el problema que se ataca y los objetivos principales de esta investigación.

En el capítulo n° 2 se presenta el marco conceptual, conformado por protocolo *OpenFlow*, la definición de las interacciones y la base del algoritmo original. Serealiza la revisión de literatura, donde se examina el estado en el arte de la programación en las reglas de *firewall* y reglas *OpenFlow*.

En el capítulo n° 3 se describe la metodología utilizada durante la investigación, además las mejoras realizadas al algoritmo, que resumen el trabajo en la elaboración del algoritmo alternativo.

El capítulo n°4 presenta la evaluación de los resultados y la validación de la hipótesis propuesta en esta investigación.

Finalmente, el capítulo n°5 presenta las conclusiones obtenidas de este trabajo y se sugieren ideas concretas sobre posibles mejoras al algoritmo.

1 GENERALIDADES DE LA INVESTIGACIÓN

1.1 MARCO CONTEXTUAL

La principal barrera para los investigadores a la hora de desarrollar nuevos protocolos de comunicación, ha sido las arquitecturas de los equipos, ya que éstas varían en cada proveedor, y no poseen plataformas estándar para que los investigadores experimenten con nuevas ideas.

Esta barrera se fundamenta en el hecho de que cada arquitectura desarrollada es parte de la propiedad intelectual de la empresa que desarrolló el equipo, y por supuesto los proveedores no están de acuerdo en mostrar la forma en que los diseñaron. Esta posición, tomada por los proveedores al decidir no abrir las interfaces dentro de los *switches* es fácil de entender, debido a que han gastado años desarrollando, perfeccionando los protocolos y algoritmos, por lo tanto temen que los nuevos experimentos destruyan su modelo de negocio. Es un hecho que las plataformas abiertas reducen la barrera de entrada para nuevos competidores, y algunas de ellas ya han sido desarrolladas, pero no tienen el desempeño ni la densidad necesaria para ser utilizadas en experimentos por los investigadores.

Para hacer más eficiente el proceso de investigación en nuevas tecnologías de redes, se han desarrollado diversas alternativas, una de ellas ha sido las redes virtuales dinámicas, las cuales son generadas a partir de protocolos programables como *OpenFlow*, éstas pueden reducir el costo de introducir nuevas ideas, incrementando la velocidad de innovación en la infraestructura de redes.

Cabe mencionar que en el 2008 se empezó a instalar redes basadas en este protocolo en la Universidad de Stanford (McKeown, Anderson, Balakrishnan, Parulkar, Peterson, Rexford y Turner, 2008).El propósito inicial del modelo de *OpenFlow* fue encontrar una manera en la cual los investigadores pudieran ejecutar pruebas experimentales de los protocolos de redes que utilizaban a diario. .

Luego, uno de los principales objetivos de la creación de *OpenFlow* fue mejorar la forma en la que se realizan las pruebas para las nuevas tecnologías de redes. Antes que apareciera esta tecnología, era difícil probar nuevos protocolos, pues resultaba complejo crear un ambiente de

pruebas que utilizara un protocolo no estándar de red. Por ésta razón muchos de los protocolos generados por la comunidad de investigadores se liberaban sin ser probados en su totalidad.

OpenFlow es un protocolo flexible que está basado en un *switch Ethernet* con una o más tablas de flujo internas y una interfaz estándar para agregar y remover entradas de dicha tabla. Además presenta ventajas comparado con la tecnología existente en el mercado actual, pues por un lado permite que los investigadores ejecuten experimentos en *switches* heterogéneos de una forma uniforme, con una alta densidad de puertos, y por otro lado los proveedores no necesitan exponer el trabajo interno de sus *switches*.

Entre las condiciones que se deben de cumplir a la hora de implementar un protocolo flexible están las siguientes:

- Tener un alto desempeño y un bajo costo de implementación.
- Ser capaz de soportar un amplio rango de investigación.
- Responsable de permitir tráfico experimental y tráfico real.
- Consistente con los protocolos cerrados de los proveedores.

La propuesta de *OpenFlow* es realmente sencilla, pues se aprovecha de la funcionalidad común que tienen todos los *switches* para leer y escribir las *TCAMs* (del inglés *Ternary Content Addressable Memories*), o tablas de flujo; por lo tanto utiliza este conjunto común de funciones de dichas tablas. Además, brinda un protocolo abierto para programar las tablas de flujo en diferentes *switches* y *routers*.

Un administrador de red, puede dividir el tráfico entre producción e investigación, y así los investigadores pueden controlar sus propios flujos, al seleccionar las rutas de sus paquetes y procesar su recibimiento. De este modo, los investigadores pueden crear nuevos protocolos, modelos de seguridad, esquemas de direccionamiento, incluso alternativas a IP. En la misma red, el tráfico de producción puede ser aislado y procesado en la misma forma que se hace actualmente sin mezclarse con los protocolos experimentales.

Un *switch OpenFlow* contiene al menos 3 partes, que son explicadas con más detalle en el siguiente capítulo, a saber:

1. Una tabla de flujo, con acciones asociadas a cada entrada de dicha tabla, para indicarle al *switch* como procesar el flujo,

2. Un canal seguro que conecta el *switch* con un control remoto de procesos, llamado controlador, permitiendo a los comandos y paquetes ser enviados entre el controlador y el *switch* usado.
3. El protocolo *OpenFlow*, que brinda una forma estándar y abierta para comunicar el controlador con el *switch*. Al definir este protocolo como una interfaz estándarse evita que los investigadores deban programar el *switch* (McKeown, Anderson, Balakrishnan, Parulkar, Peterson, Rexford y Turner, 2008).

Se pueden categorizar los *switches* en dos grupos, el primero contempla los *switches OpenFlow* dedicados, que no soportan procesamiento de capa 2 y capa 3, y el segundo abarca *OpenFlow* activo para propósito general. S on *switches* y *routers* en los cuales el protocolo *OpenFlow* ha sido agregado como una nueva característica. Algunos *switches* comerciales, *routers* y puntos de acceso van a ser mejorados al soportar *OpenFlow*; agregando las tablas de flujo, el canal de control y el protocolo *OpenFlow*. Generalmente la tabla de flujo va a reusar el hardware existente, como por ejemplo las TCAM, el canal seguro y el protocolo se van a ejecutar en el sistema operativo del *switch*.

El protocolo *Openflow* permite que este tipo de switches sean manejados por dos o más controladores para mejorar el desempeño y seguridad. El objetivo es permitir que los experimentos sean realizados en una red de producción existente, permitiendo el funcionamiento regular de las aplicaciones y el tráfico, aislando el experimental (procesado por las tablas de flujo) del tráfico de producción que es procesado por la arquitectura de capa 2 y capa 3.

Las tablas de flujo se encargan de buscar coincidencias entre los paquetes en las entradas que contienen (las entradas de ahora en adelante se denominarán FTEs por sus siglas en inglés); cuando ocurre una coincidencia se debe aplicar las instrucciones guardadas en la FTE, mientras que las tablas de grupo únicamente contienen su identificador e instrucciones para aplicar a los paquetes que son enviados a los grupos.

La programación de los *switches* se hace utilizando FTEs que son instaladas por medio del protocolo *OpenFlow*. Las FTEs contienen conjuntos de coincidencia, que determinan los flujos de red a los que la entrada aplica; y conjuntos de acciones que definen las decisiones de reenvío aplicadas a los flujos coincidentes, la prioridad que da el orden relativo entre las FTEs y un tiempo de espera que indica la expiración de la FTE.

Basado en lo anterior, y como mencionan los autores de (Bifulco y Schneider, 2013), el comportamiento del *switch OpenFlow* se puede definir como la combinación de las FTEs instaladas, ya que las FTEs que aplican sobre un mismo flujo pueden producir comportamientos distintos, dependiendo de las prioridades que tengan.

Bifulco y Schneider (2013) dieron una definición formal de las interacciones entre las FTEs instaladas en un *switch OpenFlow* y un algoritmo para su identificación. El "algoritmo de detección de interacciones" propuesto, consiste de la función principal que identifica la interacción entre dos FTEs y dos funciones auxiliares, los cuales calculan la relación entre los conjuntos de coincidencia y los conjuntos de acciones de dichas FTEs. Este algoritmo tiene complejidad $\theta(n)$ para la evaluación de una FTE contra un conjunto de n FTEs, y por cada FTE que es comparada se calculan las relaciones para todos sus campos de coincidencia (de ahora en adelante mencionados únicamente como campos).

1.2 PLANTEAMIENTO DEL PROBLEMA

Tradicionalmente, las redes de computadoras se han diseñado y configurado utilizando un paradigma estático, en el cual cada nodo de la red posee su propio conocimiento de la topología, lo cual requiere equipo de hardware con propósito específico, esto es complejo y costoso.

Actualmente está ocurriendo un cambio de paradigma, en donde se utiliza equipo de hardware con propósito general, y la configuración se hace por medio de *redes definidas por software* (Open Networking Foundation, 2014). El protocolo de comunicaciones *OpenFlow* (Open Networking Foundation, 2013) se basa en este nuevo paradigma, permitiendo separar el "plano de control" del "plano de los datos" en las redes, y esto permite mover la responsabilidad de tomar las decisiones del plano de control a un dispositivo aparte, típicamente llamado *Controlador OpenFlow*, dejando en los *switches OpenFlow* la responsabilidad del plano de los datos.

OpenFlow es un protocolo de comunicación que facilita la implementación de las SDN y se utiliza en las aplicaciones que se encargan de configurar los dispositivos de las redes. Este protocolo es relativamente reciente comparado con otros protocolos de comunicación como el modelo OSI, y todavía está en una fase de definición, por lo que es de esperar que no se haya desarrollado en todas sus áreas. Una de estos ámbitos nuevos es la programación de aplicaciones a base de reglas *OpenFlow*, una forma de programación manual que es de bajo nivel y por ende propensa a errores. .

Uno de los primeros pasos hacia el desarrollo de nuevas herramientas y metodologías que facilitan la programación de aplicaciones *OpenFlow*, se da en (Bifulco y Schneider, 2013), donde se propone una clasificación formal de las interacciones sus reglas dentro de un *switch OpenFlow*, y se provee un algoritmo para la detección de estas interacciones.

Según los autores, el algoritmo presentado tiene el problema de que la cantidad de reglas que se pueden procesar limita las aplicaciones del algoritmo, ya que este es funcional cuando se tienen como máximo unos pocos cientos de reglas dentro de un *switch*, y cuando la cantidad de reglas aumenta el tiempo de ejecución, lo hace poco viable para ser utilizado en *switches OpenFlow*.

Dusi, Bifulco, Gringoli, y Schneider (2014) muestran un estudio sobre la cantidad de reglas programadas en los *switches*, en diferentes escenarios reales de redes de acceso con muestras en el 2007 y en el 2009. En este estudio se puede observar que la cantidad promedio es aproximadamente 60.000 reglas, por lo que claramente el algoritmo original no se presta para este tipo de escenarios reales donde se supera por mucho los pocos cientos de reglas.

Por lo anterior, esta investigación propone utilizar una variante del algoritmo original con el objetivo de mejorar su rendimiento, cuando los conjuntos de reglas los superen en tamaño, y de esta forma hacerlo más conveniente en escenarios reales.

1.3 JUSTIFICACIÓN DEL PROBLEMA

En un Router o *switch* actual, la ruta de datos (¨*data path*¨) y las decisiones de enrutamiento de alto nivel (¨*control path*¨) ocurren en el mismo dispositivo. Usualmente los algoritmos de ruta de datos y control están implementados por circuitos electrónicos que no pueden ser fácilmente cambiados.

Un *switch OpenFlow* separa estas dos funciones. La porción relacionada con el manejo de la ruta de datos se mantiene implementada con circuitos electrónicos en el mismo dispositivo, sin embargo, las decisiones de alto nivel se trasladan a un controlador en el cuál los algoritmos de control están implementados por software fácilmente cambiable y adaptable.

Esto es similar a cuando se dio el cambio de los procesadores de propósito específico a los otros de propósito general, donde la programación de las aplicaciones juega un papel mucho más

importante, y manteniendo esa comparación, la programación de las reglas *OpenFlow* se podría comparar con la programación en lenguaje ensamblador, en donde este corresponda con las instrucciones del código máquina.

El trabajo presentado en (Bifulco y Schneider, 2013) propone una clasificación formal de las interacciones de las reglas dentro de un *switch OpenFlow*, y se provee un algoritmo para detectar las interacciones. Esto resulta muy útil para los desarrolladores de aplicaciones *OpenFlow*, pero tiene la limitante de que la cantidad de reglas que se pueden procesar limita las aplicaciones del algoritmo como herramienta en tiempo de ejecución, por lo que una mejora en el tiempo de ejecución del algoritmo sería un primer paso hacia su utilización como una herramienta en tiempo de ejecución.

La identificación de la interacción entre las reglas podría ayudar en la automatización del correspondiente manejo interno, por ejemplo, como se menciona en (Bifulco y Schneider, 2013) para detectar cuándo una nueva regla podría enmascarar o ser enmascarada por otra ya existente, para la simplificación de las reglas con base en los flujos de datos que concretan, para la división de las reglas y así evitar la redundancia, para tener un control avanzado sobre el tipo de reglas que se permiten agregar a las aplicaciones *OpenFlow*, etc.

El algoritmo tiene complejidad lineal para cada nuevo elemento, por lo que conforme el conjunto de nuevos elementos aumenta, la complejidad se aproxima a ser cuadrática, y como lo muestran los resultados en (Bifulco y Schneider, 2013) después de unos pocos cientos de reglas el algoritmo deja de ser adecuado para el uso dentro de un *switch OpenFlow*. Por esto los autores indican que es adecuado utilizarlo cuando se está desarrollando una aplicación y no así en tiempo real dentro de un *switch OpenFlow* con conjuntos de reglas mayores a los pocos cientos de reglas.

Como el algoritmo se encuentra en su versión inicial, la finalidad de este trabajo es buscar la mejora en el tiempo de ejecución con un conjunto considerable de reglas, al reducir el crecimiento de la cantidad de comparaciones que se tienen que ejecutar con respecto a la cantidad de reglas *OpenFlow* presentes en el *switch*.

El propósito sería habilitar su utilización en las aplicaciones *OpenFlow* tanto en los *switches*, como en nuevas aplicaciones aún no concebidas, que puedan utilizar la información provista por el algoritmo.

Además de lo mencionado con anterioridad, muchos de los *switches OpenFlow* actuales utilizan las *TCAMs* para guardar las reglas. Estas memorias consumen mucho espacio y energía comparado con el resto de las memorias utilizadas. Al implementar una variante del algoritmo que lo haga más conveniente para ser utilizado dentro de *swtiches OpenFlow*, se podría utilizar para simplificar el conjunto de reglas existente y de esta forma colaborar con el manejo de los recursos de hardware requeridos por las aplicaciones.

1.4 PREGUNTA DE INVESTIGACIÓN

¿Es posible mejorar el rendimiento del "algoritmo de detección de interacciones" entre reglas *OpenFlow*?

1.5 OBJETIVOS

1.5.1 Objetivo General

Proponer una variante al algoritmo de detección de interacciones de reglas *OpenFlow*, que bajo ciertas restricciones permita mejorar el tiempo de ejecución con respecto a la versión original.

1.5.2 Objetivos específicos

1. Implementar una variante del algoritmo que bajo ciertas restricciones, permita descartar reglas que no producirán ninguna interacción relevante para reducir la cantidad de comparaciones que tiene que realizar el algoritmo.
2. Determinar el tiempo requerido para procesar una regla *OpenFlow*, utilizando la variante del algoritmo y conjuntos base para reglas de diferente tamaño.
3. Comparar los resultados obtenidos en la ejecución del algoritmo original y su variante sobre los diferentes conjuntos de reglas, con el fin de identificar las mejoras que permite la variante del algoritmo.

1.6 ALCANCE

El alcance de la investigación comprende los siguientes puntos:

- La implementación de los algoritmos en un lenguaje de programación de alto nivel.
- La ejecución de los algoritmos por medio de una simulación en una PC convencional.

Los entregables definidos para esta investigación son los siguientes:

1. Los resultados de los tiempos de ejecución del algoritmo original, de acuerdo con los factores y niveles establecidos en el marco metodológico.
2. Los resultados de los tiempos de ejecución del algoritmo alternativo original, de acuerdo con los factores y niveles establecidos en el marco metodológico.
3. Comparación y análisis de los resultados de ambos algoritmos.

1.7 LIMITACIONES

La investigación no contempla los siguientes casos:

- Una implementación del algoritmo que considere más de un único *switch OpenFlow* para la comparación de FTEs.
- La ejecución de los algoritmos dentro de un *switch OpenFlow*.
- La generación de conjuntos de FTEs distintos a los mencionados en la sección de Marco Metodológico.
- La implementación de la variante del algoritmo incluye máscaras de valores totales o de subred, y no máscaras arbitrarias. La definición del ordenamiento de estos valores toma en cuenta las máscaras arbitrarias.
- Cualquier otro caso que no esté en los mencionados en las secciones de Alcance o Marco Metodológico.

2 MARCO CONCEPTUAL Y REVISIÓN DE LITERATURA

2.1 PROTOCOLO *OPENFLOW*

2.1.1 Switch *OpenFlow*

Según la especificación de *OpenFlow* 1.4.0 (Open Networking Foundation, 2013), la última versión disponible a junio del 2014; un *switch OpenFlow* consiste en una o más tablas de flujo y una tabla de grupo, que realizan búsquedas y reenvío de paquetes de red, y un canal *OpenFlow* hacia un controlador externo (Figura 2.1 Componentes principales de un *switch OpenFlow*). El único requerimiento para la comunicación entre un *switch* y el controlador consite en que el canal permita conectividad TCP/IP. La seguridad del canal no está incluida en la especificación.

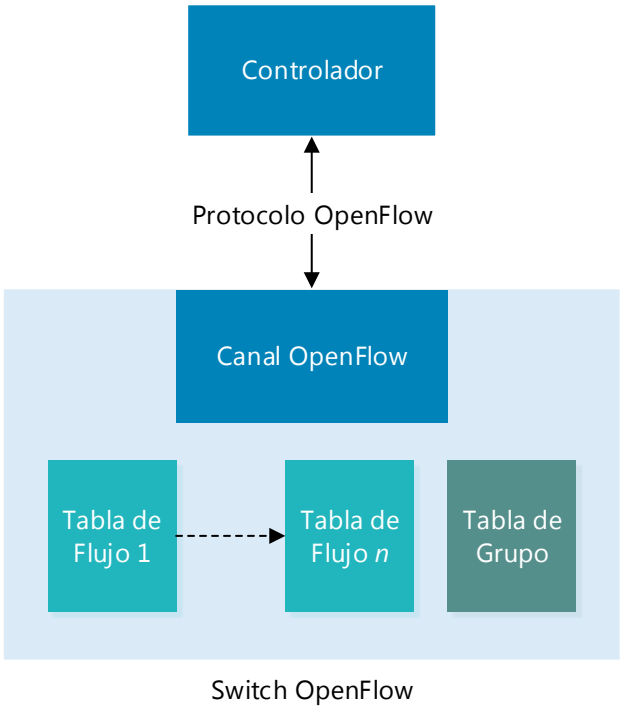

Figura 2.1 Componentes principales de un switch OpenFlow. Fuente: (Open Networking Foundation, 2013, p. 9)

La comunicación entre el *switch* y el controlador existe por medio del protocolo *OpenFlow*. Este protocolo se utiliza para agregar, actualizar y borrar FTEs en las tablas de flujo. Cada tabla de flujo en el *switch* contiene un conjunto de FTEs, y estas a su vez están conformadas por campos,

contadores (fuera del alcance de esta investigación) y un conjunto de instrucciones para aplicar a los paquetes coincidentes. En las siguientes subsecciones se verán estos conceptos más a fondo.

Un *switch OpenFlow* puede ser implementado en hardware o en software. La implementación por hardware usualmente provee mejor desempeño, ya que las búsquedas y el reenvío de paquetes se hace por medio de circuitería especializada para realizar estas tareas, mientras que en la implementación por software, estas tareas son realizadas por el procesador en el cual se está ejecutando el *switch*. Estas implementaciones por software tienen la ventaja que se pueden ejecutar en una computadora convencional, así como en diferentes sistemas operativos (Casado, 2014), y si se necesitara una actualización del *switch* basta con actualizar a una nueva versión del software.

2.1.2 Tablas *OpenFlow*

En *OpenFlow* existen dos tipos de tablas: las tablas de flujo y las tablas de grupo. Las primeras se encargan de buscar coincidencias entre los paquetes en las FTEs que contienen, además si ocurre una coincidencia aplicar las instrucciones guardadas en la FTE, En cambio, las tablas de grupo únicamente contienen su identificador e instrucciones para aplicar a los paquetes que son enviados a los grupos.

Las tablas pueden procesar los paquetes en un *pipeline*, su flujo está determinado por las acciones que se ejecutan en las FTEs coincidentes con el paquete. Las acciones pueden no redirigir el paquete a ninguna otra tabla de flujo o grupo por lo cual terminaría su procesamiento.

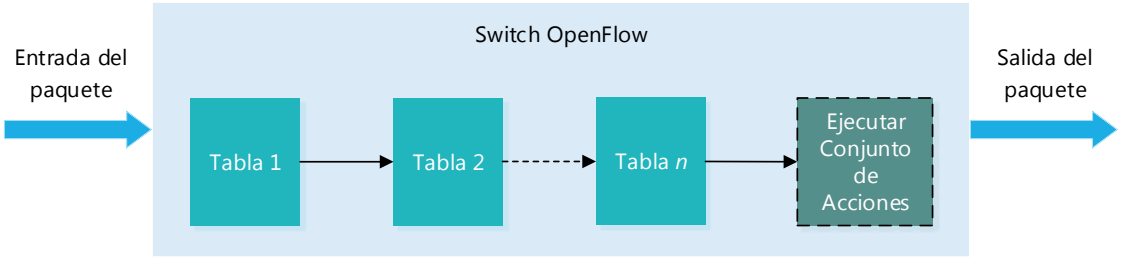

Figura 2.2 Paquete a través del pipeline de procesamiento. Fuente: (Open Networking Foundation, 2013, p. 14)

2.1.2.1 Tablas de Flujo

Las tablas de flujo están formadas por un conjunto de FTEs. No existe un límite teórico para el tamaño de las tablas, pero en la práctica el límite depende de los recursos disponibles dentro del *switch* (tanto de hardware como de software).

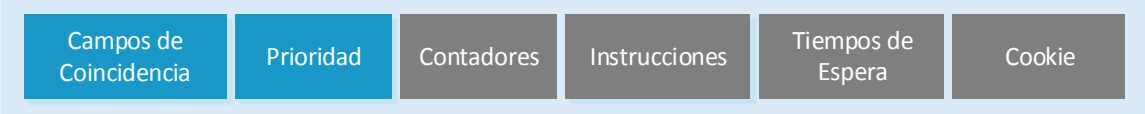

Figura 2.3 Componentes principales de una entrada de flujo. Fuente: (Open Networking Foundation, 2013, p. 15)

Cada FTE contiene los siguientes elementos (Figura 2.3 Componentes principales de una entrada de flujo:

- **Campos:** Son utilizados para buscar coincidencias con los paquetes. Formados por el puerto de ingreso, los encabezados de paquetes y opcionalmente metadatos especificados en una tabla previa.
- **Prioridad:** Es la precedencia que tiene la FTE dentro de la tabla de flujo.
- **Contadores:** Son actualizados cuando hay coincidencias con los paquetes.
- **Instrucciones:** Se utilizan para modificar el conjunto de acciones o el procesamiento en *pipeline*. Son las que definen el conjunto de acciones que se ejecutan sobre el paquete.
- **Tiempos de espera:** Definen el máximo tiempo de inactividad antes de que la FTE sea expirada por el *switch*.
- **Cookie:** Es el valor de datos opaco, elegido por el controlador. Puede ser utilizado por el controlador para filtrar estadísticas, modificar o eliminar FTEs, pero no se utilizan cuando se procesan los paquetes.

Una FTE es identificada de manera única por sus campos y por su prioridad dentro de una tabla de flujo. Esto no quiere decir que el protocolo defina un mecanismo para evitar reglas duplicadas, por lo que en caso de haber dos reglas con los mismos identificadores, la primera en detectar una coincidencia sería la que aplica las acciones.

La FTE que tiene comodines en todos sus campos (se omiten todos los campos al buscar coincidencias) y su prioridad es cero, es llamada "la FTE de fallo de tabla". Cada tabla de flujo debe soportar una FTE de fallo, porque esta describe cómo se debe procesar los paquetes que no tuvieron coincidencia en la tabla, como por ejemplo; enviando el paquete al controlador, botándolo o enviándolo a una de las siguientes tablas. Si no existe una FTE de fallo, los paquetes que no coincidan en una tabla son descartados.

2.1.2.2 Tablas de Grupo

Las tablas de grupo no tienen campos, únicamente conjuntos de acciones que se deben ejecutar a todos los paquetes que son enviados hacia los grupos por medio de las acciones. Estas tablas no forman parte de la investigación ya que se acceden directamente.

2.1.2.3 Coincidencia (Matching)

Cuando se recibe un paquete, el *switch OpenFlow* realiza una búsqueda en la primera tabla de flujo, y basado en el procesamiento en *pipeline*, podría realizar búsquedas en otras tablas. Un paquete coincide con una FTE si sus campos utilizados en las búsquedas coinciden con los campos de la FTE. Si el paquete tuviera varias coincidencias, se selecciona la FTE con mayor prioridad. El comportamiento cuando hay varias FTEs coincidentes en la misma prioridad está explícitamente indefinido.

Cuando un campo de la FTE contiene un comodín, este es omitido de la coincidencia. Algunos campos pueden tener comodines parciales por medio de máscaras de bits, donde un **0** significa "no importa" y un **1** significa que el bit debe coincidir exactamente. Las máscaras pueden ser arbitrarias o basadas en restricciones establecidas por los protocolos de red. Ciertos campos también contienen prerrequisitos, que pueden venir de restricciones de los protocolos de red, para poder ser utilizados en las coincidencias. En la tabla 1, se muestran los campos requeridos por el protocolo *OpenFlow* y su descripción, si admiten máscaras (y su tamaño en caso de permitirlas), los prerrequisitos para poder utilizar los campos.

Campo	**Descripción**	**Máscara**	**Bits**	**Prerequisito**
IN_PORT	Puerto de ingreso. Puede ser un puerto físico o uno lógico definido por el swtich	No	32	Ninguno
ETH_DST	Dirección destino Ethernet. Puede usar una máscara de bits arbitraria	Sí	48	Ninguno
ETH_SRC	Dirección fuente Ethernet. Puede usar una máscara de bits arbitraria	Sí	48	Ninguno
ETH_TYPE	Tipo Ethernet de la carga de datos del paquete.	No	16	Ninguno
IP_PROTO	Número de protocolo IPv4 ó IPv6	No	8	ETH TYPE=0x0800 ó ETH TYPE=0x86dd

IPV4_SRC	Dirección fuente IPv4. Puede usar una máscara de bits arbitraria o de subred	Sí	32	ETH TYPE=0x0800
IPV4_DST	Dirección destino IPv4. Puede usar una máscara de bits arbitraria o de subred	Sí	32	ETH TYPE=0x0800
IPV6_SRC	Dirección fuente IPv6. Puede usar una máscara de bits arbitraria o de subred	Sí	128	ETH TYPE=0x86dd
IPV6_DST	Dirección destino IPv6. Puede usar una máscara de bits arbitraria o de subred	Sí	128	ETH TYPE=0x86dd
TCP_SRC	Puerto TCP fuente	No	16	IP PROTO=6
TCP_DST	Puerto TCP destino	No	16	IP PROTO=6
UDP_SRC	Puerto UDP fuente	No	16	IP PROTO=17
UDP_DST	Puerto UDP destino	No	16	IP PROTO=17

Tabla 2.1 Campos requeridos del protocolo OpenFlow. Fuente: (Open Networking Foundation, 2013, p. 57)

2.1.2.4 Acciones

Las acciones indican lo que debe hacer el *switch OpenFlow* con los paquetes. Hay acciones requeridas por el protocolo y otras que son opcionales; las acciones requeridas son:

- **Salida:** reenviar el flujo de paquetes a través de un puerto específico, esto permite que los paquetes puedan ser enviados a través de la red.
- **Botar:** botar el flujo de paquetes (descartarlo).
- **Grupo:** procesar el flujo de paquetes en el grupo especificado.

Las demás acciones son opcionales y se enfocan en modificaciones que se realizan a los campos de los paquetes, pero están fuera del alcance de este trabajo.

2.1.2.5 Conjuntos de acciones

Los conjuntos de acciones están asociados con cada paquete. Las FTE utilizan las instrucciones para agregar acciones al conjunto de acciones del paquete. Los conjuntos de acciones contienen un máximo de una acción por cada tipo de acción.

2.2 ALGORITMO DE DETECCIÓN DE INTERACCIONES

Tal y como se vio en la sección anterior, la programación de los *switches* se hace utilizando FTEs que son instaladas por medio del protocolo *OpenFlow*. Las FTEs poseen un conjunto de coincidencias que definen los flujos de red a los que la entrada aplica, y un conjunto de acciones que define las decisiones de reenvío que deben ser aplicadas a los flujos coincidentes, la prioridad que da el orden relativo entre las FTEs y un tiempo de espera que indica la expiración de la FTE. Según la especificación de un *switch OpenFlow* (Open Networking Foundation, 2013), sólo la FTE de mayor prioridad que coincide con un paquete es aplicada sobre el mismo.

Basado en lo anterior, y como mencionan los autores de (Bifulco y Schneider, 2013) el comportamiento del *switch OpenFlow* se puede definir como la combinación de las FTEs instaladas; ya que una sola FTE puede no dar la imagen completa sobre el comportamiento con respecto a un flujo identificado por toda la coincidencia de esa FTE, ya que otras con prioridades más altas pueden introducir un comportamiento diferente.

Cuando se desarrolla una aplicación *OpenFlow* es posible que este problema surja, cuando se tiene que definir cuándo, dónde y cuáles FTEs se tienen que instalar en los *switches*, y aún más problemático si se están combinando aplicaciones o se están extendiendo a ellas.

Los autores en (Bifulco y Schneider, 2013) dieron una definición formal de las interacciones entre las FTEs instaladas en un *switch OpenFlow* y un algoritmo para su identificación, que son presentadas a continuación.

2.2.1 Definición de las interacciones

Una interacción es definida como: una relación particular entre dos FTEs. Estas interacciones pueden ser esperadas o inesperadas, ya que a la hora de crear las aplicaciones *OpenFlow* es posible que el programador no se dé cuenta que ocurren entre todas las FTEs que se están instalando, lo que podría provocar un error en la aplicación.

Para identificar la interacción entre dos FTEs, se definen las relaciones entre los conjuntos de coincidencia y entre los conjuntos de acciones, finalmente las combinaciones de estas determinan los tipos de interacción.

2.2.1.1 Relaciones de los conjuntos de coincidencia

Los conjuntos de coincidencia contienen campos que pueden tener comodines, y por la presencia de estos se pueden definir cuatro tipos de relaciones. La relación entre un campo de coincidencia c_1 y el campo de coincidencia c_2 puede ser una de las siguientes:

- **Disjunto** ($c_1 \neq c_2$): los valores de los campos son diferentes.
- **Igual** ($c_1 = c_2$): los valores de los campos son iguales.
- **Subconjunto** ($c_1 \subset c_2$): c_1 tiene un valor definido y c_2 tiene un comodín que abarca el valor de c_1.
- **Superconjunto** ($c_1 \supset c_2$): c_1 tiene un comodín que abarca el valor de c_2.

Con base en la definición de las relaciones entre los campos, se da la definición formal de las relaciones entre dos conjuntos de coincidencia. Esta definición es la base para la implementación del algoritmo. Las posibles relaciones entre dos conjuntos de coincidencia M_1 *y* M_2 son las siguientes:

- **Disjunto** ($M_1 \neq M_2$): si existe un campo en M_1 que es disjunto del campo correspondiente en M_2.
- **Exactamente coincidente** ($M_1 = M_2$): si todos los campos de M_1 son iguales a los campos correspondientes en M_2.
- **Subconjunto** ($M_1 \subset M_2$): M_1 es un subconjunto de M_2, si un campo de M_1 es subconjunto del campo correspondiente en M_2; y el resto de los campos de M_1 es igual o subconjunto de los campos correspondientes en M_2.
- **Superconjunto** ($M_1 \supset M_2$): M_1 es un superconjunto de M_2 si un campo de M_1 es superconjunto del campo correspondiente en M_2 y el resto de los campos de M_1 es igual o superconjunto de los campos correspondientes en M_2.
- **Correlacionado** ($M_1 \sim M_2$): M_1 está correlacionado con M_2 si al menos un campo de M_1 es superconjunto del campo correspondiente en M_2 y el resto de los campos de M_1 es igual o subconjunto de los campos correspondientes en M_2.

2.2.1.2 Relaciones de los conjuntos de acciones

Los conjuntos de acciones pueden contener cero o más acciones, y contienen valores que están asociados al tipo de acción que representan. Las relación entre dos acciones a_1 y a_2 puede ser cualquiera de las siguientes:

- **Disjunta** ($a_1 \neq a_2$): a_1 y a_2 tienen tipos diferentes.
- **Igual** ($a_1 = a_2$): a_1 y a_2 tienen los mismos tipos y valores.
- **Relacionada** ($a_1 \sim a_2$): a_1 y a_2 tienen los mismos tipos pero tienen diferentes valores.

Con base en estas relaciones se definen las posibles relaciones entre un conjunto de acciones A_1 y un conjunto de acciones A_2:

- **Disjunto** ($A_1 \neq A_2$): A_1 es disjunto de A_2, si todas las acciones de A_1 son disjuntas con respecto de todas las acciones de A_2.
- **Igual** ($A_1 = A_2$): A_1 es igual a A_2, si todas las acciones contenidas en A_1 son iguales acciones contenidas en A_2, y la cantidad de acciones de A_1 es igual a la cantidad de acciones en A_2.
- **Subconjunto** ($A_1 \subset A_2$): A_1 es subconjunto de A_2, si todas las acciones contenidas en A_1 son iguales acciones contenidas en A_2, y la cantidad de acciones de A_1 es menor que la cantidad de acciones en A_2.
- **Superconjunto** ($A_1 \supset A_2$): A_1 es superconjunto de A_2, si todas las acciones contenidas en A_2 son iguales acciones contenidas en A_1, y si la cantidad de acciones de A_1 es mayor que la cantidad de acciones en A_2.
- **Relacionado** ($A_1 \sim A_2$): A_1 está relacionado con A_2 si:
    - Existe al menos una acción en A_1 que está relacionada con una acción en A_2.
    - A_1 y A_2 tienen la misma cantidad de acciones y existe un subconjunto no vacío de acciones de A_1 que es igual a un subconjunto no vacío de acciones de A_2, y existe un subconjunto no vacío de acciones de A_1 que es disjunto de un subconjunto no vacío de acciones de A_2.

2.2.1.3 Tipos de interacción

Los tipos de interacciones entre dos FTEs están definidos con base en las relaciones de los conjuntos de coincidencia, las relaciones de los conjuntos de acciones y las prioridades de las FTEs. Si se tiene una FTE F_x con un conjunto de coincidencia M_x y un conjunto de acciones A_x, y una FTE F_y con un conjunto de coincidencia M_y y un conjunto de acciones A_y, y suponiendo que la prioridad de F_x es siempre menor que la prioridad de F_y la interacción entre F_x y F_y puede ser una de las siguientes:

- **Duplicación**: si dos FTEs tienen la misma prioridad y tienen los mismos conjuntos de coincidencia y de acciones.

- **Redundancia**: las FTEs redundantes tienen el mismo efecto sobre el conjunto de flujos que coinciden con las dos FTEs. Si no existe interacción con otras FTEs sería posible eliminar una de las FTEs redundante.
- **Generalización**: las dos FTEs tienen diferentes conjuntos de acciones pero el conjunto de coincidencia de F_x es un superconjunto del conjunto de coincidencia de F_y. En este caso, a los flujos que coinciden con $F_x \cap F_y$ se les aplicará las acciones de A_y, y a los flujos que coinciden con $F_x - F_y$ se les aplicará las acciones de A_x.
- **Sombreado**: F_x y F_y tienen conjuntos de acciones diferentes y si F_x es sombreada por F_y, el conjunto de acciones de F_x nunca es aplicado a los flujos en que coincide.
- **Correlación**: las FTEs tienen conjuntos de coincidencia diferentes pero la intersección de estos conjuntos no es vacía, entonces a los flujos que están en la intersección se les aplicará las acciones de A_y. Es diferente del sombreado porque hay flujos que sólo coinciden con M_x y a estos se les aplicará las acciones de A_x.
- **Inclusión**: es parecida al sombreado, pero en este caso el conjunto de acciones A_x es subconjunto de A_y por lo que las acciones de F_x también son aplicadas indirectamente.
- **Extensión**: es similar a la generalización, pero A_x es superconjunto de A_y, por ello cuando se apliquen las acciones de A_x se aplicarán también las acciones de A_y.

La tabla 2 muestra la definición formal de las interacciones dadas en (Bifulco y Schneider, 2013):

Conjunto de Coincidencia	Conjunto de Acciones	Prioridad
Duplicación		
$\boldsymbol{M_x = M_y}$	$A_x = A_y$	$prio(F_x) = prio(F_y)$
Redundancia		
$\boldsymbol{M_x \subset M_y}$	$A_x = A_y$	$prio(F_x) < prio(F_y)$
$\boldsymbol{M_x \supset M_y}$	$A_x = A_y$	$prio(F_x) < prio(F_y)$
$\boldsymbol{M_x \sim M_y}$	$A_x = A_y$	$prio(F_x) < prio(F_y)$
$\boldsymbol{M_x = M_y}$	$A_x = A_y$	$prio(F_x) < prio(F_y)$
Generalización		
$\boldsymbol{M_x \supset M_y}$	$A_x \neq A_y$	$prio(F_x) < prio(F_y)$
$\boldsymbol{M_x \supset M_y}$	$A_x \sim A_y$	$prio(F_x) < prio(F_y)$
$\boldsymbol{M_x \supset M_y}$	$A_x \subset A_y$	$prio(F_x) < prio(F_y)$

Sombreado			
	$\boldsymbol{M_x \subset M_y}$	$A_x \neq A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x \subset M_y}$	$A_x \sim A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x \subset M_y}$	$A_x \supset A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x = M_y}$	$A_x \neq A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x = M_y}$	$A_x \sim A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x = M_y}$	$A_x \supset A_y$	$prio(F_x) < prio(F_y)$
Correlación			
	$\boldsymbol{M_x \sim M_y}$	$A_x \neq A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x \sim M_y}$	$A_x \sim A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x \sim M_y}$	$A_x \supset A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x \sim M_y}$	$A_x \subset A_y$	$prio(F_x) < prio(F_y)$
Inclusión			
	$\boldsymbol{M_x = M_y}$	$A_x \subset A_y$	$prio(F_x) < prio(F_y)$
	$\boldsymbol{M_x \subset M_y}$	$A_x \subset A_y$	$prio(F_x) < prio(F_y)$
Extensión			
	$\boldsymbol{M_x \supset M_y}$	$A_x \supset A_y$	$prio(F_x) < prio(F_y)$

Tabla 2.2 Definición formal de las interacciones. Fuente: (Bifulco y Schneider, 2013, p. 3)

2.2.2 Algoritmo

El algoritmo de detección de interacciones propuesto en (Bifulco y Schneider, 2013), recibe dos FTEs F_x y F_y, suponiendo que la prioridad de F_x es menor a la prioridad de F_y. Se espera que las estructuras de datos que representan a F_x y F_y contengan la información necesaria para calcular las relaciones en los conjuntos de coincidencia y acciones.

El algoritmo consiste en la función principal que identifica la interacción entre dos FTEs y dos funciones auxiliares que calculan la relación entre los conjuntos de coincidencia y los conjuntos de acciones de dichas FTEs.

2.2.2.1 Pseudocódigo – Relaciones de coincidencia

El algoritmo para la identificación de la relación entre los conjuntos de coincidencia, se muestra en la Figura 2.4.

```
relacion_coincidencia(Rx,Ry)

relacion ← indeterminada
relaciones_de_campos ← comparar_campos(Rx,Ry)

for campo in campos_de_coincidencia do
    if relaciones_de_campos[campo] = igual then
        if relacion = indeterminada then
            relacion ← exacto
        end if
    else if relaciones_de_campos[campo] = superconjunto then
        if relacion = subconjunto or relacion = correlacionado then
            relacion ← correlacionado
        else if relacion != correlacionado then
            relacion ← superconjunto
        end if
    else if relaciones_de_campos[campo] = subconjunto then
        if relacion = superconjunto or relacion = correlacionado then
            relacion ← correlacionado
        else if relacion != correlacionado then
            relacion ← subconjunto
        end if
    else
    relacion ← disjunto
    end if
end for

return relacion
```

Figura 2.4 Relaciones de coincidencia. Fuente: (Bifulco y Schneider, 2013)

2.2.2.2 Pseudocódigo – Relaciones de acciones

El algoritmo para la identificación de la relación entre los conjuntos de acciones se muestra en la Figura 2.5.

```
relacion_acciones(Rx,Ry)

relacion ← indeterminada
relaciones_de_acciones ← comparar_acciones(Rx,Ry)

cantidad_acciones_x ← cantidad_acciones(Rx)
cantidad_acciones_y ← cantidad_acciones(Ry)

cantidad_acciones_igual ← falso
if cantidad_acciones_x = cantidad_acciones_y then
    cantidad_acciones_igual ← verdadero

for accion in conjunto_acciones do
    if relaciones_de_acciones[accion] = igual then
        if relacion = igual or relacion = indeterminada then
            relacion ← igual
        else if cantidad_acciones_igual = verdadero then
            relacion ← relacionada
        end if
    else if relaciones_de_acciones[accion] = relacionada then
            relacion ← relacionada
    else if relaciones_de_acciones[accion] = disjunta then
        if relacion = disjunto or relacion = indeterminada then
            relacion ← disjunto
        else if cantidad_acciones_igual = verdadero then
            relacion ← relacionada
        end if
    end if
end for

if relacion = igual and cantidad_acciones_igual = falso then
    if  cantidad_acciones_x < cantidad_acciones_y then
        relacion ← subconjunto
    else if cantidad_acciones_x > cantidad_acciones_y then
        relacion ← superconjunto
    end if
end if

return relacion
```

Figura 2.5 Relaciones de acciones. Fuente: (Bifulco y Schneider, 2013)

2.2.2.3 Pseudocódigo – Detección de interacciones

El algoritmo para la detección de interacciones se muestra en la Figura 2.6.

```
deteccion_interaccion(Rx,Ry)

interaccion ← Ninguna
relacion_coincidencia ← relacion_coincidencia(Rx,Ry)
relacion_acciones ← relacion_acciones(Rx,Ry)

if prioridad(Rx) = prioridad(Ry) and relacion_coincidencia = exacto and relacion_ac-
ciones = igual
then
    interaccion ← duplicacion
else if relacion_coincidencia != disjunto then
    if relacion_coincidencia = correlacionado then
        if relacion_acciones = igual then
            interaccion ← redundancia
        else
            interaccion ← correlacion
        end if

    else if relacion_coincidencia = superconjunto then
        if relacion_acciones = igual then
            interaccion ← redundancia
        else if relacion_acciones = superconjunto then
            interaccion ← extension
        else
            interaccion ← generalization
        end if

    else if relacion_coincidencia = exacto then
        if relacion_acciones = igual then
            interaccion ← redundancia
        else if relacion_acciones = subconjunto then
            interaccion ← inclusion
        else
            interaccion ← sombreado
        end if

    else if relacion_coincidencia = subconjunto then
        if relacion_acciones = igual then
            interaccion ← redundancia
        else if relacion_acciones = subconjunto then
            interaccion ← inclusion
        else
            interaccion ← sombreado
        end if
    end if
end if

return interaccion
```

Figura 2.6 Detección de interacciones. Fuente: (Bifulco y Schneider, 2013)

2.2.3 Complejidad

Cada vez que el algoritmo se ejecuta para detectar las interacciones de una FTE *f* con respecto a un conjunto existente de FTEs *C,* se itera sobre todas las FTEs de *C*, y se compara cada uno de los campos de coincidencia de *f* que contienen un valor diferente de un comodín. El algoritmo no toma en cuenta si los valores de las FTEs que pertenecen a *C* potencialmente podrían producir una interacción o no, entonces se ejecuta sobre el conjunto completo indistintamente.

Basados en lo anterior, si el conjunto *C* contiene *n* FTEs, se puede interpretar la complejidad del algoritmo como $\theta(n)$, ya que sin importar los contenidos de *C* o de *A*, el algoritmo siempre se va a ejecutar sobre las *n* FTEs que pertenecen a *C*.

Si se tiene un conjunto de FTEs *D* que contiene *m* FTEs, para agregarlo dentro de un *switch*, el algoritmo compara cada una de las FTEs de *D* contra cada una de las FTEs de *C,* por lo que la complejidad estaría dada por $\theta(n*m)$. Cuando *m* se aproximara a *n,* la complejidad se aproximaría a $\theta(n^2)$.

2.2.4 Casos de uso del ADI

El algoritmo de detección de interacciones representa un esfuerzo hacia la mejora en las metodologías del desarrollo y herramientas para la programación de las redes *OpenFlow*; ya que permite obtener información de los efectos colaterales que pueden existir en los flujos de red al manipular las FTEs que se instalan en un *switch OpenFlow*.

Como se menciona en (Bifulco y Schneider, 2013), algunos de los posibles usos del algoritmo son:

- Detección de FTEs duplicadas o redundantes.
- Detección de interacciones no esperadas que modifiquen los flujos de red.
- Comparación de un conjunto de FTEs existente contra un conjunto de FTEs predefinido como un control de admisión de FTEs.
- Optimización de las FTEs instaladas en un *switch OpenFlow*.

Es posible que surjan casos de uso que requieran un análisis semántico más complejo en la administración de las FTEs que utilicen como base este algoritmo, como por ejemplo un análisis avanzado de las FTEs en los controladores *OpenFlow* para automatizar de alguna forma la

administración de las FTEso: reordenando las prioridades de las FTEs para evitar sombreado, separándolas para evitar redundancia, entre otros.

2.3 REVISIÓN DE LITERATURA

La programación de las redes de computadoras a través de *OpenFlow*, es un área que lleva pocos años de investigación especializada al ser una tecnología reciente. Sin embargo, el problema sobre la optimización y detección de conflictos en reglas relacionadas con las políticas de red, no es un problema totalmente nuevo.

El problema de detección de conflictos o interacciones de reglas *OpenFlow* es similar a las dificultades que han existido con las reglas de los *firewall* durante años ya que las reglas de firewall representan un subconjunto de las reglas *OpenFlow*. Las reglas de firewall se utilizan para permitir o denegar el paso de paquetes de red de acuerdo a su información relacionada al direccionamiento.

Un ejemplo de la necesidad para identificar las relaciones en las reglas de firewall es, por ejemplo, si se usa una regla que permita el paso de paquetes hacia una subred, y que posteriormente se agregue otra regla de mayor prioridad que deniegue el paso de paquetes hacia un superconjunto de esa subred, haciendo que la primera sea ocultada y previniendo su ejecución.

El dominio de las reglas *OpenFlow* es más amplio, por lo que las soluciones específicas aplicadas a las reglas de *firewall* no necesariamente implican soluciones para el problema con las de *OpenFlow*. Similarmente, el análisis de las redes presenta problemas similares, como la identificación de problemas en los flujos definidos en la configuración de las redes, y los lenguajes de alto nivel para la programación de reglas *OpenFlow*. También se enfrentan a este problema al traducir construcciones de un nivel mayor de abstracción a un conjunto de reglas *OpenFlow*.

A continuación se exploran algunos trabajos relacionados a la optimización o identificación de conflictos de reglas de firewall y de reglas *OpenFlow*, en análisis de la redes y de lenguajes de alto nivel para programación de reglas *OpenFlow*. Se describen sus características más importantes y su relación con el fin de discutir y exponer al lector los trabajos realizados en estas áreas.

2.3.1 Reglas de firewall

Uno de los trabajos más referenciados en el tema de las reglas de firewall es el de Ehab S. Al-Shaer y Hamed, tanto en artículos como en patentes, ya que fue el primero en definir un análisis

completo de las anomalías de las reglas de firewall, por lo que se considera el artículo clásico sobre el tema.

Los autores sugieren un modelo, un conjunto de técnicas y algoritmos para detectar las anomalías en las reglas que puedan producir conflictos. El modelo sugerido está basado en un árbol de políticas de firewall donde pueden clasificar todas las reglas (Figura 2.7 Árbol de políticas de firewall. Fuente: (Al-Shaer y Hamed 2003, p. 21)).

El algoritmo propuesto recorre los caminos de las reglas en el árbol y si los caminos de las reglas coinciden, entonces se tiene una potencial anomalía. Como parte de la investigación, los autores definieron formalmente las siguientes relaciones entre reglas.: disjunta, coincidencia exacta, coincidencia inclusiva (de esta se deriva la relación de subconjunto y de superconjunto), parcialmente disjunta y correlacionada. Además definieron las anomalías entre reglas: sombreado, correlación, generalización y redundancia. En otro de sus trabajos, la investigación muestra que este trabajo es útil tanto en ambientes académicos como industriales (Ehab Al-Shaer y Hamed, 2002).

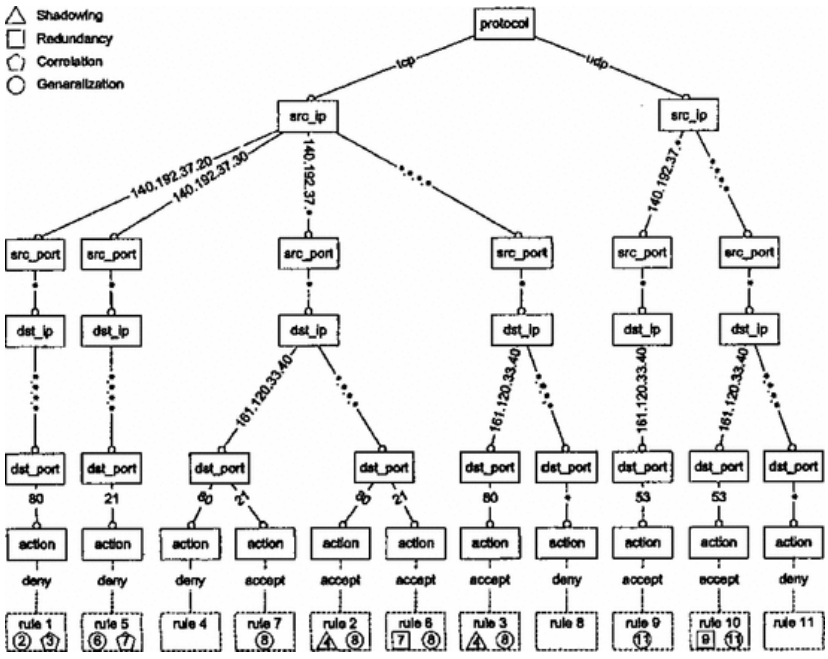

Figura 2.7 Árbol de políticas de firewall. Fuente: (Al-Shaer y Hamed 2003, p. 21)

E.S. Al-Shaer y Hamed (2004), extendieron el trabajo sobre las anomalías en las reglas de firewall con una herramienta para la inserción, modificación y eliminación de reglas libres de anomalías. Los experimentos muestran que el algoritmo basado en el árbol de políticas de firewall implementado en Java, requiere de 10 ms a 240 ms para procesar y analizar un conjunto de 10 a 90 reglas de firewall.

E. Al-Shaer, Hamed, Boutaba y Hasan (2005) continuaron el trabajo sobre el ordenamiento de las reglas de firewall, esta vez extendido en un ambiente de uno o más firewalls. Ellos sugieren que las reglas de firewall tienen que ser escritas y ordenadas de tal forma que no produzcan anomalías que puedan causar vulnerabilidades en la red. Esto requiere un análisis profundo de las reglas para determinar su correcta ubicación y ordenamiento.

Los algoritmos propuestos en la investigación permiten detectar las anomalías en un solo firewall o en firewalls interconectados (ambiente múltiple). Según los experimentos realizados por los autores, los algoritmos pueden ser utilizados en redes reales, puesto que su desempeño para firewalls con menos de 100 reglas es en orden de milisegundos, y el caso promedio tiene una complejidad similar a la lineal. Sin embargo, encontraron que al analizar reglas en una red un poco más grande (de 13 firewalls), el algoritmo tomaba entre 20 y 180 segundos en procesar todas las reglas.

En anuencia a lo anterior, la investigación también incluye la definición formal de las anomalías y la prueba de que estas son los únicos conflictos que se podrían presentar en las reglas de firewall. Basado en la cantidad de referencias hacia sus trabajos, claramente el trabajo de Ehab S. Al-Shaer y Hamed (2003) es uno de los más importantes sobre las relaciones entre las reglas de firewall. Existen otros enfoques sobre el tema, pero ninguno cuenta con el mismo respaldo.

Abedin, Nessa, Khan, y Thuraisingham (2006), realizaron una investigación con un enfoque que va más allá de la detección de las anomalías en las reglas de firewall, y presentan un algoritmo para la detección y resolución de cualquier anomalía presente por medio de operaciones de reordenamiento, además separación de reglas para obtener un nuevo conjunto libre de anomalías. Como complemento del algoritmo anterior, los autores presentaron un algoritmo para la unificación de reglas cuando es posible reducir el número de reglas en el firewall.

Además, los autores sugieren pequeñas modificaciones a las definiciones presentadas en (Ehab Al-Shaer y Hamed, 2002), simplificando la distinción entre reglas disjuntas o parciamente disjuntas, e indican que coincidirán con paquetes de red diferentes.

El algoritmo para la detección de anomalías tiene complejidad lineal con respecto al número de reglas existente, y puede generar un conjunto de reglas que dupliqueel número de campos de las reglas. En el caso de la resolución de anomalías, los autores definen su complejidad como exponencial en el peor caso. El algoritmo para comparar y unificar las reglas utiliza una estructura de árbol, donde cada nodo corresponde a un campo, pero su complejidad no es evaluada.

Se presentan ejemplos de la ejecución de los algoritmos, pero no se menciona ningún experimento que valide la eficiencia ni el comportamiento de los algoritmos con conjuntos de reglas de firewall.

Pozo, Varela-Vaca, Gasca y Ceballos (2009), evaluaron algunos algoritmos existentes para la detección de inconsistencias en reglas de firewall (llamadas *ACLs* en su trabajo) y con base en el análisis y las desventajas detectadas propusieron un algoritmo utilizando el "principio de divide y conquista" para obtener resultados sobre la consistencia de un conjunto de reglas.

Igual que en los trabajos mencionados anteriormente, las reglas contemplan los campos de prioridad: IP fuente y destino, protocolo capa 4 y puertos fuente y destino (TCP y UDP).

El algoritmo propuesto utiliza un árbol de *hashes* por cada campo, en lugar de un solo árbol para clasificar toda la regla como lo hacen E. S. Al-Shaer y Hamed (2003). Las inconsistencias no solo tienen una clasificación, sino que se definen cuando hay una intersección entre reglas que permiten y deniegan el acceso a un mismo destino (Figura 2.8 Definición de inconsistencia. Fuente: (Pozo et al, 2009, p. 4)).

Los experimentos realizados sobre conjuntos de reglas de un firewall real, muestran que el algoritmo requiere un tiempo de ejecución de 10 a 100 veces menor que los propuestos por los trabajos analizados. Se atribuye esta diferencia a la forma en la que se clasifican los campos de la reglas para permitir búsquedas mucho más rápidas que las lineales; pero mencionan que una de las desventajas de este enfoque con respecto a los otros algoritmos consiste en el consumo de memoria de las estructuras utilizadas para analizar las reglas. Si el consumo de memoria fuera obviado, esta

investigación representaría un avance importante en cuanto al desempeño de algoritmos para la identificación de las inconsistencias en reglas de firewall.

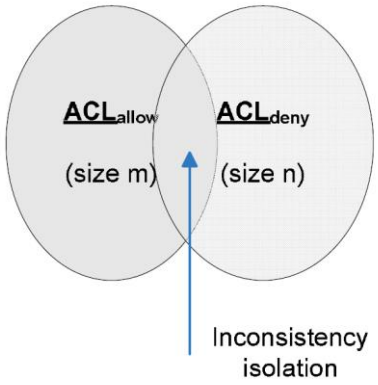

Figura 2.8 Definición de inconsistencia. Fuente: (Pozo et al, 2009, p. 4)

Wang, Ji, Chen, Chen, y Li (2007) desarrollaron un modelo de Markov para estadísticas de coincidencia de paquetes y predicción, basados en una investigación del estado de las reglas. El objetivo del modelo es la optimización del conjunto de reglas de acuerdo con la probabilidad de que los paquetes tengan coincidencia con estas. Se considera los campos de prioridad, IP fuente y destino, protocolo capa 4 y puertos fuente y destino; en el modelo de las reglas

Este método requiere calcular la probabilidad de coincidencia de cada una de las reglas, y con base a esto, presentar un algoritmo de ordenamiento para reducir la cantidad de reglas que se tienen que comparar cuando se evalúa la coincidencia de un paquete de red

Los experimentos se enfocan en el tiempo que tarda un firewall en procesar un paquete con un conjunto reglas optimizado, obteniendo resultados en un conjunto de 3000 reglas de 50 ms a 300 ms con el conjunto optimizado. No obstante no hay datos sobre el tiempo de procesamiento requerido para el cálculo de la probabilidad de coincidencia ni del algoritmo de ordenamiento, por lo que la investigación carece de válidez con respecto a otros trabajos importantes en el área.

Hu, Ahn, y Kulkarni (2012) presentan un marco para la administración de anomalías en las reglas de firewall. Utilizan una técnica de segmentación de reglas para identificarlas y obtener soluciones para estas, complementando de una representación por medio de una cuadrícula para facilitar su visualización. El objetivo de la segmentación de las reglas es obtener resultados más granulares sobre los campos de las reglas que producen las anomalías.

El enfoque de los autores requiere de varias etapas para la detección de las anomalías y su resolución. Para la detección se requiere de un algoritmo de segmentación y uno de correlación de los segmentos; para la resolución se utiliza un algoritmo de ordenamiento que es la combinación de un algoritmo de permutación y un algoritmo voraz para descubrir las soluciones a las anomalías.

Los experimentos realizados demuestran que el algoritmo de segmentación puede procesar un conjunto de 1000 reglas en alrededor de 2.3 segundos, para la detección de las anomalías y el algoritmo de correlación tarda alrededor de 0.5 segundos. Los algoritmos para la resolución de las anomalías tardan entre 100 y 250 segundos en ser ejecutados sobre el conjunto de 1000 reglas. Es evidente -gracias a esta investigación- que la detección de anomalías es un problema mucho más sencillo que su resolución.

Gawanmeh y Tahar (2012) propusieron un modelo formal, sobre las reglas de firewall y el algoritmo para detectar e identificar conflictos en un conjunto de reglas, basado en la restricción del dominio para el cálculo de conflictos de las configuraciones de firewall.

El modelo presentado contempla únicamente las direcciones IP y las acciones "permitir o denegar", pero hacen la aclaración de que puede ser adaptado para soportar los puertos capa 4. El modelo presentado, no describe los tipos de conflictos en detalle y los trata simplemente como inconsistencias entre permitir y denegar un mismo paquete de red.

Los autores indican que el algoritmo presentado tiene complejidad cuadrática n^2, donde n representa la cantidad de reglas por evaluar. Los autores sugieren que el algoritmo se puede utilizar cuando se actualizan las reglas de un firewall, pero no se presenta ningún experimento sobre la eficiencia del algoritmo ni del modelo, se da únicamente un ejemplo de su funcionamiento. De los resultados de la investigación se puede inferir que esta variación del modelo no representa una ventaja frente al trabajo propuesto por E. S. Al-Shaer y Hamed (2003).

Li, Wan y Li (2013) investigaron el problema sobre la generación de las reglas para un firewall, al basarse en el estudio de las datos en bitácoras de seguridad de las redes. Con base en estos datos propusieron un algoritmo llamado "Generación de Reglas de Dominio Específico" (DSRG por sus siglas en inglés), que utiliza información de la configuración de la red para ayudar con la generalización de las reglas de firewall. Además, definen la generalización de las reglas como la "reducción de las reglas" por medio de la combinación de las mismas.

Ellos exponen que el problema de generalización de las reglas de firewall es un problema NP-Completo, pero no presentan experimentos del algoritmo DSRG con un conjunto de reglas de firewall.

Ellos sugieren que el algoritmo DSRG puede ser utilizado para la identificación de anomalías en las reglas de firewall o en el análisis dinámico de un conjunto de reglas existente, pero no dan pruebas de la efectividad del algoritmo sobre las sugerencias dadas.

Yan, Zhaoxia y Qingrong (2013) proponen mejorar un algoritmo para detectar conflictos en las reglas de firewall basado en un árbol de binario, donde se utilizan operaciones de intersección para detectar los conflictos entre las reglas.

El algoritmo propuesto tiene complejidad: $O(n + \log n)$; y se compara únicamente con el algoritmo sobre el cual se realiza la mejora. Los experimentos demuestran que para construir la estructura sobre la cual trabaja el algoritmo con un conjunto de 1000 reglas se tarda aproximadamente 25 ms, pero no indican los tiempos que tarda el algoritmo en detectar los conflictos en las reglas de firewall.

Adicionalmente, en el trabajo no se definen los conflictos que detecta el algoritmo, por lo cual los experimentos realizados son poco concluyentes con respecto a la eficiencia presentada frente a trabajos similares.

Khummanee, Khumseela y Puangpronpitag (2013), toman un enfoque diferente sobre el problema de las anomalías en las reglas de firewall. Ellos presentan una forma para diseñar las reglas de firewall que se componen de una política de administración de las reglas de firewall, una estructura de árbol y un algoritmo para revisar las reglas en busca de conflictos.

Similar que en los otros modelos presentados, las reglas de firewall se componen de: las direcciones IP fuente y destino, protocolo capa 4, los puertos fuente/destino, y la acción permitir o denegar. Los conflictos se definen como las inconsistencias cuando dos reglas definen acciones diferentes sobre un mismo paquete de red.

La estructura utilizada para el algoritmo, es un árbol de profundidad constante, donde cada uno de los niveles corresponde a uno de los campos de las reglas de firewall. Esto causa que al insertar reglas se tenga que regenerar el árbol por la forma en que se acomodan los campos en cada nivel del árbol. Aún sí, Khummanee et al. (2013) indican que la complejidad del algoritmo se mantiene en

$O(\log_2 n)$ para la inserción de una nueva regla y de $O(m \log n)$ donde n es la cantidad de reglas y m es la profundidad del árbol.

Los experimentos están detalladamente explicados, utilizando conjuntos de entre 1 y 16384 reglas. Se demuestra que para crear la estructura se toma entre 3 ms y 27900 ms, mientras que para procesar la estructura en busca de reglas toma entre 13 ms y 28 ms. La investigación indica que un factor a considerar sobre los conflictos de las reglas de firewall está en su creación, y que un enfoque preventivo de conflictos en la creación de las reglas podría ser efectivo. Ellos aprovechan la simplicidad de las reglas de firewall para crear una estructura de datos, la cual permita recorrerla en busca de anomalías de una forma eficiente.

2.3.2 Análisis de las redes

Jarschel, Oechsner, Schlosser, Pries, Goll, y Tran-Gia (2011) investigaron el problema de la escalabilidad y el desempeño de las redes *OpenFlow*. En su trabajo realizaron mediciones de los tiempos de paso de paquetes en *switches OpenFlow*, y derivaron un modelo básico para la velocidad de enrutamiento de un *switch OpenFlow* combinado con un controlador *OpenFlow*, aclarando que el modelo presentado está limitado a un único *switch* por controlador. Para validar el modelo utilizaron una simulación de paso de paquetes, incluyendo la interacción con el controlador.

Según Jarschel et al. (2011), los experimentos mostraron que la interacción de los controladores en la decisión del enrutamiento de los paquetes tiene un impacto considerable, por lo que consideran importante el desempeño del controlador para instalar entradas en las tablas de flujos de los *switches OpenFlow*.

Es Importante recalcar que este trabajo refuerza la importancia de la eficiencia, con la cual un software tendría que evaluar las reglas dentro de un *switch OpenFlow* cuando se agregan nuevas reglas.

Perešíni y Canini (2011), realizaron un análisis de aplicaciones existentes para revisiones de modelos y de su desempeño, al revisar aplicaciones *OpenFlow*. Los autores utilizaron una aplicación de *MAC-learning* para verificar las capacidades de las herramientas analizadas.

En su investigación encontraron que las aplicaciones no especializadas tienen problemas en dos áreas principales, la capacidad de representación de los modelos, incluyendo la facilidad con la que se pueden representar las aplicaciones *OpenFlow*, y el desempeño de los revisores de modelos

cuando la representación del modelo tiene una complejidad aceptable para las aplicaciones *OpenFlow*.

Esta investigación apoya la idea de que las soluciones genéricas sobre la revisión de las aplicaciones *OpenFlow* tienen problemas lidiando con la explosión del espacio de estados del modelo.

Canini, Venzano, Peresini, Kostic y Rexford (2012), continuando con su trabajo anterior, presentaron un conjunto de técnicas para probar aplicaciones *OpenFlow* llamada NICE (del inglés *No bugs In Controller Execution*), al utilizar revisiónes de modelos para explorar el espacio de estado de todo el sistema, incluyendo el controlador, los *switches* y los clientes.

El modelo sugerido en este trabajo utiliza sistemas transiciones de estado para representar el controlador y los *switches*. En lugar de representar las reglas y sus campos, el modelo utiliza las transiciones de estado para representar los eventos que ocurren cuando un paquete pasa por la red, esto con el fin de representar una versión simplificada de los *switches OpenFlow* para reducir el espacio de estado sobre el cual se ejecuta la revisión del modelo.

Los autores realizaron experimentos para probar su prototipo contra aplicaciones existentes sobre revisión de modelos *OpenFlow* y encontraron que NICE se ejecutó hasta 5 veces más rápido hasta en ejemplos pequeños.

Este trabajo presenta una alternativa a las pruebas de aplicaciones *OpenFlow* que se ejecutan tanto en controladores como en *switches*, pero se enfoca en las transiciones que sufren los paquetes y no en la verificación de la configuración de los controladores o switches *OpenFlow*.

Kazemian, Varghese y McKeown (2012) propusieron un marco de trabajo llamado "Análisis del Espacio de Encabezado" (HSA, por sus siglas en inglés), para identificar fallas en las redes independientemente del protocolo que estén ejecutando.

Las fallas que se intentan identificar son fallas más complejas que simples interacciones entre dos reglas de firewall, como por ejemplo, fallas de accesibilidad, ciclos de enrutamiento, entre otros. Para esto, analizan todo el encabezado de los paquetes y nos los campos individuales.

El modelo de las reglas se basa en un espacio L-dimensional, donde L es el número de bits del encabezado de los paquetes de red. Además, se requiere el modelado de los *Routers* por medio de

funciones de transformación sobre el espacio *L*-dimensional, para poder modelar las transformaciones que sufren los paquetes cuando pasan por la red modelada.

Kazemian et al (2012), realizaron experimentos sobre una red real de -la Universidad de Stanford- que contenía aproximadamente 750000 entradas de enrutamiento y 1500 reglas de firewall. Para generar la topología de la red, el algoritmo propuesto tardó 151 s, y para procesar la detección de ciclos se tardó 560s.

Parece que el enfoque presentado es bastante poderoso para modelar y analizar distintas fallas en la red, pero se requiere de un modelado complejo y un tiempo considerable de ejecución. Los autores sugieren que es posible optimizar el modelo y los algoritmos para que sean viables en ambientes de producción y no sólo en pruebas de la red; utilizando aritmética de 64 bits en los cálculos realizados, procesamiento en paralelo y mapas de Karnaugh para reducir el espacio de encabezado, aunque no indican cómo se reduciría el espacio con los mapas.

2.3.3 Reglas *OpenFlow*

Bifulco y Schneider (2013) investigaron sobre la interacción que existe entre las reglas dentro de un *switch OpenFlow*. Su trabajo se basa en las definiciones de las interacciones entre las reglas de firewall dadas por E.S. Al-Shaer y Hamed (2004).

Estos autores presentaron una definición formal de las posibles interacciones en las reglas instaladas en un *switch OpenFlow*. Para esto definen las posibles relaciones entre: los campos de las reglas, los conjuntos de campos de cada regla y entre las acciones de cada . La interacción se deriva de las relaciones previamente definidas entre los campos y las acciones de las reglas.

Además de la definición formal de las interacciones, Bifulco y Schneider (2013) presentaron un algoritmo para la detección de las interacciones, el cual detecta las interacciones de una regla, comparando todos sus campos con los campos de otra regla, por lo que para un conjunto de *n* reglas, el algoritmo tiene complejidad O(*n*) al agregar una nueva regla.

Los experimentos fueron claramente explicados y fueron realizados en conjuntos de 500 a 10000 reglas, con distintas cantidades de interacciones entre los conjuntos. El algoritmo tardó entre 5 ms y 100 ms en procesar los conjuntos de reglas. Estos especialistas (2013) sugieren que las dimensiones en los conjuntos de reglas pueden limitar las aplicaciones del algoritmo en escenarios

reales, pero lo consideran apto para aplicaciones *OpenFlow* de unos cuantos cientos de reglas, o para utilizarlo durante el desarrollo de las aplicaciones.

Este trabajo se enfoca en identificar y detectar la interacción entre reglas *OpenFlow* en general y no exclusivamente en un controlador, por lo cual se considera como la base para herramientas enfocadas en el desarrollo y pruebas de aplicaciones que no requieren modelado ni transformaciones de las reglas *OpenFlow*.

Ehab Al-Shaer y Al-Haj (2010) presentaron una herramienta para identificar problemas de configuración en una tabla de flujo dentro de un *switch OpenFlow*. Ellos describen cómo se pueden dar inconsistencias entre diferentes *switches OpenFlow* en una o diferentes infraestructuras de este tipo. Para lograrlo, propusieron la codificación de las tablas de flujo utilizando diagramas de decisiones binarios y una técnica de revisión de modelos para modelar las conexiones entre los *switches OpenFlow*.

La representación por medio de diagramas de decisión binarios, utiliza álgebra Booleana para la comparación de las reglas, por ello requiere convertir los *bits* de los valores de los campos en variables booleanas, y además requiere la construcción manual de los diagramas de decisión binarios para cada caso específico que se desee probar.

Los experimentos realizados indican que el análisis de las tablas se puede hacer en orden de milisegundos, pero no indican los valores exactos. Aun así los autores sugieren que es posible utilizar esta herramienta en tiempo de ejecución por los administradores de red para resolver conflictos entre *switches OpenFlow*.

Iyer, Mann y Samineni (2013), presentaron un sistema llamado *switchReduce* para disminuir el estado (recursos necesarios para mantener su configuración) del *switch* y su interacción con el controlador *OpenFlow*. Para ello se basan en el planteamiento que el número de reglas en cualquier *switch* debe ser no más de un conjunto único de acciones de procesamiento que se ejecutan sobre los flujos de red.

El mecanismo para reducir la cantidad de reglas se diseñó con la idea de agrupar aquellas que tienen las mismas acciones, ya que según los autores, la mayoría de los *switches OpenFlow* tienen menos de 128 puertos de salida; por lo que muchas de las acciones se podrían clasificar como salidas por los puertos. Utilizando el conocimiento de la red que posee el controlador *OpenFlow*,

switchReduce, puede generar un conjunto de reglas más generales que abarquen los mismos flujos originales.

Iyer et al (2013), realizaron experimentos con topologías de red de un centro de datos simulados, una con 400 servidores y otra con 11520 servidores. En la topología pequeña obtuvieron reducciones de entre 49% y 99% del conjunto original de reglas, y en la topología grande obtuvieron reducciones de entre 40% y 90%. Los experimentos no indican el tiempo necesario para que *switchReduce* se ejecute sobre las topologías.

Este enfoque se basa en la interacción del *switch* con la red, necesita el conocimiento total que posee el controlador sobre esta. No es posible utilizarlo independientemente dentro de un switch o en el desarrollo aislado de aplicaciones para estos, tampoco toma en cuenta cambios en la topología de la red.

Vishnoi, Poddar, Mann y Bhattacharya (2014), presentaron un controlador *OpenFLow* que tiene el propósito de reducir el consumo de espacio en el hardware especializado para las reglas dentro de los *switches OpenFlow*, cuando la cantidad de reglas que se requieren es mayor que la capacidad del hardware.

Los autores mencionan que las técnicas más comunes para evitar este problema se basan en instalar reglas con tiempos de espera cortos, con lo cual se logra el desalojo de las reglas y se liberan recursos de hardware. El trabajo combina una heurística adaptativa de tiempos de espera con el desalojo proactivo de reglas, con el objetivo de utilizar los recursos de hardware efectivamente sin impactar la carga en el controlador.

Cabe mencionar que realizaron varios experimentos con diferentes cantidades de reglas y recursos de hardware, utilizando rastros de paquetes de red en cuatro centros de datos reales, e indican que en todas las combinaciones su implementación fue la primera o segunda mejor comparadas con las implementaciones existentes que analizaron.

Este trabajo no presenta un modelado ni transformacion de las reglas *OpenFlow*, sino que se enfoca en instalarlas inteligentemente cuando los recursos de hardware funcionan cerca del límite de sus capacidades, lo que no impide que se pueda combinar con otras técnicas para reducción de reglas o eliminación de conflictos entre estas.

2.3.4 Lenguajes *OpenFlow* de alto nivel

Foster, Harrison, Freedman, Monsanto, Rexford, Story y Walker (2011) presentaron un lenguaje declarativo de alto nivel para la programación de redes *OpenFlow* llamado *Frenetic*, este lenguaje transforma instrucciones de alto nivel en reglas de programación de bajo nivel.

Los autores mencionan que el aporte de su trabajo está en el sistema administrador de todos los detalles relacionados con la instalación, desinstalación y consulta de las reglas *OpenFlow* en los *switches*.

Para evaluar el lenguaje creado, estos investigadores utilizaron aplicaciones existentes para un controlador *OpenFlow* y las implementaron en Frenetic, comparando la cantidad de líneas de código, el tráfico enviado al controlador y el tráfico total de las aplicaciones. Según los resultados presentados, el uso de *Frenetic* no sólo redujo la cantidad de líneas de código necesarias, sino que también redujo ligeramente el tráfico de la aplicación y hacia el controlador. Debido a esto los autores consideran que *Frenetic* es competitivo con las aplicaciones *OpenFlow* escritas a bajo nivel con reglas *OpenFlow*.

Estos trabajos requieren una conversión en la representación de las reglas *OpenFlow* y no proveen un acceso directo a las reglas que se instalan en los *switches OpenFlow* ni a las interacciones que puedan existir entre estas, pero su implementación no descarta otra de este tipo de técnicas en las reglas generadas a partir del lenguaje de alto nivel.

Monsanto, Foster, Harrison y Walker (2012) continuaron el trabajo desarrollado por Foster et al (2011), al desarrollar *NetCore*, contribuyen con el diseño de nuevos algoritmos para compilar las consultas, administrar las interacciones que surgen entre en *switch* y el controlador como consecuencia de las consultas que son ejecutadas. El objetivo de los nuevos algoritmos era asegurarse que la mayoría de los paquetes de red fueran procesados eficientemente en los *switches*, sin tener que enviarlos al controlador.

Para evaluar *NetCore*, Monsanto et al (2012), utilizaron una serie de pruebas pasando 100K paquetes por un controlador *OpenFlow* con *NetCore* instalado, y otro sin *NetCore*. Los resultados muestran que el controlador con *NetCore* instalado, superó el desempeño en casi todas las pruebas del controlador sin *NetCore*.

Esta investigación mostró que los lenguajes de alto nivel pueden ser considerados como una alternativa a la creación manual de aplicaciones *OpenFlow*, pero no se define el poder expresivo del lenguaje ni las limitaciones que podrían llegar a tener para la creación de las aplicaciones.

Voellmy, Wang, Yang, Ford y Hudak (2013), investigaron sobre la reducción de la complejidad en la programación de las redes definidas por software (SDN por sus siglas en inglés), mediante *OpenFlow*. En este trabajo presentaron un sistema llamado *Maple* para simplificar la programación de las SDN, al permitirle al programador utilizar un lenguaje de programación estándar para diseñar un algoritmo centralizado con el fin de solventar el comportamiento de la red, y proveeno una abstracción que el algoritmo ejecuta en cada paquete que entra a la red.

Los datos de los experimentos conducidos por Voellmy et al (2013), indican que Maple puede generar reglas *OpenFlow* comparables a las aplicaciones similares de bajo nivel, y que el sistema es escalable en controladores de gran capacidad de procesamiento; pero no son concluyentes en la efectividad de Maple con respecto a escenarios reales. Este trabajo presenta un enfoque más general sobre la programación de redes *OpenFlow*, pero carece de ejemplos en aplicaciones *OpenFlow* que puedan ser generadas, y la capacidad de expresión de Maple no es evaluada ni justificada.

Nelson, Guha, Dougherty, Fisler y Krishnamurthi (2013), presentaron *FlowLog*, un lenguaje de alto nivel para la representación de flujos de red. En este trabajo se enfocaron en definir un lenguaje que permitiera fácilmente el análisis del mismo, como la verificación y búsqueda de "pulgas" en el código. A cambio de esta facilidad en el análisis los autores restringieron el lenguaje, por ejemplo; no permite ningún tipo de recursión en su programación.

Los experimentos realizados por Nelson et al (2013), para validar el desempeño de *FlowLog* muestran que el lenguaje se comporta de manera similar a los trabajos comparados, pero los autores indican que la ventaja de *FlowLog* está en ser un lenguaje que permite ser analizado y verificado para producir controladores *OpenFlow* confiables.

Existe la posibilidad de utilizar lenguajes externos en la programación de los flujos, pero esto hace que el lenguaje pierda importancia, pues cuando se detecta una limitación para programar cierta funcionalidad, seráa necesario utilizar otro lenguaje para programarla.

Nelson, Ferguson, Scheer y Krishnamurthi (2014), continuaron su labor desarrollada en *FlowLog* (Nelson et al., 2013), suministrando una sólo abstracción para los planos de control, datos y de estado del controlador para las aplicaciones *OpenFlow*.

El objetivo de la versión actualizada de FlowLog es compilar proactivamente el comportamiento de las tablas de flujo en los *switches OpenFlow*. Para las reglas que mantienen estado del controlador (envían paquetes al controlador), el compilador indica a los *switches* enviar el mínimo de tráfico necesario al controlador.

En esta investigación se implementaron aplicaciones reales utilizando *FlowLog*, como un proxy ARP, un firewall *statefull* (que recuerda el estado de las conexiones), y una aplicación para permitir el acceso a un Apple TV a través de subredes.

Para validar las aplicaciones, Nelson, Ferguson, Scheer y Krishnamurthi (2014), utilizaron una herramienta comercial de verificación, con la que además pudieron encontrar pulgas en el código de las aplicaciones. Asimismo, crearon aplicaciones que habían sido creadas previamente en *Frenetic* (Foster et al., 2011), e indicaron que la cantidad de reglas producidas era similar a las producidas por *Frenetic*.

Este trabajo también compara otros 9 lenguajes de programación de alto nivel para redes Open-Flow. Esto demuestra el auge que está teniendo esta tendencia, pero también se puede inferir que las investigaciones se han enfocado en facilitar la programación de aplicaciones y cada una ha aplicado un enfoque propietario a los problemas de creación y eliminación de reglas *OpenFlow*.

2.4 SÍNTESIS DE LA REVISIÓN

La interacción entre las reglas *OpenFlow* definida por Bifulco y Schneider (2013), es similar a la definición de las interacciones de reglas de firewall propuesta por Ehab Al-Shaer y Hamed (2002), pero el algoritmo propuesto para la detección en las reglas *OpenFlow*, es trivial e implica la comparación con todas ellas, a diferencia del algoritmo para las reglas de firewall de Ehab Al-Shaer y Hamed (2002), donde encontraron que utilizando árboles para guardar los campos de las reglas, hacía más eficiente la identificación de aquellas que podían tener una interacción, además de que mostraron que el trabajo podía ser útil en ambientes académicos e industriales.

En trabajos posteriores, E.S. Al-Shaer y Hamed (2004), encontraron que era posible utilizar el algoritmo para generar herramientas de generación de reglas sin anomalías. E. Al-Shaer et al. (2005), encontraron que el algoritmo se podía extender a la interacción de las reglas en uno o más firewalls.

En su investigación, Li et al. (2013), Khummanee et al. (2013) y Yan et al. (2013), utilizaron árboles como las estructuras de datos para identificar conflictos en las reglas de firewall, y todos encontraron que los algoritmos se comportan mejor que los triviales. Pozo et al. (2009), utilizaron un árbol de hashes para su algoritmo de detección y encontraron que esto reducía considerablemente el tiempo de ejecución frente al algoritmo trivial. Indican que el factor clave está en la forma en la que se clasifican los campos, pero mencionan que una de las desventajas está en el consumo de memoria de las estructuras utilizadas.

La tendencia de los trabajos mencionados está en utilizar estructuras de datos auxiliares, para mejorar el rendimiento de los algoritmos de identificación de conflictos entre las reglas de firewall, por lo que se considera una alternativa viable al algoritmo trivial propuesto por Bifulco y Schneider (2013).

Abedin et al. (2006) y Hu et al. (2012), utilizaron métodos de división de reglas como la base de sus trabajos, ellos encontraron que los algoritmos de detección son eficientes, pero la división de las reglas implica un sobre trabajo mayor al de un algoritmo trivial que no divide las reglas. Las propuestas científicas de Wang et al. (2007) y Li et al. (2013), se basan en el conocimiento de la red para prevenir los conflictos de la reglas, sin resultados concluyentes sobre la efectividad del enfoque.

En el caso de las reglas *OpenFlow*, el trabajo de Ehab Al-Shaer y Al-Haj (2010) es similar al presentado por Bifulco y Schneider (2013), y se enfocó en la representación de las reglas *OpenFlow* por medio de diagramas binarios de decisión y álgebra Booleana, lo que requiere una transformación completa de las reglas *OpenFlow*, aunque encontraron que el procesamiento de los diagramas es eficiente, no dieron datos del costo de construcción de los diagramas. Además, no se dieron definiciones de las posibles interacciones entre las reglas.

Los trabajos de Iyer et al. (2013) y Vishnoi et al. (2014), se enfocaron en reducir la cantidad de reglas que son utilizadas para expresar los flujos de red, pero requieren todo el conocimiento

wsobre esta, no únicamente del *switch*. Se pueden considerar como trabajos ortogonales al desarrollado por Bifulco y Schneider (2013).

Finalmente, los trabajos de Monsanto et al. (2012), Voellmy (2013), Nelson et al. (2013) y Nelson et al (2014); se enfocaron en la creación de lenguajes de alto nivel para la generación de reglas *OpenFlow*. Estos encontraron que los lenguajes de alto nivel pueden ser considerados como una alternativa a la creación manual de reglas *OpenFlow*, pero aún siguen presentando limitaciones. Esto difiere del trabajo de Bifulco y Schneider (2013), en donde la interacción de las reglas se definió independientemente de su representación y la limitación del algoritmo es limitada únicamente por el tiempo de ejecución requerido.

Queda claro que la investigación en este campo es variada, y los autores todavía aún experimentan con desarrollar el tema desde varios enfoques dado que la mayoría de los trabajos encontrados son poco concluyentes o carecen de experimentos relevantes. Esto abre paso a futuras investigaciones donde la metodología de experimentación permita obtener resultados más contundentes sobre los avances realizados.

Los trabajos que han tenido mayor importancia en las reglas de firewall, se enfocan en la definición genérica de las relaciones entre las reglas y en la optimización de algoritmos que se encarguen de identificar estas relaciones. Los resultados más relevantes se obtuvieron de los algoritmos que utilizaban estructuras de datos para mejorar el rendimiento de la búsqueda de conflictos entre reglas, por lo que esta investigación se enfoca de una manera similar en la optimización del algoritmo de identificación de interacciones entre reglas *OpenFlow*.

3 MARCO METODOLÓGICO

3.1 REVISIÓN DE LITERATURA

Para la recolección de información sobre la revisión de literatura, se utilizó un enfoque tradicional, donde se tomó como base el trabajo principal y de ahí se realizaron búsquedas en profundidad en cada una de sus referencias. Adicionalmente, se realizaron indagaciones sobre temas similares en sitios de revistas indexadas.

Luego de haber realizado un análisis inicial de la información recolectada, se ejecutó una segunda búsqueda sobre los autores considerados como más relevantes en el problema de investigación, para complementar el compilado de información inicial sobre el cuál se realizó la revisión de literatura.

A continuación se presentan las fuentes primarias y secundarias de esta investigación:

Fuentes primarias: La fuente primaria de información de esta investigación fue el experimento realizado, descrito con detalle en la sección 3.2.

Fuentes secundarias: Las fuentes secundarias de esta investigación fueron principalmente las bases de datos de revistas indexadas, aunque también se utilizaron sitios web relacionados con la tecnología *OpenFlow* y la búsqueda general de artículos científicos, los cuales se enlistan a continuación:

- IEEE Xplore http://ieeexplore.ieee.org/
- ACM Digital Library http://dl.acm.org/
- Open Networking Foundation https://www.opennetworking.org/
- Google Scholar http://scholar.google.com/

3.2 EXPERIMENTO

Basado en los principios de aleatoriedad, replicación y bloqueo expuestos por Montgomery (2008). Se realizó el diseño de experimento de confirmación, para analizar y validar el comportamiento de los algoritmos con respecto a sus factores, esto con el objetivo de comprobar la veracidad de la hipótesis propuesta en la sección 3.2.4.

Para obtener conclusiones objetivas sobre los resultados, se utilizó la prueba Kruskal-Wallis (Montgomery, 2008), por medio del paquete de software *R* (*The R Foundation*, 2014), estos son explicados con mayor detalle en las secciones 3.2.3.1 y 3.2.3.2 respectivamente. La prueba es equivalente a un análisis de varianza no paramétrico, donde no se espera que la distribución de los residuos sea normal.

3.2.1 Diseño del experimento

Para analizar el comportamiento de los algoritmos, se utilizó un experimento de un solo factor (Montgomery, 2008), al ejecutar los distintos experimentos en orden aleatorio para cumplir con el principio de aleatoriedad, se repiten 20 veces el experimento con cada una de las poblaciones generadas para cumplir con el principio de replicación, y se utilizan distintos niveles únicamente en el factor relevante a la investigación para cumplir con el principio de bloqueo.

Para generar las poblaciones, se utilizaron diferentes configuraciones basadas en el experimento realizado por Bifulco y Schneider (2013). A continuación, se presenta el factor del experimento, variables de respuesta, la generación de las poblaciones y el detalle del experimento realizado.

3.2.1.1 Factor

El factor tomado en cuenta en el experimento fue el algoritmo.

3.2.1.2 Niveles

Los niveles para el factor del experimento (considerados como los tratamientos del experimento) fueron: original y alternativo.

3.2.1.3 Variables de respuesta

La variable de respuesta o variable dependiente para el experimento, es el tiempo de ejecución del algoritmo, se tomó únicamente en cuenta el tiempo durante el cual la simulación está ejecutándose, desde que el momento que algoritmo recibe la nueva FTE hasta que se retorna el resultado.

3.2.1.4 Experimento de un factor

El objetivo del experimento fue determinar el impacto de los algoritmos con una serie de poblaciones distintas. Con base en el diseño del experimento de un factor y las poblaciones

generadas, se realizaron 20 ejecuciones de cada algoritmo con las diferentes poblaciones, para un total de 2400 ejecuciones.

3.2.1.5 Generación de las poblaciones

Para generar los conjuntos de FTEs analizados por los algoritmos, se utilizaron distintos atributos de configuración. El objetivo de los valores elegidos fue realizar un análisis en el comportamiento de ambos algoritmos, tomando en cuenta el análisis del algoritmo original realizado por Bifulco y Schneinder (2013), y con esto obtener una imagen amplia del comportamiento de los algoritmos a través de una variedad de conjuntos de FTEs. A continuación, se presentan los atributos y los valores utilizados:

1. **Cantidad de FTEs por conjunto**: este es el factor más importante, ya que es el principal indicador sobre el comportamiento del algoritmo.

   Valores utilizados: 10,000, 40,000, 70,000 y 100,000 FTEs. Los tamaños de conjuntos se basó los hallazgos de Dusi et al. (2014) sobre la cantidad de reglas instaladas en *switches* de redes reales, y por eso se utilizan los niveles de 10,000 a 100,000 FTEs por conjunto.

2. **Cantidad de campos no-comodín**: indica la cantidad de campos que tienen un valor definido en las FTEs. Entre más campos tengan un valor definido, más comparaciones necesita realizar el algoritmo alternativo.

   Valores utilizados: 2, 4, 6, 8, 10. El propósito de estos niveles fue probar el comportamiento de los algoritmos en conjuntos con niveles uniformes de superposición entre las FTEs. Los valores proveen una superposición de baja a alta respectivamente.

3. **Probabilidad de tener campos con los mismos valores**: se utiliza para dar una indicación sobre conjuntos con muchas o pocas interacciones.

   Valores utilizados: 0, 0.25 y 0.5. El propósito de estos niveles fue generar conjuntos de FTEs uniformemente con muchas o pocas interacciones entre sí, por ejemplo si se tiene un conjunto de 10.000 FTEs con el nivel 0.5 de este factor, el conjunto tendrá aproximadamente 5.000 FTEs con interacciones (campos iguales).

En la Tabla 3.1 se presentan los valores utilizados en la configuración de las poblaciones.

Atributo	Valor 1	Valor 2	Valor 3	Valor 4	Valor 5
Cantidad de FTEs	10,000	40,000	70,000	100,000	
Campos no-comodín	2	4	6	8	10
Probabilidad de campos repetidos	0	0.25	0.5		

Tabla 3.1 Atributos de las poblaciones generadas.

3.2.2 Procedimientos

3.2.2.1 Implementación de los algoritmos

La implementación del algoritmo original se realizó conforme el pseudocódigo publicado en el artículo de Bifulco y Schneider (2013).

Las variantes del algoritmo se implementaron utilizando estructuras de datos adicionales y optimizaciones, basadas en el procesamiento perezoso para el reducir el tiempo de búsqueda de las reglas *OpenFlow* relevantes.

3.2.2.2 Medición de los tiempos de ejecución

Para la medición de los tiempos de ejecución, se realizó una simulación de los algoritmos dentro de una PC convencional. Se tomó en cuenta únicamente el tiempo durante el cual la simulación se está ejecutando desde que el algoritmo recibe la nueva FTE hasta que éste finalizó y proveyó el resultado.

La finalidad de la medición en los tiempos de ejecución de los algoritmos, fue identificar el impacto que tuvo la diferencia de las implementaciones en conjuntos de FTEs que se acercaran a casos de uso reales.

3.2.2.3 Simulación de los algoritmos

La simulación de los algoritmos se realizó en el lenguaje de programación Java utilizando la versión JRE 1.8, ya que existen proyectos de *switches* y controladores *OpenFlow* basados en este lenguaje como *Project Floodlight* (2014). Por esto se consideró un lenguaje adecuado desde el punto de vista del desempeño como de la viabilidad de la implementación. La programación fue llevada a cabo utilizando el entorno de desarrollo conocida como *NetBeans 8.1*.

La PC utilizada para la simulación de los algoritmos tiene las siguientes características:

- Procesador Intel Core i7 640m a 2.8 GHz
- 8 GB de RAM
- Disco duro de 320GB
- Sistema operativo Windows 7

3.2.3 Análisis de resultados

3.2.3.1 Kruskal-Wallis

La prueba de Kruskal-Wallis se utiliza para identificar si las observaciones generadas por los tratamientos son idénticas o presentan una diferencia, y por ende determinar si los tratamientos son igualmente efectivos o no. Esta se puede ver como una prueba sobre la igualdad de las medias de los tratamientos (Montgomery, 2008).

Entonces, se tienen k tratamientos $T_1, T_2, \ldots, T_k$, con medias poblacionales $\mu_1, \mu_2, \ldots, \mu_k$, el objetivo es determinar si existe una igualdad de sus medias poblacionales, si tal igualdad existe, entonces los tratamientos son igualmente efectivos. Con base en lo anterior, las hipótesis se pueden plantear de la siguiente forma:

H_0: Las medias poblacionales no varían según los k tratamientos ($\mu_1 = \mu_2 = \ldots = \mu_k$).

H_A: Algunos de los tratamientos generan observaciones que son más grandes que otras ($\mu_1 \neq \mu_2 \neq \ldots \neq \mu_k$).

La prueba de Kruskal-Wallis permite saber si la diferencia en la media de las observaciones es significativa, debido a la influencia de alguno de los tratamientos, se permite analizar varias poblaciones.

3.2.3.2 Herramientas de análisis

Para realizar el análisis de varianza, se utilizó la herramienta provista por el proyecto R *(The R Foundation*, 2014) en su versión 3.1.2 y el entorno de desarrollo *RStudio* (RStudio, 2014) en su versión 0.98. Estas herramientas, permiten realizar sencillamente la prueba de Kruskal-Wallis sobre los datos obtenidos con los experimentos.

3.2.4 Hipótesis

Siendo *X* los tiempos de ejecución observados, *T* los algoritmos utilizados (tratamientos), y $\mu_{original}$, $\mu_{alternativo}$ las medias poblacionales; basados en la prueba de Kruskal-Wallis descrita en la sección 3.2.3.1 se definieron las siguientes hipótesis de la investigación:

3.2.4.1 Hipótesis nula (H_0)

X no varía según los tratamientos *T*, $\mu_{original} = \mu_{alternativo}$.

3.2.4.2 Hipótesis alternativa (H_A)

X varía según los tratamientos *T*, las medias de las observaciones son menores en al menos 10% cuando el atributo de *T* es *alternativo*, $\mu_{original} * 0{,}9 \geq \mu_{alternativo}$.

3.2.4.3 Nivel de aceptación

El nivel mínimo de significancia esperado del análisis Kruskal-Wallis para dar por válida la hipótesis alternativa es del 97.5%.

3.3 MEJORA DEL ALGORITMO

Como se indicó en la sección 2.2.3, el algoritmo tiene complejidad lineal para la cantidad de comparaciones de campos que hace por todas las FTEs, y luego de un análisis del algoritmo y los resultados producidos se identificaron dos puntos de mejora:

1. El algoritmo compara todos los campos cuando compara 2 FTEs.
2. El algoritmo compara todas las FTEs existentes en un conjunto contra una FTE nueva.

Con base en lo anterior, se propuso una mejora para ambos puntos, con el fin de mejorar el rendimiento aun cuando en una de los puntos se produzca el peor de los casos. La efectividad del agrupamiento con respecto al tiempo de ejecución, depende de la variedad que exista en los valores que tengan los campos de las FTEs, y en un escenario real donde los *switches* comunican miles de clientes diferentes. Es común que las FTEs tengan valores diferentes por la misma naturaleza de las redes de computadoras.

Los detalles de implementación se muestran en el Anexo A: Código fuente de los algoritmos, donde se presenta el código principal de ambos algoritmos.

3.3.1 Reducción de los campos comparados entre 2 FTEs

Bifulco y Schneider (2013), indicaron en su trabajo que se deben comparar los campos de las FTEs uno por uno cuando se comparan 2 FTEs en busca de interacciones. Después de analizar el pseudocódigo se pudo identificar que cuando hay un campo con una relación disjunta, la relación resultante de los campos es también disjunta, y por la definición de las interacciones si la relación de los campos es disjunta la interacción es inexistente.

La mejora consistió en realizar una comparación perezosa (análogo a la inicialización perezosa) de las FTEs, y en el momento en que se detecta una relación disjunta entre dos campos, el algoritmo continúa con la siguiente FTE, sin comparar el resto de los campos ni comparar el conjunto de acciones. Con esto se logró reducir las comparaciones que se tienen que realizar cuando dos FTEs tienen una interacción inexistente, siendo el peor caso que el último campo comparado fuese el campo con una relación disjunta.

En la Figura 3.1 se muestra la comparación entre dos FTEs, en esta comparación el algoritmo original compararía todos los campos de las FTEs mientras que el algoritmo alternativo compararía hasta el campo donde se encontró una relación disjunta.

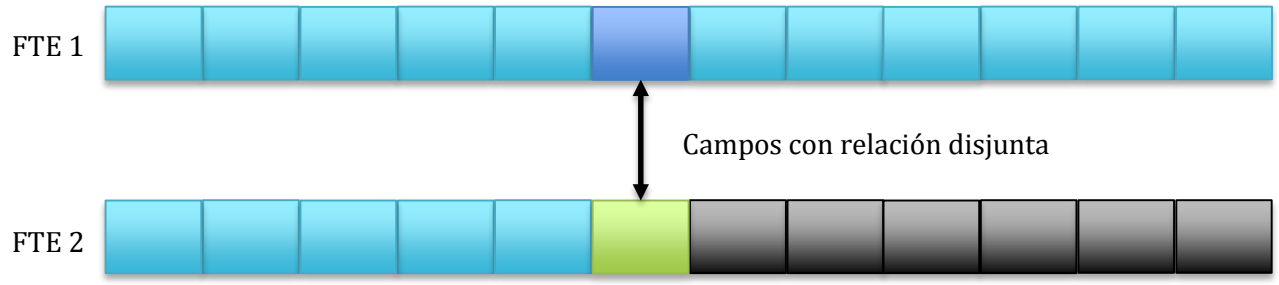

Figura 3.1 Campos comparados entre 2 FTEs.

3.3.2 Reducción del número de FTEs comparadas

De igual forma, la mejora para los campos comparados entre 2 FTEs se basó en la característica según la cual si una relación entre dos campos es disjunta, la interacción es inexistente.

La mejora consistió en agrupar los campos iguales y buscar si el campo de la nueva FTE coincidía con alguno de estos grupos, una vez encontrado el grupo de coincidencia (si había uno) el

algoritmo buscaba interacciones únicamente con las FTEs pertenecientes a ese grupo, pues el resto de FTEs tendrían una relación inexistente con la nueva FTE.

Los campos de las FTEs se pueden clasificar en dos tipos:

1. **Campos de enteros**: campos basados en números enteros donde el valor comodín es completo, y no se permite un comodín parcial. Estos campos son: *in_port, eth_type, ip_proto, tcp_src, tcp_dst, udp_src y udp_dst.*
2. **Campos complejos**: campos que no corresponden estrictamente con números enteros y que permiten comodines parciales o arbitrarios. Estos campos son: *eth_src, eth_dst, ipv4_src, ipv4_dst, ipv6_src e ipv6_dst.*

El agrupamiento de los campos se realizó utilizando como idea inicial para la mejora, el trabajo de Pozo et al. (2009), donde utilizaron un hash de hashes para agrupar los campos de valores enteros en reglas de firewall, con el fin de generar una máscara de bits que representa las reglas de firewall que tienen el valor llave del hash.

El enfoque de Pozo et al. (2009), es para un conjunto estático de reglas, pero la idea se utiliza como base para agrupar los campos enteros de las FTEs que pueden ser indizados con estructuras de datos comunes y tienen una complejidad de acceso menor que la lineal. Para agrupar los campos se utilizaron tablas de hash porque se ajustan a la mejora buscada tanto en el tiempo de acceso como en el tipo de los campos, y los valores se pueden usar como las llaves de la tabla de hash donde se agrupan las FTEs. Inicialmente se utilizó una lista de punteros a FTEs como el valor, pero se reemplazó un por un hash de punteros a FTEs por el tiempo de acceso y de borrado de los elementos.

En la Figura 3.2 se muestra la organización del hash de hashes por campo y punteros hacia FTEs, en este ejemplo si se estuviera buscando las interacciones para una nueva FTE con un valor *in_port* 25, se buscaría el valor en el *hash* de *in_port* y el resto de las comparaciones se realizarían únicamente con las FTEs F1, F2 y F6.

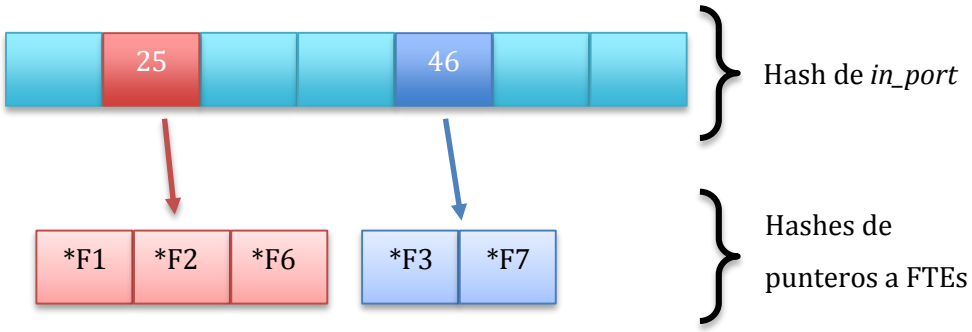

Figura 3.2 Agrupamiento del campo in_port en hash de hashes.

3.3.3 Consumo de memoria

El hash de hashes utilizado en la reducción de FTEs comparadas, tiene un consumo de memoria estimado relativamente bajo, ya que los valores guardados son referencias a otros hashes (punteros) o identificadores de FTEs, que tienen un tamaño de unos 16 *bytes*. Unos elementos son constantes pero la cantidad de elementos totales en cada hash depende de la cantidad de FTEs.

En una máquina virtual de Java de 32 bits se tiene un consumo estimado para un hash (Oracle, 2014), de 36 *bytes* + 62 *bytes* por FTE, lo que corresponde a aproximadamente 6MB para un *hash* de 100,000 entradas. El cálculo es el siguiente:

- 24 bytes del objeto *hash* (8 bytes de sobrecarga del objeto, 3 *int* y un *float*)
- 12 bytes de la constante de un arreglo de espacios.
- 4 bytes por espacio del arreglo.
- 24 bytes de sobrecarga por entrada
- 32 bytes del valor y la llave, cada uno un identificador de 16 bytes.

El consumo puede variar dependiendo de la composición de los conjuntos de FTEs y la distribución en los hash de *hashes*, pero se aproxima a una idea de la cantidad de memoria necesaria para alojar las estructuras de datos utilizadas por el algoritmo alternativo.

Como es la primera versión alternativa del algoritmo, no existe otro algoritmo de comparación, pero dado que la cantidad de memoria utilizada en los *switches OpenFlow* actualmente ronda los varios gigabytes (ya que están destinados principalmente para el sector empresarial), y la mayoría de los proyectos de software libre más populares se ejecutan sobre distribuciones de Linux

(Casado, 2014), es factible considerar que la cantidad de memoria requerida por el algoritmo cabe dentro de la arquitectura actual de los *switches*.

4 RESULTADOS

A continuación se presenta un análisis detallado de los resultados obtenidos a partir de los experimentos, además de una explicación del método elegido para realizar el análisis estadístico de los mismos. Por la naturaleza del experimento, la comparación se hace entre el algoritmo original y el algoritmo alternativo, pero en el capítulo 5 se mencionan las conclusiones sobre este enfoque comparado con otros de los enfoques encontrados.

4.1 RESULTADOS DEL EXPERIMENTO

Inicialmente para el análisis de los resultados se iba a utilizar el análisis de varianza (ANOVA), pero al recolectar los datos se identificó que estos no seguían una distribución normal. En la explicación del ANOVA, Montgomery (2008), sugiere la creación de una gráfica de probabilidad normal para evaluar si los datos cumplen con el requisito de normalidad en los residuos, como se puede observar, en la figura de la izquierda se muestra la gráfica obtenida, mientras en la figura de la derecha se muestra la gráfica esperada obtenida del libro de Montgomery(2008), por lo que se decidió utilizar el análisis Kruskal-Wallis, que es su equivalente no paramétrico, como se describe en la sección 3.2.

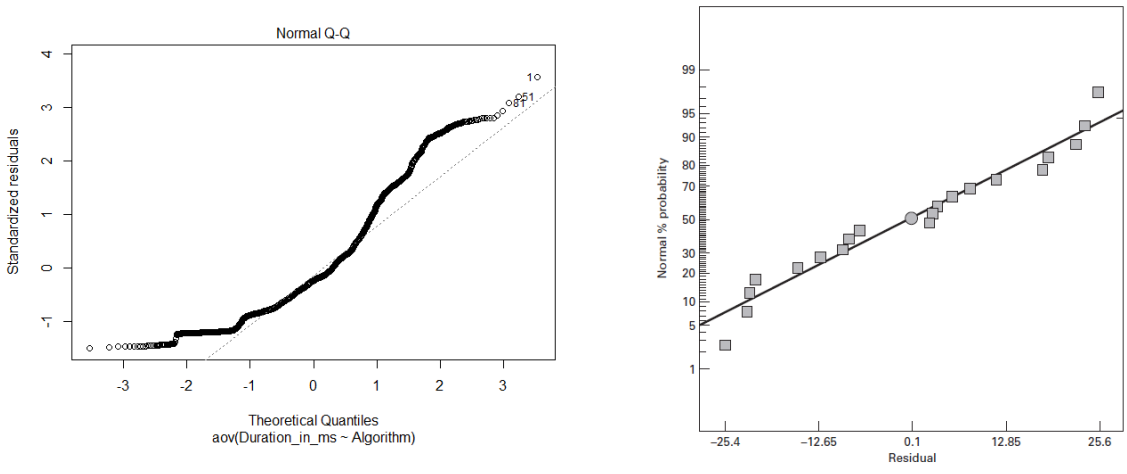

Figura 4.1 Gráfica de probabilidad normal obtenida y esperada. Fuente: ANOVA en RStudio y Montgomery (2008)

Los resultados obtenidos demuestran que la reducción en la cantidad de campos y de FTEs comparados es una forma efectiva de mejorar el rendimiento del algoritmo, la evidencia encontrada indica que la mejora es apta aun cuando la cantidad de FTEs continúa creciendo. Esto no sucede en todos los casos, en la sección 4.4.2 se explica cuando el algoritmo original presentó un mejor rendimiento que el algoritmo alternativo, dada la poca cantidad de comparaciones que hay que hacer con ciertas poblaciones, aunque esto ocurrió sólo en una pequeña parte de los experimentos.

En la Figura 4.2 se muestra la cantidad de interacciones detectadas por los algoritmos, que como es de esperarse, son la misma cantidad.

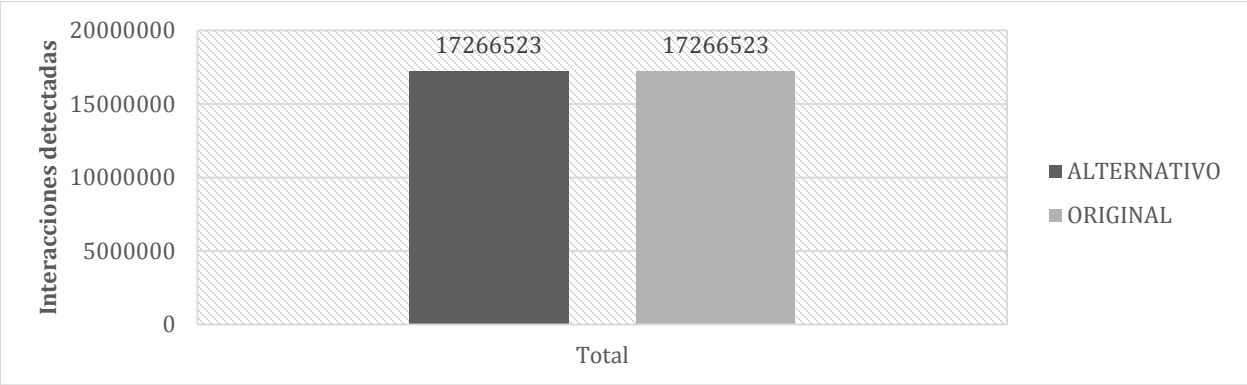

Figura 4.2 Número de interacciones detectadas por los algoritmos.

4.2 VERIFICACIÓN DE HIPÓTESIS

4.2.1 Hipótesis nula (H_0)

Según la definición de la hipótesis nula en la sección 3.2.4 y con base en los resultados obtenidos de los experimentos mostrados en la Figura 4.3, se tiene la siguiente demostración:

$\mu_{original} = \mu_{alternativo}$

$= 53.42 \neq 31.48$

Con base en lo anterior se puede dar por inválida la hipótesis nula.

4.2.2 Hipótesis alternativa (H_A)

Según la definición de la hipótesis alternativa en la sección 3.2.4 y con base en los resultados obtenidos de los experimentos mostrados en la Figura 4.3, se obtiene la siguiente demostración:

$\mu_{original} * 0,9 \geq \mu_{alternativo}$

$= 53.42 * 0.9 \geq 31.48$

$= 48.07 \geq 31.48$

Con base en lo anterior, se puede dar por cumplida la hipótesis alternativa. En la siguiente sección se describe el nivel de significancia obtenido en los resultados:

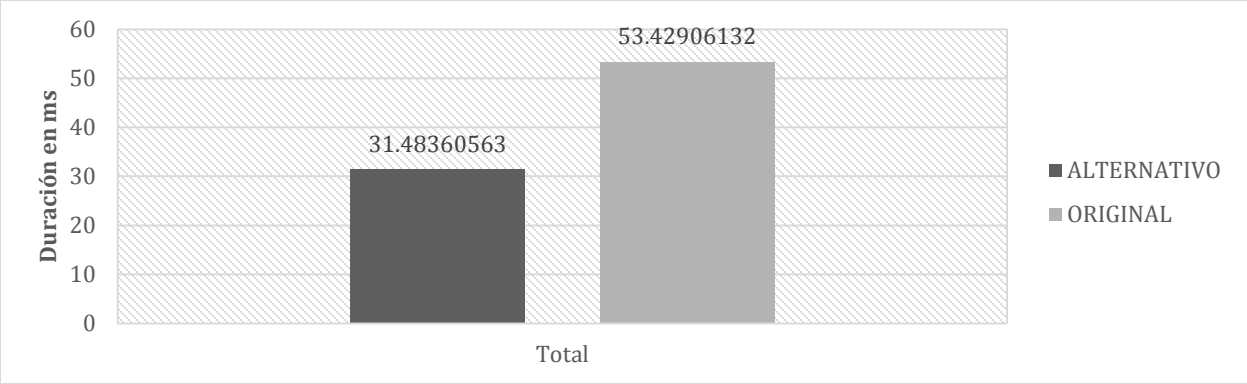

Figura 4.3 Media de los tiempos de ejecución por algoritmo.

4.3 NIVEL DE ACEPTACIÓN

El nivel mínimo de significancia definido en la sección 3.2.4 es de 97.5%. Para esto se realizó el análisis Kruskal-Wallis, con el objetivo de identificar si el algoritmo utilizado tenía influencia sobre los resultados. Los resultados indicaron un *p-value* de 2.2e-16, lo que corresponde a una significancia mayor al 99%, por lo que la hipótesis alternativa se da por válida.

A continuación, se presentan los resultados obtenidos del análisis Kruskal-Wallis realizado por medio del entorno de desarrollo RStudio (RStudio, 2014) y el módulo de software R (The R Foundation, 2014):

> kruskal.test(Duration_in_ms ~ Algorithm, data = ida_exp_csv)

Kruskal-Wallis rank sum test

data: Duration_in_ms by Algorithm
Kruskal-Wallis chi-squared = 241.0213, df = 1, p-value < 2.2e-16

4.4 ANÁLISIS POR POBLACIÓN

Además del análisis general sobre los resultados, se realizó un análisis del comportamiento de los algoritmos separado por las distintas configuraciones de las poblaciones, con el fin de identificar la forma en que el rendimiento de los algoritmos es afectado por los diferentes atributos de las configuraciones.

4.4.1 Duración promedio × (probabilidad de campos repetidos y cantidad de FTEs)

La probabilidad de los campos repetidos, indica la potencial cantidad de interacciones que podrían existir entre la nueva FTE y el conjunto existente de FTEs. Al observar los resultados en la Figura 4.4, clasificados por esta probabilidad, se identificó que la cantidad de interacciones existentes no presenta un impacto significativo en el algoritmo original, pues en todos los casos este compara todos los campos de todas las FTEs, pero como era de esperar sí tiene un impacto en el algoritmo alternativo, ya que la cantidad de reducciones que se pueden realizar es menor conforme a la cantidad de FTEs va aumentando. Aun así, cuando la probabilidad de que los campos tengan los mismos valores ronda un 50%, el algoritmo alternativo siguió presentando un mejor rendimiento que el algoritmo original.

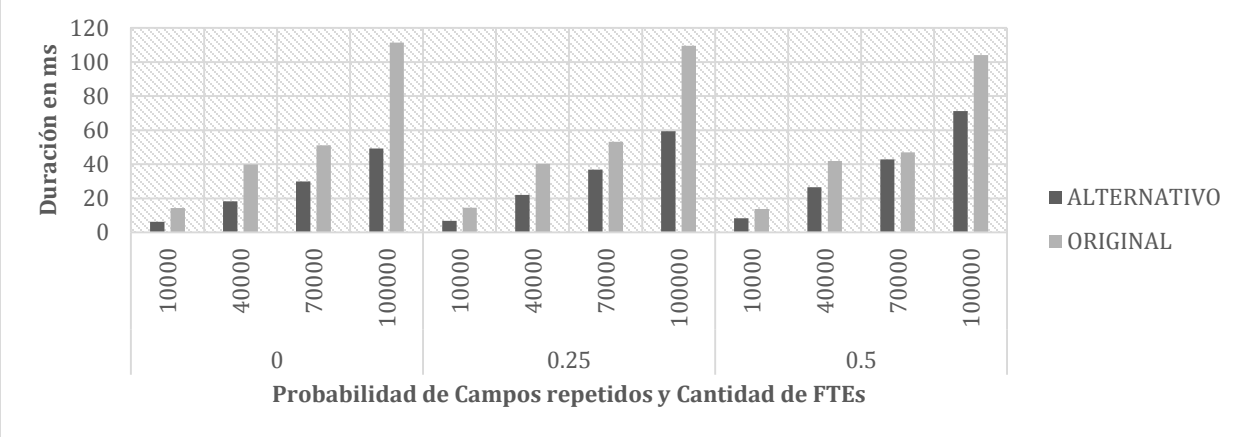

Figura 4.4 Duración por probabilidad de campos repetidos.

4.4.2 Duración promedio × (cantidad de campos no-comodín y cantidad de FTEs)

Caso extremo: campos no-comodín 2: todas las FTEs tienen dos campos con valor definido.

La cantidad de campos no comodín, indica los campos que tienen un valor definido. Para este atributo existe el caso extremo cuando la cantidad de campos no comodín es 2, porque ninguna FTE podría tener más de dos valores definidos; lo cual limitaría en gran parte la funcionalidad de las reglas *OpenFlow*.

Como se muestra en la Figura 4.5, en el caso extremo el algoritmo alternativo tiene ligeramente menor rendimiento que el algoritmo alternativo, ya que al poseer únicamente 2 valores definidos las estructuras de agrupamiento de las FTEs contienen la mayoría de las FTEs en el grupo conformado por los valores comodín, por lo que la reducción de las comparaciones es muy poca y la sobrecarga generada por utilizar estas estructuras es mayor al beneficio reducido que proveen en el caso extremo.

En el resto de las probabilidades, el algoritmo alternativo presentó una mejora, la cual se fue haciendo más notoria conforme las FTEs tenían más campos definidos mientras que el algoritmo original fue aumentando levemente el tiempo de ejecución conforme la cantidad de campos definidos iba aumentando.

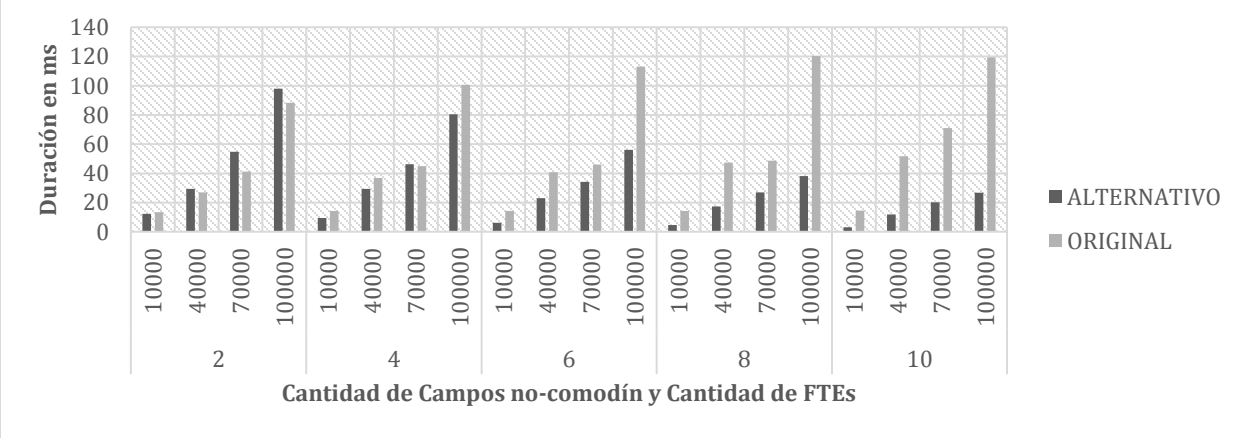

Figura 4.5 Duración por cantidad de campos no comodín.

4.4.3 Duración promedio × cantidad de FTEs

En la Figura 4.6 se presenta un resumen de la duración promedio de los algoritmos por la cantidad de FTEs analizadas, tomando en cuenta la totalidad de los experimentos. Como se muestra en la figura, el algoritmo alternativo presentó en su promedio un mejor rendimiento en todos los tamaños de conjuntos de FTEs.

Con base en estos resultados, se espera que una situación real (con respecto a la cantidad de FTEs y la variedad de valores en la población), el algoritmo alternativo tenga un mejor rendimiento que el algoritmo original.

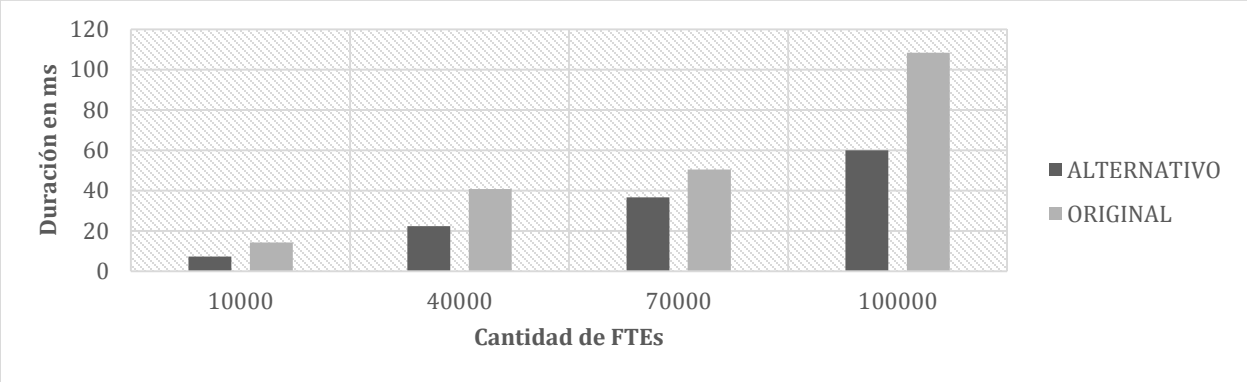

Figura 4.6 Duración por cantidad de FTEs.

4.5 SÍNTESIS DE LOS RESULTADOS

El algoritmo alternativo logró su objetivo de mejorar el rendimiento de manera global sobre el algoritmo original, esto gracias a la reducción en la cantidad de operaciones necesarias para determinar todas las interacciones existentes.

Como era de esperarse, el algoritmo alternativo no es la mejor opción en el 100% de los casos analizados, pero en los casos en que no lo son, -considerados casos extremos- dado que la cantidad de valores definidos por FTE limita enormemente la funcionalidad de estas y no se consideran un caso práctico en una red real. En el resto de los casos el algoritmo alternativo demostró un rendimiento superior a cambio de unos pocos megabytes de memoria adicional.

Como resultado interesante de los datos, el porcentaje de mejora obtenido en los experimentos realizados con conjuntos de 10.000 FTEs, es muy similar al porcentaje obtenido en los experimentos realizados con conjuntos de 100.000 FTEs, lo cual demuestra que la mejora es factible dentro de los grandes conjuntos de FTEs sin perder su eficacia.

5 CONCLUSIONES Y TRABAJO FUTURO

En este capítulo se presentan las conclusiones sobre las mejoras realizadas al algoritmo de detección de interacciones entre reglas *OpenFlow*, esto se considera como uno de los primeros pasos hacia la mejora de este proceso.

Además, se sugieren algunas direcciones que podría tomar la investigación, efectuadas con el objetivo de proporcionar ideas sobre cómo se podría continuar la investigación relacionada con la mejora del rendimiento del algoritmo.

5.1 CONCLUSIONES

En este trabajo se presentó una mejora en el rendimiento del algoritmo de detección de interacciones entre reglas *OpenFlow* utilizando estructuras de datos bien conocidas para la reducción de la cantidad de operaciones necesarias y un procesamiento perezoso de las comparaciones entre las reglas.

En el capítulo 1 se describió el inconveniente de la programación de *switches OpenFlow* por medio de las de reglas, y de cómo esto se puede convertir en un problema en *switches* que contienen más de unos pocos cientos de reglas. El algoritmo original se enfoca en este sentido, pero no es considerado por los investigadores (Bifulco y Schneider, 2013) como apto para utilizarse dentro de un *switch*.

Los experimentos realizados mostraron que la mejora obtenida en el procesamiento depende de la composición del conjunto de reglas, pero de forma general se obtuvo una mejora del 41% el cual puede variar. Se pudo determinar en los cambios realizados al algoritmo, presentan un enfoque alternativo que permite pensar en la utilización del algoritmo dentro de un *switch* y no sólo como parte de los desarrolladores cuando se están creando las aplicaciones.

Las estructuras de datos utilizadas por el algoritmo alternativo, se basaron en el trabajo de Pozo et al. (2009), donde se utilizan estructuras de datos comunes y específicas para identificar las reglas de firewall que tienen una potencial interacción.

Este principio fue utilizado en el algoritmo alternativo, con la diferencia de que se utilizaron únicamente estructuras de datos comunes, dada la diferencia en la cantidad y tipo de los campos

entre las reglas de firewall y las reglas *OpenFlow*. Esto permitió reducir la cantidad de comparaciones necesarias en la mayoría de los casos, como se pudo ver en el capítulo 4. Esta mejora podría ser incorporada a un trabajo similar al de Pozo et al. (2009) sobre reglas de *firewall* ya que permite la utilización de conjuntos de reglas no estáticos, lo cual es una desventaja en el trabajo original.

Además de las estructuras de datos, el procesamiento perezoso de la comparación entre las reglas, formó la otra parte de la mejora propuesta en el algoritmo alternativo. Esta mejora permitió descartar las FTEs no relevantes durante una comparación, evitando la comparación de todos los campos y todas las acciones como en el algoritmo original.

El procesamiento perezoso representó una ligera mejora con respecto al algoritmo original, siendo indiferente de las configuraciones de las poblaciones, gracias a que no requiere procesamiento adicional. La reducción de comparaciones por medio de estructuras de datos si presentó algunos casos extremos, donde el rendimiento se degradó con respecto al algoritmo original; pero en la mayoría de los casos representó la mayor mejora de rendimiento, ya que permitió descartar FTEs sin ni si quiera tener que comenzar a compararlas.

Contrastado con otros enfoques, este trabajo permite una mayor independencia del análisis de las reglas *OpenFlow*, mientras que en otros enfoques se requiere del conocimiento de toda la red (Iyer et al., 2013), o de un servidor externo donde se ejecute la aplicación encargada de analizar las reglas (Ehab Al-Shaer y Al-Haj, 2010). En el enfoque utilizado, se habilita la potencial utilización del algoritmo dentro de un *switch OpenFlow*, un ejemplo de esto sería añadirlo como un módulo más a proyectos de software libre sobre *switches OpenFlow* existentes.

Finalmente, se pudo concluir que aunque el algoritmo alternativo pueda requerir más memoria que el algoritmo original, logra una mejora importante en el rendimiento del algoritmo, y mantiene su efectividad inclusive cuando los conjuntos de reglas *OpenFlow* dentro de los *switches* aumentaron considerablemente de tamaño.

5.2 TRABAJO FUTURO

Esta investigación presentó un algoritmo alternativo, el cual demostró mejorar el rendimiento, no obstante todavía existen algunos problemas sin resolver que podrían conllevar a mejorar aún más el rendimiento del algoritmo.

Uno de estos problemas es la indización de todos los campos de las FTEs, ya que en algoritmo alternativo se indizaron los campos basados en valores enteros. Una de las sugerencias para resolver este problema, consiste en crear estructuras de datos específicas para cada uno de los campos faltantes, que son direcciones *MAC, IPv4 e IPv6*. Estas estructuras deberían tomar en cuenta que los valores pueden tener máscaras arbitrarias, y posiblemente utilizar una estructura de árbol para detectar las relaciones de súper y subconjunto.

Otro de los problemas a resolver, es aquel que consiste en identificar el impacto del análisis de las acciones, para determinar si es necesario realizar una optimización de su comparación, y en caso de serlo, realizar una investigación para lograr este objetivo.

Finalmente, se deja la puerta abierta a futuros investigadores en incluir el algoritmo alternativo como contribución de algún proyecto de software libre sobre *switches OpenFlow*.

REFERENCIAS

Abedin, M., Nessa, S., Khan, L., y Thuraisingham, B. (2006). Detection and resolution of anomalies in firewall policy rules. In *Data and Applications Security XX* (pp. 15–29). Springer. Obtenido de http://link.springer.com/chapter/10.1007/11805588_2

Al-Shaer, E., y Al-Haj, S. (2010). FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration* (pp. 37–44). ACM. Obtenido de http://dl.acm.org/citation.cfm?id=1866905

Al-Shaer, E., y Hamed, H. (2002). Design and implementation of firewall policy advisor tools. *DePaul University, CTI, Tech. Rep*. Obtenido de http://facweb.cti.depaul.edu/research/techreports/TR04-011.pdf

Al-Shaer, E., Hamed, H., Boutaba, R., y Hasan, M. (2005). Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications*, *23*(10), 2069–2084. doi:10.1109/JSAC.2005.854119

Al-Shaer, E. S., y Hamed, H. H. (2003). Firewall policy advisor for anomaly discovery and rule editing. In *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on* (pp. 17–30). IEEE. Obtenido de http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1194157

Al-Shaer, E. S., y Hamed, H. H. (2004). Modeling and Management of Firewall Policies. *Network and Service Management, IEEE Transactions on*, *1*(1), 2–10. doi:10.1109/TNSM.2004.4623689

Bifulco, R., y Schneider, F. (2013). OpenFlow Rules Interactions: Definition and Detection. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for* (pp. 1–6). doi:10.1109/SDN4FNS.2013.6702547

Canini, M., Venzano, D., Peresini, P., Kostic, D., Rexford, J., y otros. (2012). A NICE Way to Test OpenFlow Applications. In *NSDI* (pp. 127–140). Obtenido de https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final105.pdf

Casado, M. (2014). List of OpenFlow Software Projects. Obtenido Junio 18, 2014, de http://yuba.stanford.edu/~casado/of-sw.html

Dusi, M., Bifulco, R., Gringoli, F., y Schneider, F. (2014). Reactive logic in software-defined networking: Measuring flow-table requirements. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International* (pp. 340–345). doi:10.1109/IWCMC.2014.6906380

Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., y Walker, D. (2011). Frenetic: A network programming language. In *ACM SIGPLAN Notices* (Vol. 46, pp. 279–291). ACM. Obtenido de http://dl.acm.org/citation.cfm?id=2034812

Gawanmeh, A., y Tahar, S. (2012). Novel algorithm for detecting conflicts in firewall rules. In *Electrical & Computer Engineering (CCECE), 2012 25th IEEE Canadian Conference on* (pp. 1–4). IEEE. Obtenido de http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6334998

Hu, H., Ahn, G.-J., y Kulkarni, K. (2012). Detecting and Resolving Firewall Policy Anomalies. *IEEE Transactions on Dependable and Secure Computing*, *9*(3), 318–331. doi:10.1109/TDSC.2012.20

Iyer, A. S., Mann, V., y Samineni, N. R. (2013). switchReduce: Reducing switch state and controller involvement in OpenFlow networks. In *IFIP Networking Conference, 2013* (pp. 1–9).

Jarschel, M., Oechsner, S., Schlosser, D., Pries, R., Goll, S., y Tran-Gia, P. (2011). Modeling and performance evaluation of an OpenFlow architecture. In *Teletraffic Congress (ITC), 2011 23rd International* (pp. 1–7).

Kazemian, P., Varghese, G., y McKeown, N. (2012). Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (pp. 9–9). Berkeley, CA, USA: USENIX Association. Obtenido de http://dl.acm.org/citation.cfm?id=2228298.2228311

Khummanee, S., Khumseela, A., y Puangpronpitag, S. (2013). Towards a new design of firewall: Anomaly elimination and fast verifying of firewall rules. In *Computer Science and Software Engineering (JCSSE), 2013 10th International Joint Conference on* (pp. 93–98). IEEE. Obtenido de http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6567326

Li, W., Wan, H., y Li, S. (2013). An approach to the generalization of firewall rules. In *Computer and Information Science (ICIS), 2013 IEEE/ACIS 12th International Conference on* (pp. 201–206). IEEE. Obtenido de http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6607841

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., … Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, *38*(2), 69–74.

Monsanto, C., Foster, N., Harrison, R., y Walker, D. (2012). A compiler and run-time system for network programming languages. *ACM SIGPLAN Notices*, *47*(1), 217–230.

Montgomery, D. C. (2008). *Design and Analysis of Experiments*. John Wiley & Sons. Obtenido de http://books.google.co.cr/books?id=kMMJAm5bD34C

Nelson, T., Ferguson, A. D., Scheer, M. J. G., y Krishnamurthi, S. (2014). Tierless Programming and Reasoning for Software-defined Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (pp. 519–531). Berkeley, CA, USA: USENIX Association. Obtenido de http://dl.acm.org/citation.cfm?id=2616448.2616496

Nelson, T., Guha, A., Dougherty, D. J., Fisler, K., y Krishnamurthi, S. (2013). A balance of power: Expressive, analyzable controller programming. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (pp. 79–84). ACM. Obtenido de http://dl.acm.org/citation.cfm?id=2491201

Open Networking Foundation. (2013, October 14). OpenFlow switch Specification 1.4.0. Open Networking Foundation. Obtenido de https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

Open Networking Foundation. (2014). Software-Defined Networking (SDN) Definition. Obtenido julio 20, 2014, de https://www.opennetworking.org/sdn-resources/sdn-definition

Oracle. (2014). HashMap (Java Platform SE 8). Obtenido noviembre 27, 2014, de http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

Perešíni, P., & Canini, M. (2011). Is your OpenFlow application correct? In *Proceedings of The ACM CoNEXT Student Workshop* (p. 18). ACM. Obtenido de http://dl.acm.org/citation.cfm?id=2079345

Pozo, S., Varela-Vaca, A. J., Gasca, R. M., & Ceballos, R. (2009). Efficient Algorithms and Abstract Data Types for Local Inconsistency Isolation in Firewall ACLs. In *SECRYPT* (pp. 42–53). Obtenido de http://www.lsi.us.es/~quivir/sergio/SECRYPT09.pdf

Project Floodlight. (2014). Open Source Software for Building Software-Defined Networks. Obtenido julio 17, 2014, de http://www.projectfloodlight.org/projects/

RStudio. (2014). RStudio. Obtenido de http://www.rstudio.com/products/rstudio/

The R Foundation. (2014). The R Project for Statistical Computing. Obtenido Octubre 8, 2014, de http://www.r-project.org/

Vishnoi, A., Poddar, R., Mann, V., y Bhattacharya, S. (2014). Effective switch memory management in OpenFlow networks (pp. 177–188). ACM Press. doi:10.1145/2611286.2611301

Voellmy, A., Wang, J., Yang, Y. R., Ford, B., y Hudak, P. (2013). Maple: Simplifying SDN programming using algorithmic policies. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (pp. 87–98). ACM. Obtenido de http://dl.acm.org/citation.cfm?id=2486030

Wang, W., Ji, R., Chen, W., Chen, B., y Li, Z. (2007). Firewall Rules Sorting Based on Markov Model (pp. 203–208). IEEE. doi:10.1109/ISDPE.2007.40

Yan, X., Zhaoxia, W., y Qingrong, T. (2013). Efficient Algorithm for Detecting Firewall Rule Conflict (pp. 340–343). IEEE. doi:10.1109/INCoS.2013.63

ANEXOS

ANEXO A: CÓDIGO FUENTE DE LOS ALGORITMOS

El código fuente está estructurado de forma que los algoritmos sean independientes de los experimentos. En la Figura A.1 se muestra la estructura del código.

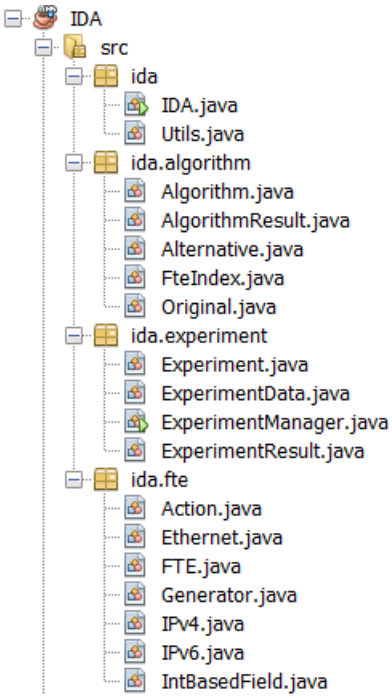

Figura A.1 Estructura del código fuente del proyecto

A continuación se presenta el código fuente de los algoritmos, el resto del código se encuentra disponible en la siguiente dirección web: https://github.com/ovasquez/ida.

Código fuente del algoritmo original

El código fuente del algoritmo original se encuentra en los archivos *Original.java* y *Algorithm.java*, y se muestra en la Figura A.1 y la Figura A.2 respectivamente.

Original.java

```
import ida.fte.FTE;
import java.util.ArrayList;
import java.util.LinkedList;
```

```
public class Original extends ida.algorithm.Algorithm {
    public Original() {

    }

    public LinkedList<AlgorithmResult> interactionDetection(FTE newFte, ArrayList<FTE>
fteArrayList) {
        // Using liked list for O(1) add operation
        LinkedList<AlgorithmResult> result = new LinkedList<>();

        for(int i = 0; i < fteArrayList.size(); i++) {
            InteractionType interaction = interactionDetection(newFte,
fteArrayList.get(i));
            if (!interaction.equals(InteractionType.NONE)) {
                result.add(new AlgorithmResult(i, interaction));
            }
        }
        return result;
    }
}
```

Figura A.1 Código fuente del archivo Original.java.

Algorithm.java

```
import ida.Utils;
import ida.fte.Action;
import ida.fte.FTE;
import java.util.ArrayList;
import java.util.List;

public class Algorithm {

    public enum AlgorithmType {
        ORIGINAL,
        ALTERNATIVE,
    }

    public enum InteractionType {
        NONE,
        DUPLICATION,
        REDUNDANCY,
        GENERALIZATION,
        SHADOWING,
        CORRELATION,
```

```
        INCLUSION,
        EXTENSION,
    }

    public enum FieldRelationType {
        DISJOINT,
        EQUAL,
        SUBSET,
        SUPERSET
    }

    public enum MatchSetRelationType {
        UNDETERMINED,
        DISJOINT,
        EXACT,
        SUBSET,
        SUPERSET,
        CORRELATED,
    }

    public enum ActionRelationType {
        DISJOINT,
        EQUAL,
        RELATED,
    }

    public enum ActionSetRelationType {
        UNDETERMINED,
        DISJOINT,
        RELATED,
        SUBSET,
        SUPERSET,
        EQUAL,
    }

    public InteractionType interactionDetection(FTE fte1, FTE fte2) {
        InteractionType result = InteractionType.NONE;

        // fteX has lower (or equal) priorty than fteY to fulfill the Interactions
Requirement
        FTE fteX, fteY;
        if(fte1.getPriority() < fte2.getPriority()) {
            fteX = fte1;
            fteY = fte2;
```

```
        }
        else {
            fteX = fte2;
            fteY = fte1;
        }

        MatchSetRelationType matchSetRelation = this.matchSetRelations(fteX, fteY);
        ActionSetRelationType actionSetRelation = this.actionSetRelations(fteX, fteY);

        if( fte1.getPriority() == fte2.getPriority() &&
            matchSetRelation.equals(MatchSetRelationType.EXACT) &&
            actionSetRelation.equals(ActionSetRelationType.EQUAL)) {
            result = InteractionType.DUPLICATION;
        }
        else if (!matchSetRelation.equals(MatchSetRelationType.DISJOINT)) {
            switch (matchSetRelation) {
                case CORRELATED: {
                    if(actionSetRelation.equals(ActionSetRelationType.EQUAL)) {
                        result = InteractionType.REDUNDANCY;
                    }
                    else {
                        result = InteractionType.CORRELATION;
                    }
                    break;
                }
                case SUPERSET: {
                    if(actionSetRelation.equals(ActionSetRelationType.EQUAL)) {
                        result = InteractionType.REDUNDANCY;
                    }
                    else if(actionSetRelation.equals(ActionSetRelationType.SUPERSET)) {
                        result = InteractionType.EXTENSION;
                    }
                    else {
                        result = InteractionType.GENERALIZATION;
                    }
                    break;
                }
                case EXACT: {
                    if(actionSetRelation.equals(ActionSetRelationType.EQUAL)) {
                        result = InteractionType.REDUNDANCY;
                    }
                    else if(actionSetRelation.equals(ActionSetRelationType.SUBSET)) {
                        result = InteractionType.INCLUSION;
                    }
```

```
                    else {
                        result = InteractionType.SHADOWING;
                    }
                    break;
                }
                case SUBSET: {
                    if(actionSetRelation.equals(ActionSetRelationType.EQUAL)) {
                        result = InteractionType.REDUNDANCY;
                    }
                    else if(actionSetRelation.equals(ActionSetRelationType.SUBSET)) {
                        result = InteractionType.INCLUSION;
                    }
                    else {
                        result = InteractionType.SHADOWING;
                    }
                    break;
                }
            }
        }
        return result;
    }

    ActionSetRelationType actionSetRelations(FTE fte1, FTE fte2) {
        ActionSetRelationType result = ActionSetRelationType.UNDETERMINED;

        List<ActionRelationType> actionRelations = compareActions(fte1, fte2);

        int actionsCount1 = fte1.getActions().length;
        int actionsCount2 = fte2.getActions().length;

        boolean actionsCountIsEqual = actionsCount1 == actionsCount2;

        for (ActionRelationType relation : actionRelations) {
            switch (relation) {
                case EQUAL: {
                    if(result.equals(ActionSetRelationType.UNDETERMINED) ||
result.equals(ActionSetRelationType.EQUAL)) {
                    result = ActionSetRelationType.EQUAL;
                    }
                    else {
                        result = ActionSetRelationType.RELATED;
                    }
                    break;
                }
```

```
                case RELATED: {
                    result = ActionSetRelationType.RELATED;
                }
                case DISJOINT: {
                    if(result.equals(ActionSetRelationType.UNDETERMINED)) {
                        result = ActionSetRelationType.DISJOINT;
                    }
                }
            }
        }

        if (!actionsCountIsEqual && result.equals(ActionSetRelationType.EQUAL)) {
            if(actionsCount1 > actionsCount2) {
                result = ActionSetRelationType.SUPERSET;
            }
            else {
                result = ActionSetRelationType.SUBSET;
            }
        }

        return result;
    }

    List<ActionRelationType> compareActions(FTE fte1, FTE fte2) {
        List<ActionRelationType> result = new ArrayList<>();

        Action[] actions1 = fte1.getActions();
        Action[] actions2 = fte2.getActions();

        for (Action action1 : actions1) {
            ActionRelationType relation = ActionRelationType.DISJOINT;
            for (Action action2 : actions2) {
                if(action1.getType().equals(action2.getType())) {
                    if(action1.getValue() == action2.getValue()){
                        relation = ActionRelationType.EQUAL;
                    }
                    else {
                        relation = ActionRelationType.RELATED;
                    }
                    break;
                }
            }
            result.add(relation);
        }
```

```
        return result;
    }


    MatchSetRelationType matchSetRelations(FTE fte1, FTE fte2) {
        MatchSetRelationType result = MatchSetRelationType.UNDETERMINED;


        List<FieldRelationType> fieldRelations = Utils.compareFTEs(fte1, fte2);


        for (FieldRelationType fieldRelation : fieldRelations) {
            if (fieldRelation.equals(FieldRelationType.EQUAL)) {
                if (result.equals(MatchSetRelationType.UNDETERMINED)){
                    result = MatchSetRelationType.EXACT;
                }
            }
            else if (fieldRelation.equals(FieldRelationType.SUPERSET)) {
                if (result.equals(MatchSetRelationType.SUBSET) ||
result.equals(MatchSetRelationType.CORRELATED)) {
                    result = MatchSetRelationType.CORRELATED;
                }
                else if (!result.equals(MatchSetRelationType.DISJOINT)) {
                    result = MatchSetRelationType.SUPERSET;
                }
            }
            else if (fieldRelation.equals(FieldRelationType.SUBSET)) {
                if (result.equals(MatchSetRelationType.SUPERSET) ||
result.equals(MatchSetRelationType.CORRELATED)) {
                    result = MatchSetRelationType.CORRELATED;
                }
                else if (!result.equals(MatchSetRelationType.DISJOINT)) {
                    result = MatchSetRelationType.SUBSET;
                }
            }
            else {
                result = MatchSetRelationType.DISJOINT;
            }
        }
        return result;
    }
}
```

Figura A.2 Código fuente del archivo Algorithm.java

Código fuente del algoritmo alternativo.

El código fuente del algoritmo original se encuentra en los archivos *Alternative.java* (que sobrecarga algunas de las funciones definidas en Algorithm.java) y *FteIndex.java* (estructura para indizar los campos), y se muestras en la Figura A.4 y la Figura A.5 respectivamente.

Alternative.java

```
package ida.algorithm;

import ida.Utils;
import ida.fte.FTE;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.UUID;

public class Alternative extends ida.algorithm.Algorithm {

    /**
     * HashMap of the FTEs
     */
    private HashMap<UUID, FTE> fteHash;

    /**
     * Structure containing the index of the FTEs for faster searches
     */
    private FteIndex fteIndex;

    public Alternative() {

    }

    public HashMap<UUID, FTE> getFteHash() {
        return fteHash;
    }

    public void setFteHash(HashMap<UUID, FTE> fteHash) {
        this.fteHash = fteHash;
    }

    /**
```

```
     * Creates the FTE Index structure for the current FTE HashMap
     */
    public void createFteIndex() {
        if (this.fteHash != null) {
            this.fteIndex = new FteIndex(this.fteHash);
        }
    }

    /**
     * Optimized version of the published algorithm without the use of index
     * @param newFte
     * @param fteArrayList
     * @return
     */
    public LinkedList<AlgorithmResult> interactionDetection(FTE newFte, ArrayList<FTE>
fteArrayList) {

        LinkedList<AlgorithmResult> result = new LinkedList<>();

        for(int i = 0; i < fteArrayList.size(); i++) {
            InteractionType interaction = this.interactionDetection(newFte,
fteArrayList.get(i));
            if (!interaction.equals(InteractionType.NONE)) {
                result.add(new AlgorithmResult(i, interaction));
            }
        }
        return result;
    }

    /**
     * Optimized version of the algorithm using indexes to reduce the search space
     * for FTEs that could have an interaction
     * @param newFte
     * @param fteHashMap
     * @return
     */
    public LinkedList<AlgorithmResult> interactionDetection(FTE newFte, HashMap<UUID, FTE>
fteHashMap) {

        LinkedList<AlgorithmResult> result = new LinkedList<>();

        // Used to make sure we get the smallest HashSet to compare
        int sizeHS = 0;
```

```
        // Used when no matches are found for a defined Field, so the search must be done
only in the WildCard Hash
        int sizeWC = 0;

        HashSet<UUID> tempHS, tempWC, smallHS, wcHS;
        smallHS = wcHS = null;
        boolean foundNoneWC = false;


        if (!newFte.inPort.wildcard) {
            foundNoneWC = true;
            tempHS = this.fteIndex.hashInPort.get(newFte.inPort.value);
            tempWC = this.fteIndex.hashInPort.get(FteIndex.WILDCARD_HASH);
            if (tempHS != null) {
                if (sizeHS == 0 || tempHS.size() < sizeHS) {
                   smallHS = tempHS;
                   wcHS = tempWC;
                   sizeHS = tempHS.size();
                }
            }
            else if (wcHS == null || (tempWC != null && tempWC.size() < sizeWC)) {
                wcHS = tempWC;
            }
        }

        if (!newFte.ethType.wildcard) {
            foundNoneWC = true;
            tempHS = this.fteIndex.hashEthType.get(newFte.ethType.value);
            tempWC = this.fteIndex.hashEthType.get(FteIndex.WILDCARD_HASH);
            if (tempHS != null) {
                if (sizeHS == 0 || tempHS.size() < sizeHS) {
                   smallHS = tempHS;
                   wcHS = tempWC;
                   sizeHS = tempHS.size();
                }
            }
            else if (wcHS == null || (tempWC != null && tempWC.size() < sizeWC)) {
                wcHS = tempWC;
            }
        }

        if (!newFte.ipProto.wildcard) {
            foundNoneWC = true;
            tempHS = this.fteIndex.hashIpProto.get(newFte.ipProto.value);
```

```
            tempWC = this.fteIndex.hashIpProto.get(FteIndex.WILDCARD_HASH);
            if (tempHS != null) {
                if (sizeHS == 0 || tempHS.size() < sizeHS) {
                   smallHS = tempHS;
                   wcHS = tempWC;
                   sizeHS = tempHS.size();
                }
            }
            else if (wcHS == null || (tempWC != null && tempWC.size() < sizeWC)) {
                wcHS = tempWC;
            }
        }

        if (!newFte.tcpSrc.wildcard) {
            foundNoneWC = true;
            tempHS = this.fteIndex.hashTcpSrc.get(newFte.tcpSrc.value);
            tempWC = this.fteIndex.hashTcpSrc.get(FteIndex.WILDCARD_HASH);
            if (tempHS != null) {
                if (sizeHS == 0 || tempHS.size() < sizeHS) {
                   smallHS = tempHS;
                   wcHS = tempWC;
                   sizeHS = tempHS.size();
                }
            }
            else if (wcHS == null || (tempWC != null && tempWC.size() < sizeWC)) {
                wcHS = tempWC;
            }
        }

        if (!newFte.tcpDst.wildcard) {
            foundNoneWC = true;
            tempHS = this.fteIndex.hashTcpDst.get(newFte.tcpDst.value);
            tempWC = this.fteIndex.hashTcpDst.get(FteIndex.WILDCARD_HASH);
            if (tempHS != null) {
                if (sizeHS == 0 || tempHS.size() < sizeHS) {
                   smallHS = tempHS;
                   wcHS = tempWC;
                   sizeHS = tempHS.size();
                }
            }
            else if (wcHS == null || (tempWC != null && tempWC.size() < sizeWC)) {
                wcHS = tempWC;
            }
        }
```

```
        if (!newFte.udpSrc.wildcard) {
            foundNoneWC = true;
            tempHS = this.fteIndex.hashUdpSrc.get(newFte.udpSrc.value);
            tempWC = this.fteIndex.hashUdpSrc.get(FteIndex.WILDCARD_HASH);
            if (tempHS != null) {
                if (sizeHS == 0 || tempHS.size() < sizeHS) {
                   smallHS = tempHS;
                   wcHS = tempWC;
                   sizeHS = tempHS.size();
                }
            }
            else if (wcHS == null || (tempWC != null && tempWC.size() < sizeWC)) {
                wcHS = tempWC;
            }
        }

        if (!newFte.udpDst.wildcard) {
            foundNoneWC = true;
            tempHS = this.fteIndex.hashUdpDst.get(newFte.udpDst.value);
            tempWC = this.fteIndex.hashUdpDst.get(FteIndex.WILDCARD_HASH);
            if (tempHS != null) {
                if (sizeHS == 0 || tempHS.size() < sizeHS) {
                   smallHS = tempHS;
                   wcHS = tempWC;
                   //sizeHS = tempHS.size();
                }
            }
            else if (wcHS == null || (tempWC != null && tempWC.size() < sizeWC)) {
                wcHS = tempWC;
            }
        }

        if(smallHS != null || wcHS != null) {
            if(smallHS != null){
                for(UUID uuid : smallHS) {
                    InteractionType interaction = this.interactionDetection(newFte,
this.fteHash.get(uuid));
                    if (!interaction.equals(InteractionType.NONE)) {
                        result.add(new AlgorithmResult(uuid, interaction));
                    }
                }
            }
```

```
            if(wcHS != null){
                for(UUID uuid : wcHS) {
                    InteractionType interaction = this.interactionDetection(newFte,
this.fteHash.get(uuid));
                    if (!interaction.equals(InteractionType.NONE)) {
                        result.add(new AlgorithmResult(uuid, interaction));
                    }
                }
            }
        }
        else if (!foundNoneWC) {


            // No filters were found, so loop all
            for(UUID uuid : fteHash.keySet()) {
                InteractionType interaction = this.interactionDetection(newFte,
this.fteHash.get(uuid));
                if (!interaction.equals(InteractionType.NONE)) {
                    result.add(new AlgorithmResult(uuid, interaction));
                }
            }
        }
        else {
            // No compatible FTEs were found
        }

        return result;
    }

    private void combineHS(HashSet<UUID> smallHS, HashSet<UUID> otherHS) {
        for(UUID uuid : smallHS){
            if(!otherHS.contains(uuid)){
                smallHS.remove(uuid);
            }
        }
    }

    /*
     * OVERRIDES WITH OPTIMIZATIONS OF THE PUBLISHED ALGORITHMS FOR FTEs WITH NO
     * INTERACTIONS
     */

    @Override
    public Algorithm.InteractionType interactionDetection(FTE fte1, FTE fte2) {
```

```
        Algorithm.InteractionType result = Algorithm.InteractionType.NONE;

        // fteX has lower (or equal) priorty than fteY to fulfill the Interactions
Requirement
        FTE fteX, fteY;
        if(fte1.getPriority() < fte2.getPriority()) {
            fteX = fte1;
            fteY = fte2;
        }
        else {
            fteX = fte2;
            fteY = fte1;
        }

        Algorithm.MatchSetRelationType matchSetRelation = this.matchSetRelations(fteX,
fteY);
        // Optimizing for FTEs with no interaction
        if(matchSetRelation.equals(MatchSetRelationType.DISJOINT)){
            return result;
        }


        Algorithm.ActionSetRelationType actionSetRelation = this.actionSetRelations(fteX,
fteY);

        if( fte1.getPriority() == fte2.getPriority() &&
            matchSetRelation.equals(Algorithm.MatchSetRelationType.EXACT) &&
            actionSetRelation.equals(Algorithm.ActionSetRelationType.EQUAL)) {
            result = Algorithm.InteractionType.DUPLICATION;
        }
        else if (!matchSetRelation.equals(Algorithm.MatchSetRelationType.DISJOINT)) {
            switch (matchSetRelation) {
                case CORRELATED: {
                    if(actionSetRelation.equals(Algorithm.ActionSetRelationType.EQUAL)) {
                        result = Algorithm.InteractionType.REDUNDANCY;
                    }
                    else {
                        result = Algorithm.InteractionType.CORRELATION;
                    }
                    break;
                }
                case SUPERSET: {
                    if(actionSetRelation.equals(Algorithm.ActionSetRelationType.EQUAL)) {
                        result = Algorithm.InteractionType.REDUNDANCY;
```

```
                    }
                    else
if(actionSetRelation.equals(Algorithm.ActionSetRelationType.SUPERSET)) {
                        result = Algorithm.InteractionType.EXTENSION;
                    }
                    else {
                        result = Algorithm.InteractionType.GENERALIZATION;
                    }
                    break;
                }
                case EXACT: {
                    if(actionSetRelation.equals(Algorithm.ActionSetRelationType.EQUAL)) {
                        result = Algorithm.InteractionType.REDUNDANCY;
                    }
                    else
if(actionSetRelation.equals(Algorithm.ActionSetRelationType.SUBSET)) {
                        result = Algorithm.InteractionType.INCLUSION;
                    }
                    else {
                        result = Algorithm.InteractionType.SHADOWING;
                    }
                    break;
                }
                case SUBSET: {
                    if(actionSetRelation.equals(Algorithm.ActionSetRelationType.EQUAL)) {
                        result = Algorithm.InteractionType.REDUNDANCY;
                    }
                    else
if(actionSetRelation.equals(Algorithm.ActionSetRelationType.SUBSET)) {
                        result = Algorithm.InteractionType.INCLUSION;
                    }
                    else {
                        result = Algorithm.InteractionType.SHADOWING;
                    }
                    break;
                }
            }
        }

        return result;
    }

    @Override
    MatchSetRelationType matchSetRelations(FTE fte1, FTE fte2) {
```

```
        MatchSetRelationType result = MatchSetRelationType.UNDETERMINED;

        List<FieldRelationType> fieldRelations = Utils.compareFTEsOptimized(fte1, fte2);

        // Optimizing for FTEs with no interaction.
        // Don't waste time if the field relations have a disjoint value, and thus the
variable is null
        if(fieldRelations == null) {
            return MatchSetRelationType.DISJOINT;
        }

        for (FieldRelationType fieldRelation : fieldRelations) {
            if (fieldRelation.equals(FieldRelationType.EQUAL)) {
                if (result.equals(MatchSetRelationType.UNDETERMINED)){
                    result = MatchSetRelationType.EXACT;
                }
            }
            else if (fieldRelation.equals(FieldRelationType.SUPERSET)) {
                if (result.equals(MatchSetRelationType.SUBSET) ||
result.equals(MatchSetRelationType.CORRELATED)) {
                    result = MatchSetRelationType.CORRELATED;
                }
                else if (!result.equals(MatchSetRelationType.DISJOINT)) {
                    result = MatchSetRelationType.SUPERSET;
                }
            }
            else if (fieldRelation.equals(FieldRelationType.SUBSET)) {
                if (result.equals(MatchSetRelationType.SUPERSET) ||
result.equals(MatchSetRelationType.CORRELATED)) {
                    result = MatchSetRelationType.CORRELATED;
                }
                else if (!result.equals(MatchSetRelationType.DISJOINT)) {
                    result = MatchSetRelationType.SUBSET;
                }
            }
            else {
                result = MatchSetRelationType.DISJOINT;
            }
        }

        return result;
    }
}
```

Figura A.3 Código fuente del archivo Alternative.java

FteIndex.java

```
package ida.algorithm;


import ida.fte.FTE;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map.Entry;
import java.util.UUID;


/**
 * Data structure used by the alternative algorithm for faster searches of
 * relevant FTEs
 *
 * @author vasqueos
 */
public class FteIndex {


    public static int WILDCARD_HASH = -1;


    /**
     * Container of the FTEs
     */
    public HashMap<UUID, FTE> fteHash;


    /**
     * HashMaps for each IntBasedField of the FTEs
     */
    public HashMap<Integer, HashSet<UUID>> hashInPort;
    public HashMap<Integer, HashSet<UUID>> hashEthType;
    public HashMap<Integer, HashSet<UUID>> hashIpProto;
    public HashMap<Integer, HashSet<UUID>> hashTcpSrc;
    public HashMap<Integer, HashSet<UUID>> hashTcpDst;
    public HashMap<Integer, HashSet<UUID>> hashUdpSrc;
    public HashMap<Integer, HashSet<UUID>> hashUdpDst;


    public FteIndex(HashMap<UUID, FTE> fteHash) {
        this.fteHash = fteHash;
```

```
        this.hashInPort = new HashMap<>(Short.MAX_VALUE);
        this.hashEthType = new HashMap<>(Short.MAX_VALUE);
        this.hashIpProto = new HashMap<>(Short.MAX_VALUE);
        this.hashTcpSrc = new HashMap<>(Short.MAX_VALUE);
        this.hashTcpDst = new HashMap<>(Short.MAX_VALUE);
        this.hashUdpSrc = new HashMap<>(Short.MAX_VALUE);
        this.hashUdpDst = new HashMap<>(Short.MAX_VALUE);


        fillHashMaps(this.fteHash);
    }


    /**
     * Inserts the values of the FTEs in the HashMaps of IntBasedFields
     */
    private void fillHashMaps(HashMap<UUID, FTE> fteHash) {
        FTE tempFTE;
        for (Entry<UUID, FTE> entry : fteHash.entrySet()) {
            tempFTE = entry.getValue();
            addFteToIndex(tempFTE);
        }
    }


    /**
     * Adds the IntBasedFields of the FTE to this FteIndex structure
     *
     * @param fte
     */
    private void addFteToIndex(FTE fte) {
        int value = fte.inPort.isWildcard() ? WILDCARD_HASH : fte.inPort.value;
        this.addIbfToHashMap(value, fte.id, this.hashInPort);


        value = fte.ethType.isWildcard() ? WILDCARD_HASH : fte.ethType.value;
        this.addIbfToHashMap(value, fte.id, this.hashEthType);


        value = fte.ipProto.isWildcard() ? WILDCARD_HASH : fte.ipProto.value;
        this.addIbfToHashMap(value, fte.id, this.hashIpProto);
```

```
        value = fte.tcpSrc.isWildcard() ? WILDCARD_HASH : fte.tcpSrc.value;
        this.addIbfToHashMap(value, fte.id, this.hashTcpSrc);


        value = fte.tcpDst.isWildcard() ? WILDCARD_HASH : fte.tcpDst.value;
        this.addIbfToHashMap(value, fte.id, this.hashTcpDst);


        value = fte.udpSrc.isWildcard() ? WILDCARD_HASH : fte.udpSrc.value;
        this.addIbfToHashMap(value, fte.id, this.hashUdpSrc);


        value = fte.udpDst.isWildcard() ? WILDCARD_HASH : fte.udpDst.value;
        this.addIbfToHashMap(value, fte.id, this.hashUdpDst);
    }


    /**
     * Adds the value of an IntBasedField to a HashMap<Integer, HashSet<UUID>>.
     * This should make it faster to find the IDs of FTEs related to that value.
     *
     * @param value
     * @param id
     * @param hash
     */
    private void addIbfToHashMap(int value, UUID id, HashMap<Integer, HashSet<UUID>> hash)
{
        HashSet<UUID> set;


        // Check if the HashSet already exists, otherwise create it
        if (hash.get(value) != null) {
            set = hash.get(value);
        } else {
            set = new HashSet<>();
            hash.put(value, set);
        }
        set.add(id);
    }
}
```

Figura A.4 Código fuente del archivo FteIndex.java