POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master Thesis

A framework for Virtual Network Functions (VNF) modeling and Service Graph verification in SDN/Cloud context

Supervisors
prof. Riccardo Sisto prof. Guido Marchetto

Candidate
Rui Zhao

February 2018

To my family.

Contents

Chapter 1

Introduction

With the explosion of intelligent hardware, a large number of applications access the 4G network, and the traffic demand of people is surging like a tsunami. But dedicated hardware corresponding to a dedicated service, such a cost is expensive, in order to save costs, accelerate the deployment of new network services and flexibly define the network behavior, operators tend to give up bulky expensive private network equipment, in turn using standard IT virtualization technology to split network function modules such as DNS, NAT, Firewall and so on. the creation of new paradigms like Network Function Virtualization (NFV) and Software Defined Network (SDN) comes from the purpose.

So in the presence of middle-boxs, such as web cache and DNS server, whose forwarding behaviors often depends on previously observed traffic, it is crucial to verify the network invariants that address reachability, isolation and traversal between hosts before the middle-boxs are deployed to guarantee the correctness of automatic reconfiguration of network service graphs in a cloud-like environment.

According to previous studies, great progress has been made recently in verifying different network properties in the presence of dynamic data path. Our work leverages recent advances in Z3 which is a kind of SMT solvers to be used determining the satisfiability of the First Order Logic formula simply called FOL (give a model to satisfy if necessary). Based on its advantage, we can use FOL to define the behavior rules of all the VNF and supply them as a model to Z3 to check the satisfiability. As a result, SAT means the network property is satisfied and UNSAT means the property can not be satisfied.

the main challenge lies in scaling the approach to handle large and more complicated network functions, We address by developing more VNFs to complete this framework and increase its flexibilities and funtionalities, which allow larger network-wide verification.

The objective of this thesis is to allow the interested actors to define the behavior of any VNF in a more developer-friendly Java-like fashion and allow the extraction of an abstract model from the Java code in order to verify the important network

properties in a wider network environment.

This thesis is structured as follows:

- **Chapter 2:** the reasons of the project.
- **Chapter 3:** introduce all tools and Acknowledge needed before starting the thesis.
- **Chapter 4:** describes developted java library used by interested actor to create abstract VNF models .
- **Chapter 5:** details the framework implementation(parser).
- **Chapter 6:** introduce the rules, test cases and results for each one
- **Chapter 7:** exposes the conclusions and provides future work that will follow this thesis work.
- **Chapter 8:** Bibliography

Chapter 2

Background

2.1 NFV

Traditional network virtualization deployment requires manual hop-by-hop deployment, which is inefficient and labor-intensive. In scenarios such as data centers, automated deployment must be used for rapid deployment and dynamic adjustment. Of course, you can implement network virtualization through SDN.

Network virtualization technology, which uses software to install, control, and operate network functions running on general-purpose hardware, integrates cloud and virtualization technologies, enabling next-generation network services to have better scalability and automation capabilities.

Using NFV can reduce or even remove middleware deployed in existing networks. It enables a single physical platform to run different applications. Users and tenants can use network functions through multi-version and multi-tenancy. These new emerging technologies are often referred to indiscriminately as NFV (Network Function Virtualization) and SDN (Software-Defined Networking). Although the two have gradually converged, the original intention and architecture of the two are not the same. So some operators jointly set up the European Telecommunications Standards Institute (ETSI), and one of his working groups (ETSI ISG NFV) is responsible for developing and developing a virtualized architecture for telecommunications networks, such as NFV MANO. ETSI NFV Standard Architecture includes NFV infrastructure (NFVI), MANO (Management and Orchestration), and VNFs, which are the top conceptual entities in the standard architecture.

This design achieves the following goals:

1. The NFV architecture separates some of the functions of the physical network element. This makes it easier for operators to choose the most suitable VNF from multiple vendors.

2. VNF can be used for different physical hardware and hypervisors.

3. Can be released quickly through software only.

4. The standard open interface facilitates the interaction of VNFs between multi-vendors.

5. Use low-cost, general-purpose hardware that is not subject to specific vendors.

2.2 SDN

Some people think that SDN and network virtualization are at the same level, but this is a wrong claim. SDN is not network virtualization, and network virtualization is not SDN. SDN is a centralized control network architecture that divides the network into data and control planes. Network virtualization is a kind of network technology that can create a virtual network in a physical topology.

SDN originated in the campus network and is developed in the data center. SDN takes the network as a whole platform, separates the network control plane from the data forwarding plane, and implements programmable control. In the control plane, the network operating system is the controller. It controls all APPs as well as security policies. In the data plane, the network infrastructure includes OpenFlow switches. In addition to the flow table, each switch is a switching CPU. The connection between the control plane and the data plane is via an application program interface, such as OpenFlow, which follows the OpenFlow protocol. SDN controllers are increasingly built on open platforms that use open standards and open APIs to enable them to orchestrate, manage, and control network devices from different vendors.

2.3 SP-DevOps

SP-DevOps(a clipped compound of "development" and "operations"). Mainly due to historical reasons (most of the operation and maintenance personnel come from the fields of hardware and telecom services), operation and maintenance personnel and developers belong to different branches of the organizational structure. Developers belong to the R & D department, and operation and maintenance personnel belong to the infrastructure department (or specialized operation and maintenance department) most of the time. Do not forget, they have different purposes:

1. developer wants to maximize change;

2. operator wants to optimizestability.

In this case, DevOps comes about as the software industry is increasingly recognizing that in order to deliver software products and services on time, development and

operations must work closely together. DevOps, a combination of Development and Operations in English, is a generic term for a group of processes, methods, and systems used to facilitate communication, collaboration, and integration between development (application / software engineering), technology operations and quality assurance (QA) departments.

2.4 Motivation

Take a look at the current location of these online giants in this area, to give you a few examples:

1. Facebook has thousands of development and operations staff and tens of thousands of servers. On average, one operator is responsible for 500 servers (do you think Automation is optional?) They are deployed twice a day (the concept of a ring deployment, the deployment ring).

2. Flickr deployed 10 times a day.

3. Netflix clearly designed for a variety of failures! Their software is designed to withstand system failures from the bottom up and they test thoroughly in a production environment: perform 65,000 failed tests in a production environment each day by shutting down virtual machines at random ... and ensuring that in this case Everything can still work normally.

A typical O & M team spends nearly half (47%) of its time on deployment-related work:

1. Perform actual deployment work, or

2. Fix problems related to deployment.

So successful deployment means that software can run as expected in a production environment. Failure to successfully deploy means that something went wrong, and you may need to do the necessary analysis to understand where the error occurred during deployment, whether you need to apply a patch or need to modify some of the configurations.

Chapter 3

Tools And Acknowledge

In this chapter we introduce all tools and Acknowledge needed before starting the thesis.

3.1 Z3 and FOL

First Order Logic (FOL) [1]has a wide range of applications in computer science. It is not only the important foundation for the study of programming theory and program logic, but also a powerful tools for the correctness of the program proof, theorem machine proofs, and knowledge representation. The well-formed formula of first-order logic recursively defines the formula in a formalized first-order language. It is a formal language of first–order logic, including logical symbols and non–logical symbols. In our thesis, we mainly use the logical symbols: $\forall$(for every), $\exists$(exists), $\vee$(idsjunction), $\wedge$(conjunction), $\Longrightarrow$ (implication), $=$, $\neg$, $\subset$, $\subseteq$ and so on.

In this session, i would like to introduce and explain only a complex logical symbol **implication(** $\Longrightarrow$ **)** [2] that is barely intelligible .

Well-formed formula in FOL, also known as predicate formula, is a formal language expression, that is, an expression formed by certain rules in a formal system. It is recursively defined as follows:

1. If A is a well-formed formula, then $\neg$ is also a well-formed formula;
2. If A and B are both formulas, then (A$\vee$B), (A$\wedge$B), and (A $\Longrightarrow$ B) are also a formula;
3. If A is a well-formed formula, x is a variable symbol in A, then (x)A is also a well-formed formula;

The classification of predicate formula: Let G be a predicate formula,

$*$ If there is an interpretation I that can make the formula G true (abbreviated as I satisfies G), it is said G is satisfiable;

* If all interpretations I do not satisfy (abbreviated as I fakes G), then G is called permanent or not satisfiable;

* If all the interpretations I of G can satisfy G, then G is called logically valid or tautology, that means G is always true.

But if the tautology (or constant false) formula G in the first-order logic requires that all interpretations I satisfy (fake) the formula G, due to that the interpretation I relies on a nonempty individual set D, the set D can be an infinite set, so the "all" interpretations of the so-called formula G are actually very difficult to consider, which makes the determination of the tautology in the first-order logic very difficult. However, the first-order logic is semi-decidable, that is, if the predicate formula G is true, there are algorithms that test the continuity of G in finite steps.

Logical properties of implication: Let A,B,C be all predicate formulas,

1. if A $\Longrightarrow$ B and A is tautology, then B must be tautology;

2. If A $\Longrightarrow$ B, B $\Longrightarrow$ C, then A $\Longrightarrow$ B (Distributive Property);

3. If A $\Longrightarrow$ B, A $\Longrightarrow$ C, then A $\Longrightarrow$ B$\wedge$C;

4. If A $\Longrightarrow$ B, C $\Longrightarrow$ B, then A$\vee$C $\Longrightarrow$ B.

The properties above is always used in inference .

To understand the meaning of the created FOL formulas of the network functions, I think it may be useful insert also the truth table of implication (Figure 3.1). It may be difficult to understand the truth table of implication, let's give an example in order to understand it better.

If the weather is good, then I will pick you up: p $\Longrightarrow$ q

p: If the weather is good. q: I will pick you up.

Only when p is true and q is false, then it is considered a rumor.

But if the weather is not good, according to the original words, he was not wrong. Because Since the premise cannot be judged whether it is established or not, the truth of the conclusions under the conditions will not be known.

In summary, it cannot be used as an argument. It is not a statement that can determine the authenticity, so we will always consider p $\Longrightarrow$ q true, that is why when p is false, the p $\Longrightarrow$ q is always true. If explain implication by a graphic, it will be as following graphic (Figure 3.2):

My understanding is this, the set of A $\Longrightarrow$ B can be decomposed into two parts:

1. In the intersection of A and B, satisfy "if A then B";

2. not in A, may be in B ((B-(A$\wedge$B)) B minus the intersection of A and B), or not in B (the complementary set of the union of A and B). In fact, we can simply understand: If A then the complementary set of ¬B

p	q	$p \Rightarrow q$
true	true	true
true	false	false
false	false	true
false	true	true

Figure 3.1. Truth table of Implication

The gray area in the (Figure 3.2) is the set of AB with respect of the whole aquare. **Z3** Z3 is an open-source constraint solver from Microsoft that can solve the problem of finding a set of conditions that satisfy a given condition in a given part of the constraints. The Z3 constraint solver is a universal solver for the Satisfiability modulo theories Problem. Z3 is actually commonly used in software verification, program analysis, etc. in industrial applications. However, due to its powerful functionality, it is also used in many other areasincluding the CTF field and the well-known binary analysis framework **Angr**.

In this thesis, we use Z3 to determine the satisability of the network functions represented in the form of FOL. As a result, SAT means the network property is satis ed and UNSAT means the property can not be satisfed.

3.2 AST

Throughout the development process, an Abstract Syntax Tree (AST) is used as an intermediate representation of the program, so first we must learn to establish the AST corresponding to the source code and access the AST. The Eclipse AST is an important part of the Eclipse JDT, defined in the package org.eclipse.jdt.core.dom

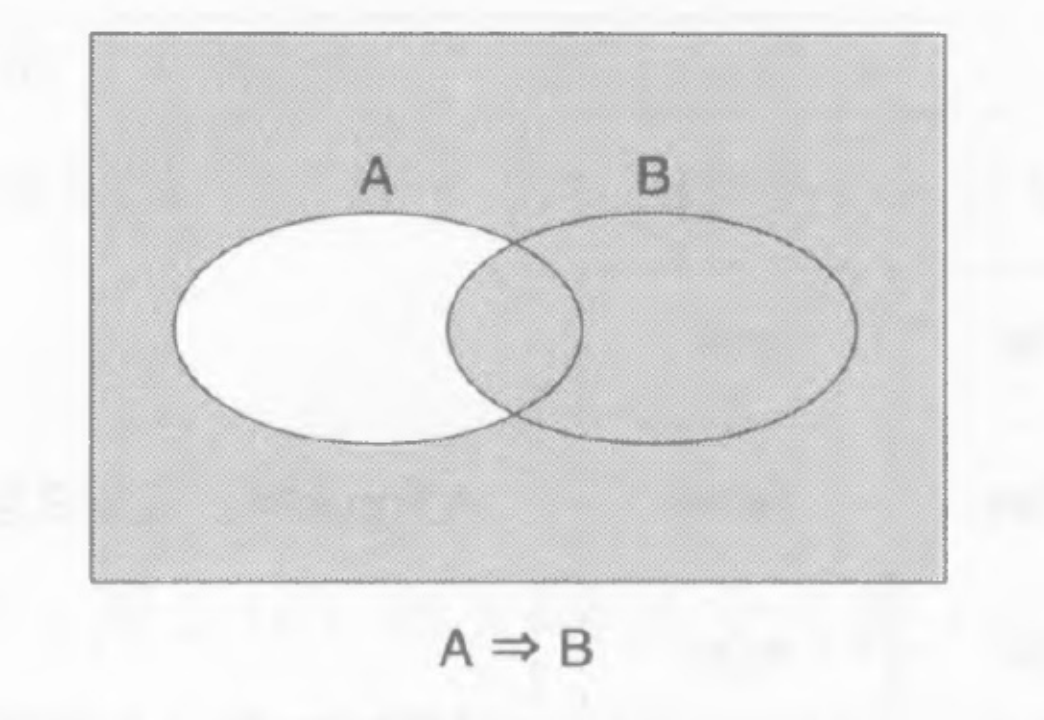

Figure 3.2. Graphic of Implication

used to represent all grammatical structures in the JAVA language. The overall structure of Eclipse AST:

* **org.eclipse.jdt.core.dom.AST (AST node class)** The Eclipse AST factory class is used to create nodes that represent various grammatical structures.

* **org.eclipse.jdt.core.dom.ASTNode and its derived classes (AST class)** It is used to represent all syntax structures in the JAVA language and is often used as a node on the AST in actual use.

* **org.eclipse.jdt.core.dom.ASTVisitor (ASTVisitor class)** The Eclipse AST visitor class defines a unified approach to accessing various nodes in the AST.

The design of this part of Eclipse AST access node adopts the visitor pattern. Different types of nodes are the specific elements to be visited, ASTNode acts as the abstract element role, ASTVisitor serves as the abstract visitor, and our own written ASTVisitor subclass acts as the specific visitor. The program code is an object structure that contains different kinds of nodes for visitors to visit. In fact, we can use ASTVisitor to iterate over all nodes without going through them one by one. A visitor is customized to iterate over all method calls. We apply it to the tree

of CompilationUnit (which can also be applied to any of the subtrees), So it walks the tree and meets the right node to work. The effect of "return true" is to tell the visitor to proceed. If the return is false, the visitor will stop. Detailed introduction:

- **AST node class** The overall structure includes CompilationUnit class (compilation unit), TypeDeclaration class (type declaration), MethodDeclaration class (method declaration);

  Statements include Block (statement block), ExpressionStatement class (expression), IfStatement (if statement), WhileStatement class (while statement), EmptyStatement class (null statement), BreakStatement class, and ContinueStatement class;

  Expressions include MethodInvocation class (method call), Assignment class (assignment expression) ("=", "+=", "-=", "*=", "/="), InfixExpression class (infix expression)) ("+", "-", "*", "/", "==", "!=", "<", "<=", ">=", "&&", "||".), PrefixExpression class (prefix expression) ("+" PLUS "-" MINUS "!" NOT), ParenthesizedExpression class (parenthesized expression), NumberLiteral class (integer), Name class (simple) MethodInvocation class (method call).

- **AST class** The key is to create a compilation unit node and create an instance of the class AST.

  AST ast = AST.newAST (JLS3);

- **ASTVisitor class** It provides the visit() method and the endVisit() method related to the node class, the preVisit() method and the postVisit() method which is independent of the node class.

  Boolean visit(T node): If this method returns true, then it will access the child node. If false is returned, child nodes are no longer accessed.

  Void endVisit(T node): This method is called after the child node's children have been visited or after visit(node) returns false.

  Void preVisit(): This method is called before visit(node).

  Void postVisit(): This method is called after endVisit(node).

  endVisit() is called after the node's children have been accessed.

  In the process of doing a simple parser, I mainly use the above visit() method when analyzing java code.

3.3 Eclipse with Java 8

Java is the world's most popular programming language. It is widely used in enterprise projects, game design, Android applications, etc and its IDE environment

is also highly concerned by developers. In this project, we choose java v8 as the programming language and Eclipse as the development platform. Eclipse is an open source, Java-based, extensible development platform. For its part, it is just a framework and a set of services for building a development environment through plug-in components. Fortunately, Eclipse comes with a standard set of plug-ins, including Java Development Tools (JDT). JDT is actually able to build Java program text into an abstract syntax tree (AST) that is a DOM-based structure. It contains a serial of necessary jar files that are needed during constructing the AST, these jars are listed in chapter 9.

For example, a parser that handles an arithmetic expression can convert a string of 1, 2 and 3 characters such as "1+2" into an object. This object is generated like a Java constructor call like new BinaryExpression(ADD, new Number(1), new Number(2)).

The reason for this conversion from a string to a data structure is because the compiler cannot directly manipulate strings such as "1+2". In fact, the nature of the code is not a string at all, it is a data structure with a complex topology, just like a circuit. The "1+2" string is just an "encoding" of this data structure, just as ZIP or JPEG encodes only the data they compress.

This encoding makes it easy for you to save the code to disk so that you can modify it with a text editor. However, you must know that the text is not the code itself. So after reading the text from the disk, you must "decode" it before you can manipulate the data structure of the code. For example, if the AST node generated by the Java code above is called node, you can use node.operator to access ADD and node.left to access node.right to access 2. This is very convenient.

The method of how to use AST can be detailly reflected in varieties of visitors in chapter 5 of the thesis.

3.4 Verigraph

Verigraph is a porting of the Berkeley verification tool [**panda2014verifying**] in the Java programming language. It defines almost all packet fields and corresponding functions and some basic constraints in order to model and verify the satisfiability of some FOL formulas. With this tool is possible to model complex networks with different kind of VNF interacting with each other. In particular they say that following the abstract syntax tree of the written code they can extract a path of condition that must hold in order to reach the send action, and from this path they derive the formulas. The process is not specified and it will be taken as starting point for this thesis. After they create a set of variables that permit testing some properties of the built network. These properties are:

∗ **Node reachability and isolation** Usually we want that two end-hosts can send

packets with each other, but sometimes in order to avoid the endless loop and keep two nodes isolated, we must keep the isolation satisfiability to be UNSAT.

* **Data reachability and isolation** The two end-hosts can send packets with each other, but the initial packet must be sent from only a specific host a. the host b can also sent a packet a, but that is only after the contact has been established between a and b. It states that a never accesses data originating from b(as HTTP protocol).

* **Content reachability and isolation** The content of an end-host can not be sent directly from the end-host, but it can cache its content in a proxy server, then these content can be accessed from the proxy server. The original host and the proxy server only need to keep the content consistent(as a CDN network).

* **Node traversal** To check that all the traffic from a to b passes through some middlebox m

Chapter 4

VNF Library

With this library the user can simply describe the functionality of the network function through the implementation of the defined methods and using the basic instructions of Java and the instructions offered.

4.1 Interface

The Interface class (Figure 4.1) models a logical interface on which the network function will receive or send a packet.

private Type interfaceType;

private final Integer id;

The integer 'id' locally identifies the logical interface. The 'Type' property (INTERNAL, EXTERNAL) represents the interface is internal or external for a network, which will enable the network function to understand how to process the received packets. For example, a VPN gateway separates a private local network and a public network and protects the packet(with an internal source IP address) from the private network to pass through the public network. The VPN gateway received the packet from an internal interface, this will drive a constraint on the packet that is going to be forwarded. In the opposite direction, when a VPN gateway received a packet with a public header from an external interface from a public network, the gateway will know it should remove the public header and then forward the packet into the corresponding internal private network. The method 'isInternal()' enable the parser understands and translates the condition into a Z3 formula (isInternal(p_0.IP_SRC)).

public String IP_ADRESS;

The IP_ADDRESS property represents an IP address (32bits). A network function can know if an interface belongs to itself or not by checking its value. It is used by routeTable in 'vRouter' function.

```
1 public class Interface {
2     public static final String INTERNAL_ATTR = "INTERNAL";
3     public static final String EXTERNAL_ATTR = "EXTERNAL";
4
5     public enum Type{INTERNAL, EXTERNAL};
6     public String IP_ADRESS;
7
8     private Type interfaceType;
9     private final Integer id;
10    private List<String> attributes;
11
12    public Interface(Integer id, Type type) {
13        this.id = id;
14        this.interfaceType = type;
15        this.attributes = new ArrayList<>();
16    }
17    public Integer getId() {
18        return id;
19    }
20    public List<String> getAttributes() {
21        return attributes;
22    }
23    public void addAttribute(String attribute) {
24        this.attributes.add(attribute);
25    }
26    public boolean isInternal(){
27        if(interfaceType == Type.INTERNAL)
28            return true;
29        return false;
30    }
31 }
```

Figure 4.1. Code of Interface.class

4.2 NetworkFunction

NetworkFunction (Figure 4.2) is the core class of the framework. It is an abstract class which developers need to extend in order to let the parser models the behaviour of their network function.

The main method of this class is

public abstract RoutingResult onReceivedPacket(Packet packet, Interface iface)

All middle-boxes needs to implement the abstract method. This method and the other method name 'defineSendingPacket()' are both set as the main methods that needed to be analyzed by the parser. Once the parser finds those two methods, it

```
1 public abstract class NetworkFunction {
2
3     protected List<Interface> interfaces;
4
5     protected Interface internalInterface;
6     protected Interface externalInterface;
7
8     public NetworkFunction(List<Interface> interfaces) {
9         this.interfaces = interfaces;
10
11        this.internalInterface = new Interface(0, Interface.Type.INTERNAL);
12        this.externalInterface = new Interface(1, Interface.Type.EXTERNAL);
13
14    }
15
16    public abstract RoutingResult onReceivedPacket(Packet packet, Interface iface);
17
18    public Interface getInternalInterface(){
19        return this.internalInterface;
20    }
21
22    public Interface getExternalInterface(){
23        return this.externalInterface;
24    }
25
26 }
```

Figure 4.2. Code of NetworkFunction.class

will continue to generate the rules according to the FORWARD action found in the VNF.

4.3 Packet

The Packet class (Figure 4.3 4.4) models an IP packet and contains all standard fields that our VNF will need to use in order to exchange data.

These packet fields correspond with that of the packet defined in the 'Verigraph-timeless' packet. The 'INNER_SRC' and the 'INNER_DEST' are used by the scenario where there are two IP headers of a packet, such as the VPN network and the IPv4-in-IPv6 network. The 'ENCRYPTED' field is a boolean variable, which represents if the content a packet is encrypted or not.

The HashMap stores the values of the different fields which are listed inside an enum structure. The 'equalsField()' method allows parser that will model it depending on the values that are checked inside this function.

```
1 public class Packet {
2
3     public enum PacketField {
4         IP_SRC,
5         IP_DST,
6         PORT_SRC,
7         PORT_DST,
8         PROTO,
9         ORIGIN,
10        ORIG_BODY,
11        BODY,
12        INNER_SRC,
13        INNER_DEST,
14        SEQUENCE,
15        EMAIL_FROM,
16        URL,
17        OPTIONS,
18        ENCRYPTED,
19    };
20    public static final String HTTP_REQUEST = "HTTP_REQ";
21    public static final String HTTP_RESPONSE = "HTTP_RESP";
22    public static final String POP3_REQUEST = "POP3_REQ";
23    public static final String POP3_RESPONSE = "POP3_RESP";
24    public static final String DNS_REQUEST = "DNS_REQ";
25    public static final String DNS_RESPONSE = "DNS_RESP";
26
27    private Map<PacketField, String> fields;
28    public Packet() {
29        this.fields = new HashMap<>();
30    }
31    public Packet(Packet p) {
32        this.fields = new HashMap<>(p.getAllFields());
33    }
34
```

Figure 4.3. First part of Packet.class

4.4 RoutingResult

The RoutingResult class (Figure 4.5) is a wrapper containing all the informations about the result of network function's core method.

The constructor takes as inputs an "Action", an Enumeration type which contains the decision the middle-box took such as DROP or FORWARD the packet. A "Packet" type, which contains the packet generated from the core function and the "Interface" on which the packet was sent. Thanks to this class the parser has all the information it needs to model the behavior of the network function.

```
35      public HashMap<PacketField, String> getAllFields(){
36          return (HashMap<PacketField, String>) this.fields;
37      }
38      public void setField(PacketField pField, String value){
39          this.fields.put(pField, value);
40      }
41      public String getField(PacketField pField){
42          return this.fields.get(pField);
43      }
44      public boolean equalsField(PacketField field, String value){
45          String temp = this.fields.get(field);
46          if(temp!=null){
47              if(value.compareTo(temp)==0)
48                  return true;
49          }
50          return false;
51      }
52      public boolean notEqualsField(PacketField field, String value){
53          String temp = this.fields.get(field);
54          if(temp!=null){
55              if(value.compareTo(temp)!=0)
56                  return true;
57          }
58          return false;
59      }
60      @Override
61      public Packet clone() throws CloneNotSupportedException {
62          Packet p = new Packet();
63          for(Entry<PacketField, String> entry : fields.entrySet())
64              p.setField(entry.getKey(), entry.getValue());
65          return p;
66      }
67  }
```

Figure 4.4. Second part of Packet.class

4.5 Table

The Table class (Figure 4.6 4.7) models the behaviour of a table containing all the information our network function needs to store.

The 'TableTypes' enumeration presents the type of the content that a table can store, including IP address, port number, protocol (used in both application layer and transportation layer), application data, URL and the generic data. If looking at the 'ClassGenerator.class' in chapter 5 explanation below, we will know that the generic data will not be analyzed in the translation process.

In order to retrieve an entry from the table it is possible to use 'matchEntry()' method by passing a chosen value it is possible to check if the entry is stored in

```
1 public class RoutingResult {
2
3     public enum Action { FORWARD, DROP, UNKNOW };
4
5     private Action action;
6     private Packet packet;
7     private Interface iface;
8
9     public RoutingResult(Action action, Packet packet, Interface iface) {
10        setAction(action);
11        setPacket(packet);
12        setIface(iface);
13    }
14
15    public Action getAction() {
16        return action;
17    }
18    public void setAction(Action action) {
19        this.action = action;
20    }
21    public Packet getPacket() {
22        return packet;
23    }
24    public void setPacket(Packet packet) {
25        this.packet = packet;
26    }
27    public Interface getIface() {
28        return iface;
29    }
30    public void setIface(Interface iface) {
31        this.iface = iface;
32    }
33 }
```

Figure 4.5. Code of RoutingResult.class

the table. According to the different principles of the middle-boxes, they will decide if DROP or FORWARD the received packet. If considering the web cache model (Figure 4.8):

public boolean setDataDriven();

By studying the way middle-boxes update their tables it was clear that it is possible to split them into two types. The ones which dynamically update their entries by checking the network traffic, "Data-Driven Tables", and the other ones which are configured statically and not by the incoming packets. NAT or Web-Cache are good examples of "Data-Driven Tables", the first adds an entry to the table when a new connection is established while the second one adds or updates the content of a cached web page when it receives an HTTP-Response from a server. On the other hand, an Acl-Firewall is a good example of "Non-Data-Driven Tables" because the

```
1 public class Table {
2     public static enum TableTypes{
3         Ip,
4         Port,
5         Proto,
6         BodyData,
7         URL,
8         Generic
9     }
10    protected List<TableEntry> entries;
11    protected int primaryFields;
12    protected int secondaryFields;
13    protected boolean dataDriven = false;
14    protected List<TableTypes> typeList = new ArrayList<>();
15
16    public Table(int primaryFields, int secondaryFields) {
17        entries = new ArrayList<>();
18        this.primaryFields = primaryFields;
19        this.secondaryFields = secondaryFields;
20
21    }
22    public void setTypes(TableTypes... types){
23        assert types.length == primaryFields + secondaryFields;
24
25        for(TableTypes type : types){
26            typeList.add(type);
27        }
28    }
29    public boolean storeEntry(TableEntry entry) {
30        return entries.add(entry);
31    }
32
```

Figure 4.6. First part of Table.class

entries are statically added and updated by the system's administrator. By default a table is modeled statically, the above method needs to be invoked when it is a dynamic one. In the next chapters, I will analyse in depth, with further examples, the way this method works and how the middle-box's model changes when is invoked or not.

4.6 TableEntry

This class models the behaviour of a table's entry.

Its constructor takes as input the entry's length. A "TableEntry" is stored inside

```
33    public boolean removeEntry(TableEntry entry){
34        return entries.remove(entry);
35    }
36    public TableEntry matchEntry(Object... fields) {
37        for(TableEntry entry : entries)
38        {
39            int i;
40            boolean flag = true;
41            for(i=0; i<fields.length && i<entry.size(); i++)
42                if(isValid(fields[i]) && !fields[i].equals(entry.getValue(i)))
43                    flag = false;
44
45            if(flag)
46                return entry;
47        }
48        return null;
49    }
50    protected boolean isValid(Object object) {
51        if(!(object instanceof String))
52            return false;
53        String s = (String) object;
54        if(s.equals(Verifier.ANY_VALUE))
55            return false;
56        else
57            return true;
58    }
59    public void clear() {
60        entries.clear();
61    }
62    public boolean setDataDriven() {
63        return dataDriven = true;
64    }
```

Figure 4.7. Second part of Table.class

```
1 TableEntry entry = cacheTable.matchEntry(packet.getField(PacketField.URL), Verifier.ANY_VALUE);
2    if(entry != null)
3    { // reply with a new packet to the web client
4        Packet p = packet.clone();
5        p.setField(PacketField.URL, (String)entry.getValue(0));
6        //...
7    return new RoutingResult(Action.FORWARD, p, internalInterface);
8    }
9    Else{
10       // drop the packet
11   }
```

Figure 4.8. Usage of matchEntry method

a “Table” and contains all the information a developer needs to gather in order to model the middle-box’s functionality. For example, if we consider a Web-Cache the

```
1 public class TableEntry {
2
3     protected List<Object> values;
4
5     public TableEntry(int length) {
6         this.values = new ArrayList<Object>(length);
7         for(int i=0; i<length; i++)
8             values.add(new Object());
9     }
10
11
12    public Object getValue(int index){
13        if(index < 0 || index >= values.size())
14            return null;
15        return values.get(index);
16    }
17
18    public void setValue(int index, Object value) {
19        values.set(index, value);
20    }
21
22    public int size() {
23        return values.size();
24    }
25
26 }
```

Figure 4.9. Code of tableEntry.class

entry will contain a web page while if we consider a NAT it will contain the Ip and Port source/destination. Its usage is shown in (Figure 4.8).

Chapter 5

Translation and Verification Process

Ideas:

1. Create more NFs(router, CDN, firewall...) to extend a fundamental library (packet, table..)to be used for modeling VNFs;

2. A general parser parses Java code via AST, saves/fetch information into/from XML file by using marshal/unmarshal method in JAXB Architecture that enables cross-platform;

3. Translate network scenarios into FOL formulas that are then analyzed by Z3 prover (Verigraph) to verify some basic network properties(Isolation, reachability in packet and content levels);

4. Verification in Verigraph.

5.1 it/polito/parser

In order to translate Java-like virtual network functions into model representation in the form of Z3 boolean expressions, Throughout the development process, an Abstract Syntax Tree (AST) is used as an intermediate representation of the program, so first we must learn to establish the AST corresponding to the source code and access the AST. First, we need to understand how to convert Java source code to AST, that is, parse the source code. Eclipse AST provides ASTParser class for parsing source code, ASTParser has two ways to import source code, one is in the form of Java Model and the other is in the form of a character array.

1. **Parser.class**

· **public Parser(String fileName)**

This class is the starting point of operation of this project, the Parser constructor receives the full path of the file containing the VNF source code to be parsed, fileName is an absolute local path of the VNF source code.

· **parse()**

This method parses the fileName source file provided in the constructor and generates First Order Logic (FOL) formulas as output. The ASTParser is created and used as follows: CompilationUnit represents a Java source (.java) file. Once the AST has been created, the parser will then create one class context in order to store all the data that are parsed from the VNF source code, these data will be needed during generating the models in the form of Z3 representation. In this case, one class visitor is created to visit the whole AST tree. During the process, a serial of different context classes will be generated to store different VNF functionalities.For Example:

MethodContext: keep track of the general pieces of information of the method like the method name, class context, and method variables. Stores all the snapshots taken during the parsing phase. Of course, these data are analyzed by methodVisitor. Moreover, the method name is used by the parser to specify which method in the class will be referred to generate the FOL rules.

StatementContext: stores all special information about the received packet and the packet that is going to be forwarded, including all if, else-if, else and setField statements. These data are analyzed by statementVisitor.

TableEntryContext: It is used when the VNF model is data-driven. It stores information of table entry (what is the value at each column). These table entry objects are created by a expression visitor and used in RuleContext class in modeling phase.

2. **ClassVisitor.class**

Main class performing the Stage 1 parsing phase, it specifies the parser will parse only the main methods (with a standard name) onReceivedPacket() and defineSendingPacket() when visiting MethodDeclaration ASTNode.

Main tasks of this phase:

∗ scan the variables in the source code, use visit(FieldDeclaration node) method to parse all the variables declared inside the class but outside any method every time a member variable declaration is found inside the class source code.

```
@Override
    public boolean visit(MethodDeclaration node) {
        ...
        String methodName = node.getName().toString();
        if(!methodName.equals(Constants.MAIN_NF_METHOD) && !methodName.equals(classContext.getClassName())
            && !methodName.equals(Constants.DEFINE_SENDING_PACKET_METHOD))  // Check the method name
            return true;
        MethodContext methodContext = new MethodContext(methodName, classContext);
        classContext.addMethodContext(methodContext);
        ...
        StatementVisitor myStmtVisitor = null;
        StatementContext statementContext = new StatementContext(methodContext);

        @SuppressWarnings("unchecked")
            /* We parse all the statements, one after the other */
        List<Statement> stmts = (List<Statement>)node.getBody().statements();
        for(Statement s : stmts)
        {
            myStmtVisitor = new StatementVisitor(statementContext);
            s.accept(myStmtVisitor);
        }
}
```

- find matchEntry() method calls and parse the involved fields (which fields are being matched), these fields will drive some constraints on the sending and received packet.
- find storeEntry() method calls and parse the involved fields (which fields are inserted in the table), during the parsing, one table entry is created for each entry setter method. StoreEntry() method is used when the network function is data-driven.

3. **ExpressionVisitor.class**

```
1 public class ExpressionVisitor extends ASTVisitor {
2
3     private List<MyExpression> predicates;
4     private int nestingLevel;
5     private StatementContext context;
6
```

This class visits all the expression presents in the method. It follows a variety of expression statements outside all methods and in the main parsed methods specified by the class visitor in the VNF source code. For example: setDataDriven(), setTypes(), setField() and setValue().

· **this.natTable.setDataDriven();**

If the expression visitor finds that the method name is setDataDriven (Constants.DATA_DRIVEN), the variable isDataDriven will set to true in the class context object, which means the table of network function is data-driven, the table data depends on the previously received packets.

```
public boolean visit(MethodInvocation node) {
        StringBuilder builder = new StringBuilder();
        switch(node.getName().toString())
        {
            case Constants.DATA_DRIVEN:
                context.getMethodContext().getContext().setDataDriven(true);
                break;
            case Constants.SET_TYPES:
                Context ctx = context.getMethodContext().getContext();
                        //...a new visitor to visit MethodInvocation node.
                ctx.tableTypes.add(builder.toString());
            case Constants.SET_FIELD_METHOD:
                        //...get sending packet name : pName
                Expression field = (Expression) node.arguments().get(0);
                Expression value = (Expression) node.arguments().get(1);
                MyExpression expression = new MyExpression(field, value, nestingLevel);
                expression.setPacketName(pName);
                predicates.add(expression);
                break;
            case Constants.ENTRY_SETTER:
                Expression fieldEntry = (Expression) node.arguments().get(0);
                Expression valueEntry = (Expression) node.arguments().get(1);
                        //...parse fieldEntry to get position = Integer.parseInt(builder.toString());
                position = (//...a Integer)
                String EntryValue = (//...a packet field or self define string)
                MyExpression expressionEntry = new MyExpression(fieldEntry, valueEntry, -1);

                TableEntryContext entry = new TableEntryContext(context.getConditions(), builder.toString(), position);
                entry.setExpression(expressionEntry);
                context.getMethodContext().addEntryValues(entry);
                break;
```

· **this.aclTable.setTypes(Table.TableTypes.Ip,Table.TableTypes.Ip);**

In order to parse 'setType'(Constants.SET_TYPES) method of table class, a new visitor will be created to analyze and record the type of data content stored in different table columns. These parsed table types will be stores in 'tableTypes' global variable of class context. The types of data content are shown in table types in table class: Ip, Port, Proto, BodyData, Generic.

· **p.setField(PacketField.IP_DST, packet.getField(PacketField.IP_SRC));**

If the method name is 'setField'(Constants.SET_FIELD_METHOD), the different field values of exiting packet will set up one by one. So for each setField method, a instant of MyExpression class will be created which stores the packet field with corresponding value expression, it will be used for generating a constraint on exiting packet.

· **cacheEntry.setValue(0, packet.getField(PacketField.URL));**

If the method name is 'setValue' (Constants.ENTRY_SETTER), expression visitor can get the information about the values in different positions of a VNF table. During the process, table entry context object is instantiated to store the information. Of course, these objects belong to the corresponding method context of the specified main method by a class visitor. Usually, this method means the network function is data-driven.

```java
public class ReturnStatementExplorator extends ASTVisitor {

    private List<ReturnStatement> returnList;
    private List<Action> actionList;
    private boolean foundReturn;

    public ReturnStatementExplorator(){
        this.foundReturn = false;
        this.returnList = new ArrayList<>();
        this.actionList = new ArrayList<>();
    }
    @Override
    public boolean visit(ReturnStatement node) {

        ReturnStatementVisitor visitor = new ReturnStatementVisitor();

        node.getExpression().accept(visitor);
        actionList.add(visitor.getAction());

        this.foundReturn = true;
        returnList.add(node);
        return true;
    }
```

Figure 5.1. ReturnStatementExplorator.class

4. **ReturnStatementExplorator.class**

   In the code above(Figure 5.1) is possible to notice how the parser looks for actions performed on the received packet (FORWARD or Drop). Once a return statement explorator finds a return statement, it will create a new return statement visitor to check the result, store it in own actionList variable, set variable 'foundReturn' to be true. This information is needed in 'IfElseBranch' class in order to display the parsing process.

5. **ReturnStatementVisitor.class**

```java
public class ReturnStatementVisitor extends ASTVisitor {
    private Action action;
    private String packetName;
    private String interfaceName;
```

Figure 5.2. ReturnStatementVisitor.class

This return statement visitor will also keep the information of the action, packet name and interface name of the forwarding packet. This process is very important because these data will be delivered to 'returnSnapshot' object in statement visitor class when creating the return snapshot, that will directly generate the rules of a model. The process is shown in (Figure 5.3) below.

6. **StatementVisitor.class**

```
1  public class StatementVisitor extends ASTVisitor {
2      private List<IfElseBranch> conditions;
3      private List<IfElseBranch> previousIfElseBranch;
4      private List<ForLoop> loops;
5      private List<MyExpression> predicatesOnSentPacket;
6      private boolean foundReturn;
7      private boolean isDeadCode;
8      private boolean foundForward;
9      private int nestingLevel;
10     private int skippedActions;
11     private CompilationUnit compilationUnit;
12     private Map<String, List<Variable>> variables;
13     private StatementContext statementContext;
14     private String simpleName;
15     private String variableTypeName;
16     private boolean isGlobal;
```

Figure 5.3. StatementVisitor.class

Statement visitor is firstly called by a class visitor when the class visitor finds the main method needed to be parsed, then statement visitor will recursively call itself when meeting 'ifStatement' component in java source code.

· **simpleName and isGlobal variable**

used when statement visitor finds an assignment node in parsing phase, 'simpleName' records the name of a local and global variable, and 'isGlobal' indicates if the variable is global or not. If it is global and its variable type name is equal to 'Table', then statement visitor will get the table size of the network function.

· **public boolean visit(ReturnStatement node)** (Figure 5.4)

If the visitor finds return statement, it calls a return statement visitor to check the action performed on the received packet. If forwarding the packet, the visitor will create a new return snapshot and add it to the corresponding

```
public boolean visit(ReturnStatement node) {
        this.foundReturn = true;
        ReturnStatementVisitor r = new ReturnStatementVisitor();
        node.accept(r);
        if(r.getAction() == Action.DROP)
        {
            skippedActions++;
            return false;
        }
        if(r.getAction() == Action.FORWARD)
        {
            ....
            //create a new return snapshot and add it to method context.
            if(!statementContext.getMethodContext().getMethodName().equals(Constants.DEFINE_SENDING_PACKET_METHOD))
                statementContext.getMethodContext().addReturnSnapshots(statementContext.createSnapshot(...);
            else{       // the main method name is 'onReceivedPacket'
                statementContext.getMethodContext().addReturnSnapshots(statementContext.createSnapshot(...);
            }
        }
        ....
        return false;
    }
```

Figure 5.4. visit(ReturnStatement node)

method context of the statement context belongs to. From the code above, we can know that the number of return snapshot (also the number of the rules for the model) is equal to the number of the FORWARD action.

'loops' variable records all the FOR-EACH statement. 'variables' is a HashMap instant and records all local variables found in the main method. These data are significant, they are analyzed in 'RuleContext' class to generate the rules. 'conditions' and 'previousIfElseBranch' are used to store the information about the analyzing process performed on the received packet in different parsing level of 'ifStatement' node, they point out the constraints on 'p_1' *inthegeneratedrules*.

7. **MyExpression.class**

```
public class MyExpression {
    private String packetName;
    private Expression field;
    private Expression value;
    private int nestingLevel;
}
```

The 'field' global variable stores the name of packet field, and the 'value' variable stores the value data of the packet field. The 'nestingLevel' variable indicates its object is generated from the which level of 'ifStatement' node.

Its object is initiated by expression visitor in two cases. The first case is when the visitor finds the 'setField()' method invocation node, the initiated object stores

the data of the packet that is going to be forwarded outside. The second case is when the visitor finds the 'setValue()' method invocation node, the initiated object belongs to the corresponding table entry context which stores the table data of the network function, these data comes from the previously received packet, because the 'setValue()' method is used when the function is data-driven.

5.2 it/polito/parser/context

1. **ReturnSnapshot.class** As we said above, the object of return snapshot is

```
1 public class ReturnSnapshot {
2     private MethodContext methodContext;
3     private List<IfElseBranch> conditions;
4     private List<IfElseBranch> previousConditions;
5     private List<MyExpression> returnPredicates;
6     private String packetName;
7     private String interfaceName;
8     private int nestingLevel;
9     private boolean initialPacket;
```

Figure 5.5. ReturnSnapshot.class

generated by statement visitor when it finds FORWARD action in 'Return-Statement' node (Figure 5.4), all values of its properties are from the 'method-Context' variable. All the return snapshots will be analyzed by a rule generator. Each snapshot belongs to a specific method context (see createSnapshot() method in Statement.class)

2. **Context.class** It is initiated when parser starts to parse the java source code in Parser.class, a class visitor for each java code stores all information in a context object. It uses the 'classVariables' property to record all global variables in the network function. A context can have several method contexts, the number depends on the number of the main method (onReceivedPacket() and defineSendingPacket()) that a class visitor wants to parse. A network function is set to be not data-driven by default, unless that the table of the VNF explicitly calls the 'setDataDriven()'method.

   After the whole java code is parsed, a rule generator will use the data in the context to generate the rules. The context fetches out all method contexts with the expected main methods name. For each of them, there are several return snapshots that contain the detailed constraints on the forwarding packet, the

```
1 public class Context {
2     private String className;
3     private Map<String, List<Variable>> classVariables;
4     private Map<String, MethodContext> methodContexts;
5     private CompilationUnit compilationUnit;
6     private boolean isDataDriven = false;
7     public int tableSize = 0;
8     public List<String> tableTypes = new ArrayList<>();
9 }
```

Figure 5.6. Context.class

rule generator will generate a rule for each of the return snapshots. The process is the Figure 5.7. . . line 7-19

```
1  Context classContext = new Context(compilationUnit);
2          /* Perform STAGE1 parsing phase */
3          ClassVisitor v1 = new ClassVisitor(classContext);
4          compilationUnit.accept(v1);
5          RuleGenerator ruler = new RuleGenerator(classContext.getClassName(),true);
6                      // specify that analyze "onReceivedPacket()" method
7          MethodContext methodContext = classContext.getMethodContext(Constants.MAIN_NF_METHOD);
8          if(methodContext!=null){
9              for(ReturnSnapshot returnSnapshot : methodContext.getReturnSnapshots()){
10                 ruler.setSnapshot(returnSnapshot);
11                 ruler.generateRule();
12             }
13         }
14                 // specify that analyze "deifneSendingPacket()" method
15         methodContext = classContext.getMethodContext(Constants.DEFINE_SENDING_PACKET_METHOD);
16         if(methodContext!=null){
17             for(ReturnSnapshot returnSnapshot : methodContext.getReturnSnapshots()){
18                 ruler.setSnapshot(returnSnapshot);
19                 ruler.generateRule();
20             }
21         }
22         ruler.saveRule();
23         //......
24     }
```

Figure 5.7. Code for generating rules from each return snapshot

3. **MethodContext.class** The figure 5.8 above tells us that almost all data stored by a new method context object is empty, it is filled during parsing phase. The 'methodVariables' property is filled by a class visitor when the visitor finds the declarations of local variables. The 'returnSnapshot' property is filled by a statement visitor in figure 5.4. The 'entryValues' property is filled by an expression visitor when it finds a MethodInvocation node with method

```
public MethodContext(String methodeName, Context context) {
        this.methodName = methodeName;
        this.context = context;
        this.methodVariables = new HashMap<>();
        this.returnSnapshots = new ArrayList<>();
        this.entryValues = new ArrayList<>();
}
```

Figure 5.8. MethodContext.class

name equal to string 'setValue', the code is shown in figure 3. The 'context' property means this method context belongs to a specific context of the VNF in class level.

4. **StatementContext.class** The first new statement context is created by a

```
public StatementContext(MethodContext methodContext) {
        assert methodContext != null;
        this.conditions = new ArrayList<>();
        this.previousConditions = new ArrayList<>();
        this.forLoops = new ArrayList<>();
        this.returnPredicates = new ArrayList<>();
        this.nestingLevel = 0;
        this.skippedActions = 0;
        this.isDeadCode = false;
        this.methodContext = methodContext;
        this.compilationUnit = methodContext.getContext().getCompilationUnit();
    }
```

Figure 5.9. StatementContext.class

class visitor when the visitor finds the 'onReceivedPacket()' method, all data that it stores are empty. Then it will be called and filled by all statement visitors in order to parse all statements in the main method. Its all properties are delivered to that of the statement visitor. The class has a significant method called 'createSnapshot() ', which provide an interface for a statement visitor to add a new snapshot in the method context.

5.3 it/polito/rule/generator

JAXB is an acronym for Java Architecture for XML Binding. It allows us to use JAXB annotations to serialize Java objects to XML files and vice versa. Generally speaking, serialization is used for communication. For example, the server serializes

the data and sends it to the client. The client deserializes the received data and then manipulates the data. After the completion, the serialization is sent to the server. The server then retransmits the data. To put it plainly, data needs to be serialized before it can be transmitted between the server and the client in different platforms. Of course, The concepts of the server and the client are broad and can be communicated on the network, in different processes on the same machine, and even in the same process.

But why serialization was needed? The data can be transmitted without serialization, but it cannot be cross-platform and security cannot be guaranteed. But If the data is transmitted through a specific protocol after serialization, the presentation layer sends a specific data format to the service layer through the proxy or channel. This data is serialized, such as XML, and the server must perform reverse sequence after receiving it. In this way, the client can use a completely different development platform than the server, as long as it can deserialize the XML data, and XML is an industry standard data format that is supported by all basic platforms. This also applies to interprocess communication.

Our work leverages recent advances in JAXB by using marshaling (Converting Java objects to XML files.) and unmarshalling (converting XML content to Java objects). Using JAXB is simple. Just annotate the object with JAXB annotations, then use jaxbMarshaller.marshal() or jaxbMarshaller.unmarshal() to do the XML/Object conversion.

Based on its advantage, we defined a schema called “LogicalExpression.xsd” under the folder “./xsd” of the thesis project. The root element is called “ExpressionResult”, which contains all data and constraints parsed from the middle-box by all previous visitors. Moreover, each “LogicalExpressionResult” element may contain multiple “ExpressionObject” elements, which means this middle-box has multiple rules (Figure 5.10).

```
1  <complexType name="ExpressionResult">
NFDev/CV
2      <sequence>
3          <element name="TableSize" type="int"></element>
4          <element name="DataDriven" type="boolean"></element>
5          <element name="RecordPreviousPacket" type="boolean"></element>
6          <element name="TableFields" type="string" minOccurs="0" maxOccurs="unbounded"></element>
7          <choice maxOccurs="unbounded">
8              <element name="Node" type="tns:LU_Node"></element>
9              <element name="Packet" type="tns:LU_Packet"></element>
10         </choice>
11         <element name="LogicalExpressionResult" maxOccurs="unbounded" type="tns:ExpressionObject"></element>
12     </sequence>
13 </complexType>
```

Figure 5.10. Root element of schema in LogicalExpression

1. **RuleGenerator.class**

   A rule generator receives different “returnSnapshot” objects as a parameter by calling the “setSnapshot()” method to start to construct the rules. The

```
public RuleGenerator(String name, boolean verbose){
        this.factory = new ObjectFactory();
        this.units = new HashMap<String,LogicalUnit>();
        this.expressions = new ArrayList<>();
        this.verbose = verbose;

        fileNameXml = "./xsd/Rule_"+name+".xml" ;
        fileNameTxt = "./xsd/txt/Rule_"+name+".txt";
    }
```

Figure 5.11. ruleGen.class

main process of construction is actually performed by a “RuleContext” class. It creates a condition visitor to analyze all “IfElseBranch” objects from the “conditions” and “previousConditions” properties. The data of “returnPredicates” property whose data comes from an expression visitor, will drive the constraints on different fields of the received packet (p_1) and the forwarding packet (p_0) by means of a “ReturnExpressionVisitor ” class. For the remaining fields of the forwarding packet, their values are set to always equal to that of the received packet (p_1) by calling to the “setExitPacketConditions()” method of the “RuleContext” object.

· **public void generateRule()**

A rule generator specifies the main tasks by "generateRule()" method to start analyze a return snapshot, see the following code (Figure 5.12).

2. **RuleContext.class**

The “RuleContext” class contains all methods that are used to construct the “result” variable with the “ExpressionObject” type from the past “returnSnapshot” argument coming from the “RuleGenerator” class. Each “returnSnapshot” parameter will be converted to a rule. The “unit” property contains all logical units defined in the schema “./xsd/LogicalExpression”. The variables “packetCounter”, “nodeCounter”, “valueCounter” are used to counter the number of packets that are needed in the rules. The main methods used are in the following code:

· **public void setDefaultRule(String methodName)**

Construct the basic rule for the ‘send()’ method on the packet (p_0) in the antecedent part of the logical operator “Implies” and the ‘exist()’ method in the consequent part of the logical operator “Implies” on the packet (p_1).

```
public void generateRule(){
    ConditionVisitor visitor = new ConditionVisitor(ruleContext);
        for(IfElseBranch branch : returnSnapshot.getConditions()){
            //....
            branch.getStatement().accept(visitor);
            ExpressionObject temp =visitor.getExpression();
            if(temp!=null){
                //....
                    ruleContext.setLastExpression(temp);
            }
            visitor.clean();
        }
        visitor.clean();
        //....call to returnSnapshot's 'getPreviousConditions()' method to analyze one by one like above code
        if(verbose){
            if(!returnSnapshot.getReturnPredicates().isEmpty()){
                    for(MyExpression expression : returnSnapshot.getReturnPredicates()){
                        //....
                        expression.getValue().accept(new ReturnExpressionVisitor(ruleContext, field));
                        fields.add(field);
                        //....
                    }
                }
                ruleContext.setExitPacketConditions(fields);
        }
}
```

Figure 5.12. generateRule() method of RuleGenerator.class

```
public class RuleContext {
    private ObjectFactory factory;
    private ExpressionObject result;
    private List<LogicalUnit> units;
    private List<TableEntryContext> entryList;
    private ReturnSnapshot returnSnapshot;
    private LogicalOperator entryPoint_p1;
    private LogicalOperator entryPoint_p2;
    private Map<String,List<Variable>> localVariable;
    private Map<String,List<Variable>> globalVariable;
    public boolean isDataDriven;
    private String netFunction;
    private int packetCounter = -1;
    private int nodeCounter = -1;
    private int valueCounter = -1;
    public RuleContext(ObjectFactory factory, ReturnSnapshot returnSnapshot){
    //....
    }
}
```

Figure 5.13. RuleContext.class

This part is always fixed for each of the rules. The “entryPoint_p1” is set to be equal to a logical operator ‘AND’, it points to the “expression” property of the “exist” method. The code is as follows (Figure 5.14):

· **public void setExitPacketConditions(List<String> removeField)**

```
1 ( send(n_Router,n_0,p_0,t_0) ==>
2     E(n_1, p_1, t_1 |
3        ( recv(n_1,n_Router,p_1,t_1) &&....)))
4
```

Figure 5.14. entryPoint_p1 (AND) logical operator

It is called by a "RuleGenerator" class when it analyses the 'ReturnPredicates' of a return snapshot. All values of the packet(p_0) fields except the fields inside 'removeField' parameter (involved in "ReturnPredicates" property) will be set to be equal to that of the received packet(p_1) by default.

· **public LFIsInternal isInternalRule(String packetName, String packetField)**

This will generate the rule *!(isInternal(p_0.IP_DST)* in Z3 presentation in the format of FOL.

· **boolean setLastExpression(ExpressionObject expression)**

The argument 'expression' presents a constraint on field of the forwarding packet(p_0) and the receive packet(p_1), it will be added as a operand of the 'entryPoint_p1' operator. For Example, if the 'expression' argument is the constraints (isInternal(p_1.IP_SRC) && (p_0.IP_SRC == p_1.IP_DEST)), then the rule in (Figure 5.14) will be the code in (Figure 5.15):

```
1 ( send(n_Router,n_0,p_0,t_0) ==>
2    E(n_1, p_1, t_1 |
3       ( recv(n_1,n_Router,p_1,t_1) && isInternal(p_1.IP_SRC) && (p_0.IP_SRC == p_1.IP_DEST)....)))
4
```

Figure 5.15. Insertion in entryPoint_p1 (AND) logical operator

· **public ExpressionObject generateRuleForVariable(String variableName, Oper**

The rule context class will check the name of the variable type (if it is 'tableEntry' type) from its local variables and the global variables. Once found the type name is *(Constants.TABLE_ENTRY_TYPE)*, then it starts to construct the rules by considering if the middle-box is data-driven or not. The variable comes from the IF condition *(if (entry != null))*. If data-driven, it constructs the second logical operator 'entryPoint_p2' (AND). it means the table data

```
(( send(n_WebCache,n_0,p_0,t_0) && isInternal(p_0.IP_DST) ==>
    E(n_1, p_1, t_1 |
       ( recv(n_1,n_WebCache,p_1,t_1)&& isInternal(p_1.IP_SRC) && (p_1.PROTO == HTTP_REQUEST)
       && E(n_2, p_2, t_2 |
           (recv(n_2,n_WebCache,p_2,t_2)&& !(isInternal(p_2.IP_SRC)) && (p_1.URL == p_2.URL)))
           && (p_0.IP_DST == p_1.IP_SRC) && ...)))
```

of the middle-box comes from the previously received packets, so when the middle-box received a packet, it must try to match some packet fields with previous packets, if there is one table entry is matched, then the packet is allowed to be processed more. So our parser will construct a new packet (p_2) and a new middle-box node (n_2) to express the previous packets and its source address. This previous existing packet (p_2) will become a constraint that is parallel with the existing received packet (p_1), then the new "Exist" expression object will be put inside the logical operator (entryPoint_p1), and the 'entryPoint_p2' is set up to the expression property of this "Exist" constrint. The matched packets fields between packet p_1 and p_2 depend on the matched field names and the 'tabelEntryContext' properties of the rule context class and their drived constrints will be part of the 'AND' (entryPoint_p2) operator. The code is in (Figure 5.16).

If non-data-driven, it means the table of the middle-box is static, its data is inserted manually. In this case, the parser will construct the 'matchEntry' expression object and put it inside 'the entryPoint_p1' logical operator. The matched packet fields depend on the matched field names.

· **public ExpressionObject generateRuleForMethod(String variableName,Meth**

```
(( send(n_IPv4Exit,n_0,p_0) && (p_0.INNER_SRC == null)) ==>
    E(n_1, p_1 |
       ( recv(n_1,n_IPv4Exit,p_1) && (p_1.IP_SRC == accessIp) && (p_1.IP_DST == exitIp)
       && !((p_1.INNER_SRC == null)) && (p_1.ENCRYPTED == true) &&...)))
```

Figure 5.17. Rule generated by 'generateRuleForMethod()'

This method is called by a condition visitor (look at the 'ConditionVisitor.class' explanation). If the method name is 'isInternal', then the source address of the received packet(p_1(if non-data-driven) or p_2(if data-driven)) will an internal address from the internal network for the middle-box. It will drive the constraint *isInternal(p_2.IP_SRC)* or *isInternal(p_1.IP_SRC)*, or the constraint is their negation when the interface is not internal.

If the method name is 'equalsField', then the parser can construct the constraints on the field of the newly received packet (p_1), its value may be a fixed application protocol *(Packet.POP3_REQUEST)*, a string('true' or 'false' for the packet field 'ENCRYPTED', 'null' for the packet field 'INNER_SRC', or fixed parameter 'exitIp'...). The rule is in (Figure 5.17)

· **private boolean setExpressionForPacket(ExpressionObject expression, String**

This method receives a packet name as the second argument, usually, its value is 'p_1' or 'p_2'. This is called when a rule context call to "generateRuleForExitingPacket()" with a 'MethodInvocation' node as the second argument. If the method name is the string 'getField' (Constants.GET_FIELD_METHOD), it means one field of the forwarding packet(p_0) is equal to that of the received packet, then this 'expression' is inserted into the 'entryPoint_p1' operator, in this case, the packet name is set to 'p_1', the existing packet means the packet(p_0) that is going to be forwarding; if the method name is the string 'getValue' (Constants.ENTRY_GETTER), it means the middle-box is data-driven, then there is a relationship between the packet that is going to be forwarded(p_0) and the previously received packet(p_2). Then they constraints driven from the 'MethodInvocation' node will be inserted into the 'entryPoint_p2' operator, in this case, the packet name is set to 'p_2'.

3. **ConditionVisitor.class**

   A "ConditionVisitor" class starts analyzing a condition by checking which is the primary logical operator, it can be a CONDITIONAL AND, a CONDITIONAL OR, an EQUALS or a NOT EQUALS. It recursively analyzes the different operands and creating a logical tree with simple conditions as terminating nodes, once it reaches an operand, it passes all the information about the condition to the "RuleContext" class by invoking the appropriate method. If the leaf condition is a known variable or method it generates an appropriate rule and recursively adds it to the final rule otherwise it is skipped. It is able to analyze the conditions on an interface, "equalsField()" method, "entry" variable. The possible conditions are in the following code (Figure 5.18):

   · **isInternal() or equalsField()**

   When a condition visitor finds a 'MethodInvocation' ASTNode component from the 'IF-ELSE' condition in the java source code, the method name may be "isInternal" *(if (entry != null)* or "equalsField" *(packet.equalsField(PacketField.PROTO, Packet.HTTP_REQUEST))*, the visitor will call to the "generateRuleForMethod" method of the 'RuleContext' class to analyze the two different methods in order to construct the different constraints on the packet fields.

```
1 if (iface.isInternal())
2 {
3     if (packet.equalsField(PacketField.INNER_SRC,String.valueOf(null)
4         && !packet.equalsField(PacketField.ENCRYPTED, String.valueOf(true)))
5     {
6         TableEntry entry = dnsTable.matchEntry(packet.getField(PacketField.URL), Verifier.ANY_VALUE);
7         if (entry != null) {
8         //....
9         }
10    //....
```

Figure 5.18. Conditions on interface, method and variable

· **entry!=null** If a condition visitor finds a 'SimpleName' ASTNode component from the 'IF-ELSE' condition *(if (entry != null))* in the java source code, it will pass the information to the method 'generateRuleForVariable()' of the 'RuleContext' class to analyze the variable in different use cases (data-driven or non data-driven).

5.4 it/polito/rule/unmarshaller

As we have already said before, to guarantee cross-platform and security, we can serialize and store the java object in XML format, so that a server is able to unmarshal the rule into its own suitable data format after receiving it. In my thesis, the 'ruleUnmarshaller' and 'ClassGenerator' classes will do work this work together and translate the XML files into appropriate Verigraph classes in the form of FOL to be verified by the Z3 prover. They are called by the Parser and takes all the information inside the 'ExpressionResult' object and creates a new file called Rule_VNFname.java that supply the input to Z3. inside the package mcnet.netobjs.NF of the project verigraph-timeless.

1. **RuleUnmarshaller.class** A rule unmarshaller has two main tasks. The first one is to unmarshall each XML file in folder './xsd' generated by a 'RuleGenerator' class to generate the 'ExpressionResult' root element. The second one is to generate the method called install_VNFname(). It may need several arguments that need to be configured manually in the test file. The rule unmarshaller add different constraints on in the z3 solver. The number of the rules depends on the number of 'FORWARD' actions found from the VNF source file. It firstly declares the logical units(p_0, p_1, n_0, n_1...) according to the 'logicalUnits' unmarshalled from the XML file, and secondly generates the FOL logical expression, see the code in (Figure 5.19).

2. **ClassGenerator.class** A classGenerator specifies the paths of the XML files and the Z3 java files, calls to a rule unmarshaller to get the names of table types and the table size. Their values are set when creating the table in the

```
public void installAAA(){
    Expr n_0 = ctx.mkConst("n_AAA_" + n_AAA + "_n_0", nctx.node);
    Expr n_1 = ctx.mkConst("n_AAA_" + n_AAA + "_n_1", nctx.node);
    Expr p_0 = ctx.mkConst("n_AAA_" + n_AAA + "_p_0", nctx.packet);
    Expr p_1 = ctx.mkConst("n_AAA_" + n_AAA + "_p_1", nctx.packet);
constraints.add(ctx.mkForall(new Expr[] { p_0, n_0 },
            ctx.mkImplies((BoolExpr) nctx.send.apply(n_AAA, n_0, p_0),
                ctx.mkExists(
                    new Expr[] { p_1, n_1 }, ctx.mkAnd(
                        (BoolExpr) nctx.recv.apply(n_1, n_AAA, p_1),
                        ...),
                    1, null, null, null, null)),
            1, null, null, null, null));
```

```
public void startGeneration() {
        try {
            u = new RuleUnmarshaller(fileNameXml, originalName, ast);
            tableSize = u.getReuslt().getTableSize();
            isDataDriven = u.getReuslt().isDataDriven();
            tableTypes = u.getReuslt().getTableFields();
            isRecordPreviousPacket = u.getReuslt().isRecordPreviousPacket();
            //...
            List<String> tempList = new ArrayList<>();
            for(String type : tableTypes){
                if(type.compareTo(Constants.ENUM_GENERIC)==0)
                    tempList.add(type);
            }
            tableTypes.removeAll(tempList);
            tableSize = tableTypes.size();
            //...
            }
            TypeDeclaration td = ast.newTypeDeclaration();
            //...
            fieldDeclaration(td);
            constructorDeclaration(td);
            initDeclaration(td);
            getZ3NodeDeclaration(td);
            addConstraintsDeclaration(td);
            setInternalAddressDeclaration(td);
            if (!isDataDriven && tableSize > 0) {
                addEntryInit(td);
            }
            td.bodyDeclarations().add(u.generateRule());
```

Figure 5.20. ClassGenerator.class

Java source code. Due to the ‘tableTypes’ in the ‘Table’ class includes the ‘Generic’ value and actually, the content inside the table is not used, so the class generator will remove the ‘Generic’ table types parsed from an XML file. Of course, the table size will reduce by one (Figure 5.20 line 3 14).

After that, class generator will call to its own methods to generate the field declaration part, ‘Rule_VNFname()’, ‘init()’, ‘getZ3Node()’, ‘addConstraints()’, ‘setInternalAddress()’, ‘addEntry()’, ‘install_VNFname()’ methods (Figure 5.20line 19 28).

Chapter 6

VNF Models

6.1 Router/CPE

CPE(Customer-premises equipment) Wireless CPE is a wireless terminal to receive WiFi signal access device, is any terminal and associated equipment located at a user's premises to receive WiFi signal. The CPE simply said that it is a transponder, it receives wifi signal in the distance, and then spreads out via a local area network (LAN) on the client side. CPE can be widely used in rural areas, cities, factories, residential and other wireless network access, and can save the cost of laying a wired network. CPE generally refers to devices such as telephones, routers, network switches and so on. in our model, Router and CPE are same.

```
( send(n_Router,n_0,p_0) ==>  E(n_1, p_1 |   ( recv(n_1,n_Router,p_1)
&& matchEnty(p_1.IP_DST) && (p_0.IP_SRC == p_1.IP_SRC) && (p_0.IP_DST == p_1.IP_DST)
&& (p_0.PORT_SRC == p_1.PORT_SRC) && (p_0.PORT_DST == p_1.PORT_DST)
&& (p_0.PROTO == p_1.PROTO) && (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY ==
p_1.ORIG_BODY) && (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL)
&& (p_0.OPTIONS == p_1.OPTIONS) && (p_0.INNER_SRC == p_1.INNER_SRC)
&& (p_0.INNER_DEST == p_1.INNER_DEST) && (p_0.ENCRYPTED == p_1.ENCRYPTED))))
                                                                Router Rule_0
```

Figure 6.1. Rule of Router Model .

6.2 AAA

RADIUS security is composed of three components: authentication, authorization, and accounting. These three links in the RADIUS security chain are often referred to by their acronym, "AAA" [3] .

- **Authentication** , is the process that a RADIUS server determines whether a client (a person, a device, or a software process) is a legitimate user of the system, so that RADIUS server can prevents unauthorized users from accessing the system. Authentication usually involves some form of identification and a piece of secret information, such as a password.
- **Authorization** , is the process that a RADIUS server restricts what each user can and cannot do while logged into the system. goes hand-in-hand with the authentication process.
- **Accounting** , is the process that a RADIUS server monitors and records a client's use of the network and records the logon and logoff time of each user. So it's possible to correlate network access with malfunctions, security breaches, and other problems.

The accounting features of the RADIUS protocol can be used independently of RADIUS authentication or authorization. The RADIUS accounting functions allow data to be sent at the start and end of sessions, indicating the amount of resources (such as time, packets, bytes, and so on) used during the session. An Internet service provider (ISP) might use RADIUS access control and accounting software to meet special security and billing needs. The accounting port for RADIUS for most Cisco devices is 1646, but it can also be 1813 (because of the change in ports as specified in RFC 2139 leavingcisco.com).

How to model AAA server? Model Steps as shown in (Figure 6.7).

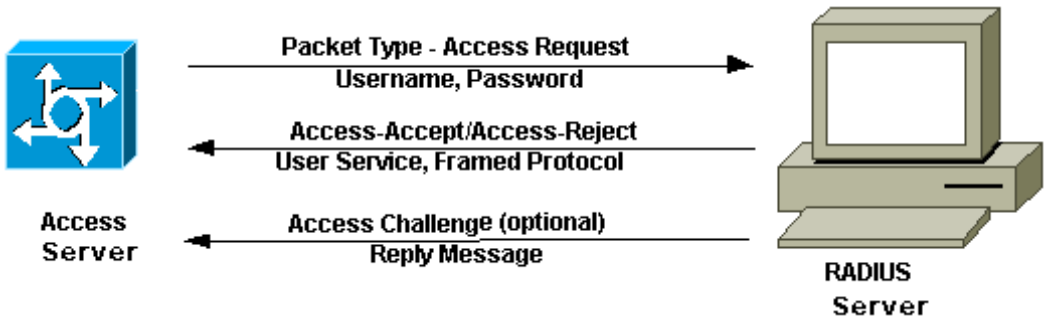

Figure 6.2. Traffic Flow in Radius Authentication Process.

1. When a client NAS is configured to use RADIUS, any user of the client presents authentication information to the client.(username and password are put in packet body in our model).
2. Once the NAS has obtained a packet with information about the user's name and password in body, it creates an encrypted "Access-Request" packet, and the request packet is then sent to RADIUS server for authentication.

3. Once the RADIUS server receives the request packet, it validates the sending Client by checking there is a matched user entry in the database. If the client is valid, The user entry in the database contains a list of requirements which must be met to allow access for the user. For simplicity, our abstract model will only reply with a "Access-Accept" packet. If any of the checks fail—if the user name doesn't exist, or the password is incorrect, and so on—the RADIUS server returns a "Access-Reject" message to the client.

   Of course, in real environment, depending on the security rules defined on the server, the client may have the opportunity to try again a certain number of times, after which the account is locked, either permanently (until an administrator unlocks it) or for a certain amount of time.

The FOL formulas are shown in (Figure 6.3)

```
Rule_0:

( send(n_AAA,n_0,p_0) ==>
E(n_1, p_1 |
( recv(n_1,n_AAA,p_1) && (p_1.PORT_DST == AUTHENTICATION_PORT_1812) && !(matchEnty(p_1.BODY))
&& (p_0.BODY == ACCESS_REJECT) && (p_0.IP_SRC == p_1.IP_DST) && (p_0.PORT_SRC == p_1.PORT_DST)
&& (p_0.IP_DST == p_1.IP_SRC) && (p_0.PORT_DST == p_1.PORT_SRC) && (p_0.PROTO == p_1.PROTO)
&& (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS)
&& (p_0.INNER_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == p_1.INNER_DEST)
&& (p_0.ENCRYPTED == p_1.ENCRYPTED))))

Rule_1:

( send(n_AAA,n_0,p_0) ==>
E(n_1, p_1 |
( recv(n_1,n_AAA,p_1) && (p_1.PORT_DST == AUTHENTICATION_PORT_1812) && matchEnty(p_1.BODY)
&& (p_0.BODY == AUTHORIZATION_RESPONSE) && (p_0.IP_SRC == p_1.IP_DST) && (p_0.PORT_SRC == p_1.PORT_DST)
&& (p_0.IP_DST == p_1.IP_SRC) && (p_0.PORT_DST == p_1.PORT_SRC) && (p_0.PROTO == p_1.PROTO)
&& (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS)
&& (p_0.INNER_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == p_1.INNER_DEST)
&& (p_0.ENCRYPTED == p_1.ENCRYPTED))))

Rule_2:
( send(n_AAA,n_0,p_0) ==>
E(n_1, p_1 |
( recv(n_1,n_AAA,p_1) && (p_1.PORT_DST == ACCOUNTING_PORT_1813) && matchEnty(p_1.BODY)
&& (p_0.IP_SRC == p_1.IP_DST) && (p_0.PORT_SRC == p_1.PORT_DST) && (p_0.IP_DST == p_1.IP_SRC)
&& (p_0.PORT_DST == p_1.PORT_SRC) && (p_0.BODY == ACCOUNTING_RESPONSE) && (p_0.PROTO == p_1.PROTO)
&& (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS)
&& (p_0.INNER_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == p_1.INNER_DEST)
&& (p_0.ENCRYPTED == p_1.ENCRYPTED))))
```

Figure 6.3. Rules of AAA server

Test in Verigraph
/*
* Test AAA <p/>
*

* | AAAClient a | ——| firewall | —-| AAA Server | <p/>
*
*/
Test case (packet directions): Assume firewall allows all packets

1. a -> b: expected result: SAT final result: SAT;

2. b -> a: expected result: UNSAT final result: UNSAT;

   expected result is UNSAT, because of flow isolation, AAA has never before sent a packet to client a.

6.3 CDN Network

Traditional DNS modes of access seriously affect the efficiency and quality of Internet users' access due to several factors that include service interruption of DNS server, delay, Compatibility between different networks, Retransmission and so on. For example, a user is in North America, while the origin that holds the content requested is all the way across the globe.

However, By adding a new layer of network architecture to the existing Internet, CDN [4] publishes the content of the website to the "edge" of the network closest to the user so that the user can obtain the content needed nearby to solve the problem of Internet network congestion and improve the speed of response to visiting the site. Its technical principle is to avoid bottlenecks and links on the Internet that may affect data transmission speed and stability as much as possible. The system places the content of the website closest to the user by placing a local cache acceleration in the carefully selected network, avoiding the above-mentioned "bottleneck of interconnection between networks" that affect the transmission performance of the Internet, thus accelerating the service across networks and carriers and achieving an effective solution to visiting the site landing slowly, due to the network distance and the router in the process of switching, which led to the current technology delay.

A CDN has Points of Presence (PoPs) or data centers that are situated around the world. Within each PoP are thousands of servers. Both the PoPs and servers help accelerate the speed at which content is delivered to the end user.

In order for users to be able to view content, CDN uses the caching principles and, regardless of the country they are in, CDNs must redirect requests to the nearest server. This will ensure the discussion of the loading time as short as possible. CDN key players like Akamai or OVH have thousands of servers in each country. This allows you to redirect users to servers that are no more than 100 kilometers away from it. And thus speed up the best access to content Here are two scenarios when CDN works (dynamic content):

- Once the request from a user is sent to the CDN, If the files are cached on PoP, they will be sent directly through the cache. In this case, the site hosting the content will not need to go to the site's server. This will allow faster access.
- If, on the other hand, the content is not in the cache or the cache entry has expired, then the edge server makes a request to the origin server to retrieve the information. The origin server is the source of truth for content and is capable of serving all of the content that is available on the CDN. When the edge server receives the response from the origin server, it stores the content in a cache based on the HTTP headers of the response. It will then send them to the user via PoP (Figure 6.4).

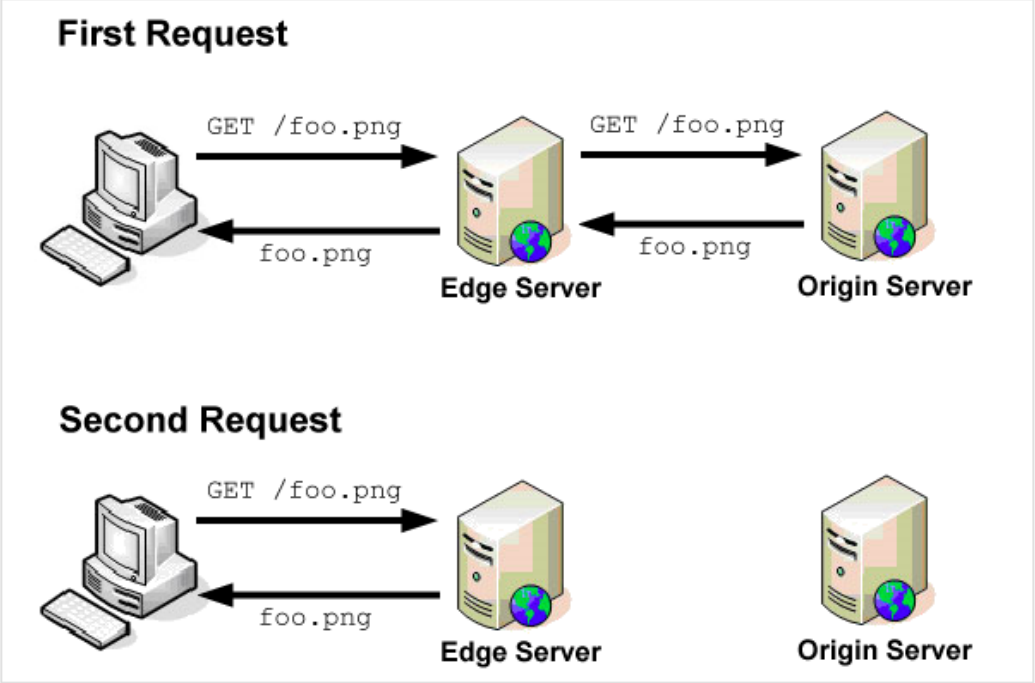

Figure 6.4. Traffic flow in CDN network [5].

The following image shows my CDN model.

1. The user enters the domain name www.web.com into the browser. When the browser first discovers that there is no dns cache locally, the browser requests the website's DNS server.
2. The DNS domain name resolver of the website sets the CNAME, points to www.web.51cdn.com, and the request points to the intelligent DNS load balancing system in the CDN network;

3. The intelligent DNS load balancing system resolves domain names and returns the fastest IP node to the user. Its model is shown in (Figure 6.6).

4. The user sends a request to the IP node (CDN server);

5. Because it is the first visit, the CDN server will request the original web site and cache the content;

6. The result of the request is sent to the user.

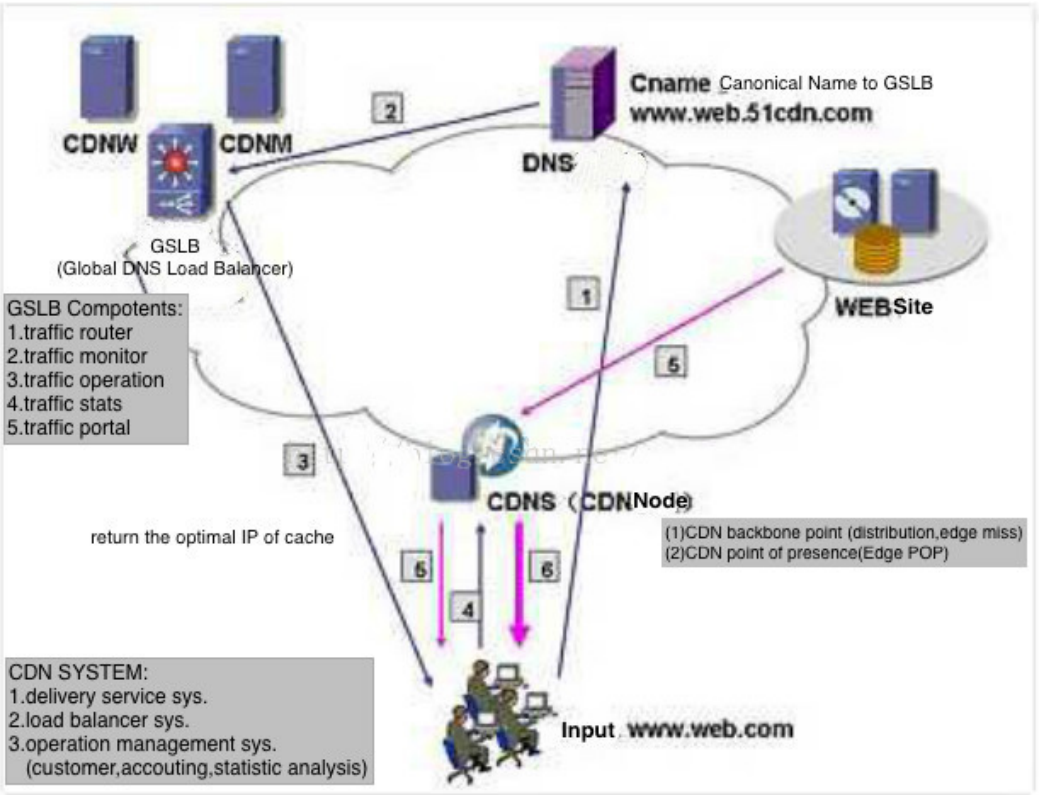

Figure 6.5. Traffic flow in CDN network [6].

6.4 SIP Server

SIP (Session Initiation Protocol) is a multimedia communication protocol established by IETF (Internet Engineering Task Force). It is a text-based application-layer control protocol used to create, modify, and release sessions for one or more

```
( send(n_GlobalDnsBalancer,n_0,p_0) ==>
E(n_1, p_1 |
( recv(n_1,n_GlobalDnsBalancer,p_1) && (p_1.PROTO == DNS_REQUEST)
&& matchEnty(p_1.IP_SRC, p_1.URL) && (p_0.INNER_DEST == ip_url)
&& (p_0.IP_SRC == p_1.IP_DST) && (p_0.IP_DST == p_1.IP_SRC) && (p_0.PROTO == DNS_RESPONSE)
&& (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.BODY == p_1.BODY)
&& (p_0.SEQUENCE == p_1.SEQUENCE) && (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL)
&& (p_0.OPTIONS == p_1.OPTIONS) && (p_0.INNER_SRC == p_1.INNER_SRC)
&& (p_0.ENCRYPTED == p_1.ENCRYPTED))))
```

Figure 6.6. Rule of Global DNS Server.

participants. Conversation composition: SIP sessions use up to four major components: the SIP user agent, the SIP registrar, the SIP proxy, and the SIP redirect server.The following is a summary of the various SIP components and their role in this process.

SIP User Agents (UA) (Figure 6.8)User agent are end-user devices such as mobile phones, multimedia handsets, PCs, PDAs, etc. used to create and manage SIP sessions. User Agent client sends a message. User Agent Server responds to the message.

Registration server The SIP registrar is a database that contains the location of all user agents in the domain. In SIP communications, these servers retrieve each other's IP address and other related information and send it to the SIP proxy server.

Proxy server The SIP proxy server accepts the SIP UA session request and queries the SIP registration server to obtain the address information of the recipient UA. It then forwards the session invitation information directly to the recipient UA (if it is in the same domain) or to the proxy server (if the UA is in another domain).

Redirect server The SIP redirect server allows the SIP proxy server to direct SIP session invitation information to external domains. The SIP redirect server can be on the same hardware as the SIP registrar and SIP proxy.

As we know, SIP protocol is a Client / Server protocol, so there are two kinds of SIP messages: request message and response message. So in my model, I define 4 packet types: SIP_REGISTER, SIP_INVITE, SIP_OK, SIP_ENDING. They can present the interactions between the user agent and SIP server by setup the packet 'proto' field. Besides, the SIP server model has all functions of SIP redirect server, SIP registrar, SIP proxy. User agent:

(1) sends a SIP_REGISTER packet to sip server, its 'body' stores the telephone number. Sip server will then store the number and IP_SRC of the packet into table.

(2) sends a SIP_INVITE packet to sip server, server firstly checks whether the IP_DST is server itself. If yes, the server must fetch the callee's IP address which corresponds to the number, and then forward the packet to callee. If no, it means the caller know the destination IP address, sip server will directly forward the packet.

The AAA server model in the following (Figure 6.9) contains all the functionalities of above SIP components.

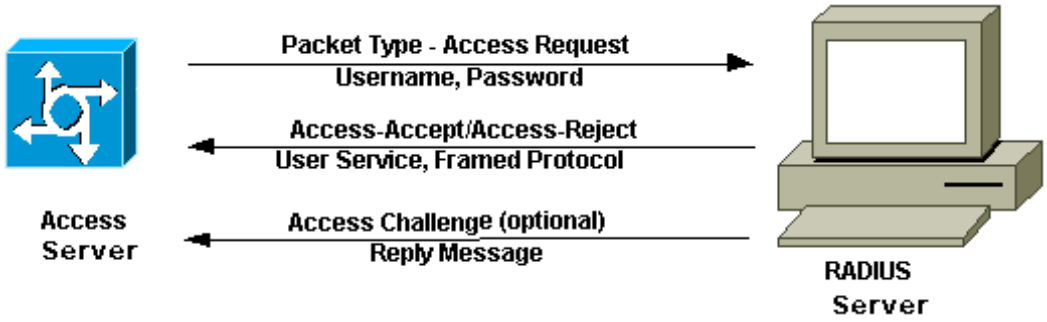

Figure 6.7. Traffic flow of radius)

```
1(( send(n_UA,n_0,p_0) && (p_0.URL == domain)) ==>
2    ((p_0.IP_SRC == ip_caller) && (p_0.IP_DST == ip_sipServer) && (p_0.BODY == num)
3    && (p_0.PROTO == SIP_REGISTE)))
4
```

Figure 6.8. Rule of UA (User agent) model

6.5 VPN

In this session, we focus on Intranet VPN. Intranet VPN is one of the most important VPN technologies. It forwards internal packets from one gateway to another, and connect the resources from the same company through the company's network architecture;

The VPN gateway achieves remote access by encrypting the data packets and converting the destination address of the data packet.

In our model, we assume the two gateways are called VPN access and VPN exit separately and they communicate bidirectionally. VPN access function enables a user in a VPN internal network A to send a private encrypted message to one another internal network B. The information of internal network is modeled by means of a function (*isInternal*). During the process, the exchanged packets must pass through the two VPN gateways and are processed.

VPN Access In order to send a packet towards network B whose inner source field is equal to null ($p0.inner\_src == null$), and the destination address of this packet must be a private address in network B, and it is not encrypted ($p\_0.encrypted == false$). Moreover, VPN access must firstly receive another packet (p_1) from network A whose source IP address is equal to the VPN exit ($p\_1.ip\_src == exitIp$) and destination IP address is equal to VPN access($p\_1.ip\_dest == accessIp$), the packet must be encrypted ($p\_1.encrypted == true$) when traverse the VPN tunnel (Figura 6.10).

When send to a encrypted packet ($p\_0.encrypted == true$) in an opposite direction to network A, the packet must have a inner source field that is not null

```
Rule_0:
( send(n_SipServer,n_0,p_0) ==>
E(n_1, p_1 |
( recv(n_1,n_SipServer,p_1) && (p_1.PROTO == SIP_REGISTE) && (p_1.IP_DST == ip_sipServer)
&& (p_1.URL == domain) && (p_0.IP_SRC == p_1.IP_DST) && (p_0.IP_DST == p_1.IP_SRC)
&& (p_0.PROTO == SIP_REGISTE_OK) && (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY)
&& (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE) && (p_0.EMAIL_FROM == p_1.EMAIL_FROM)
&& (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS) && (p_0.INNER_SRC == p_1.INNER_SRC)
&& (p_0.INNER_DEST == p_1.INNER_DEST) && (p_0.ENCRYPTED == p_1.ENCRYPTED))))

Rule_1:
(( send(n_SipServer,n_0,p_0) && (p_0.URL == domain)) ==>
E(n_1, p_1 |
( recv(n_1,n_SipServer,p_1) && (p_1.PROTO == SIP_INVITE) && (p_1.IP_DST == ip_sipServer)
&& (p_1.URL == domain) &&  E(n_2, p_2 |
( recv(n_2,n_SipServer,p_2) && (p_0.IP_DST == p_2.IP_SRC))) && (p_0.IP_SRC == p_1.IP_SRC)
&& (p_0.PROTO == p_1.PROTO) && (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY)
&& (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE) && (p_0.EMAIL_FROM == p_1.EMAIL_FROM)
&& (p_0.OPTIONS == p_1.OPTIONS) && (p_0.INNER_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == p_1.INNER_DEST)
&& (p_0.ENCRYPTED == p_1.ENCRYPTED))))
Rule_2:
(( send(n_SipServer,n_0,p_0) && !((p_0.URL == domain))) ==>
E(n_1, p_1 |
( recv(n_1,n_SipServer,p_1) && (p_1.PROTO == SIP_INVITE) && (p_1.IP_DST == ip_sipServer)
&& !((p_1.URL == domain)) &&  E(n_2, p_2 |
( send(n_SipServer,n_2,p_2) && (p_2.IP_SRC == ip_sipServer) && (p_2.IP_DST == ip_dns)
&& (p_2.PROTO == DNS_REQUEST) && (p_2.URL == p_1.URL))) &&  E(p_3 |
( recv(n_2,n_SipServer,p_3) && (p_3.IP_SRC == p_2.IP_DST) && (p_3.IP_DST == p_2.IP_SRC)
&& (p_3.PROTO == DNS_RESPONSE) && (p_3.URL == p_2.URL) && !((p_3.INNER_DEST == null))
&& (p_0.IP_DST == p_3.INNER_DEST))) && (p_0.URL == p_1.URL) && (p_0.IP_SRC == p_1.IP_SRC)
&& (p_0.PROTO == p_1.PROTO) && (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY)
&& (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE) && (p_0.EMAIL_FROM == p_1.EMAIL_FROM)
&& (p_0.OPTIONS == p_1.OPTIONS) && (p_0.INNER_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == p_1.INNER_DEST)
&& (p_0.ENCRYPTED == p_1.ENCRYPTED))))
```

Figure 6.9. Rule of SIP Server model

value ($p\_0.inner\_src! = null$), a destination address equal to VPN exit address ($p\_0.ip_dest == exitIp$). Of course, an unencrypted internal packet from network B must be received firstly whose inner source and the destination address is equal to null. Moreover, the two packets must satisfy the following requirements: ($p\_1.ip\_dest == p\_0.inner\_dest \wedge p\_1.ip\_src == p\_0.inner\_src$), and the sending packet has all other fields that are copied from the received packet, as it was shown in (Figure 6.11).

VPN Exit VPN exit function is used with VPN access, and their functions are particularly similar. Except when a packet is going to be sent to internal network, there must be received packet with the IP destination address equal to VPN exit ($p\_1.ip\_dest == exitIp$), IP source address equal to VPN access ($p\_1.ip\_src == accessIp$). It was shown in (Figure 6.12) . In the opposite direction, when VPN exit sends an encrypted packet to VPN access, the IP destination must be equal to access IP ($p\_0.ip\_dest == accessIp$) and IP source must be equal to exit IP ($p\_0.ip\_src == exitIp$). The source and destination addresses of inner header are same with that of the received unencrypted packet.

Besides, all other packet fields are always same in the sending packet and the received packet. It was shown in (Figure 6.13).

```
(( send(n_VpnAccess,n_0,p_0) && (p_0.INNER_SRC == null)) ==> E(n_1, p_1 |  (
recv(n_1,n_VpnAccess,p_1) && (p_1.IP_SRC == exitIp) && (p_1.IP_DST == accessIp)
&& (p_1.ENCRYPTED == true) && (p_0.IP_DST == p_1.INNER_DEST)
&& (p_0.IP_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == null)
&& (p_0.ENCRYPTED == false) && isInternal(p_0.IP_DST)
&& (p_0.PROTO == p_1.PROTO) && (p_0.ORIGIN == p_1.ORIGIN)
&& (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.BODY == p_1.BODY)
&& (p_0.SEQUENCE == p_1.SEQUENCE) && (p_0.EMAIL_FROM == p_1.EMAIL_FROM)
&& (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS))))
                                                        (VPN Access Rule_0)
```

Figure 6.10. VPN Access Rule_0)

```
(( send(n_VpnAccess,n_0,p_0) && !((p_0.INNER_SRC == null))) ==> E(n_1, p_1 | (
recv(n_1,n_VpnAccess,p_1) && !((p_1.ENCRYPTED == true))
&& (p_1.INNER_SRC == null) && (p_1.INNER_DEST == null)
&& (p_0.IP_SRC == accessIp) && (p_0.IP_DST == exitIp)
&& (p_0.INNER_DEST == p_1.IP_DST) && (p_0.INNER_SRC == p_1.IP_SRC)
&& (p_0.ENCRYPTED == true) && isInternal(p_0.INNER_SRC)
&& (p_0.PROTO == p_1.PROTO) && (p_0.ORIGIN == p_1.ORIGIN)
&& (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.BODY == p_1.BODY)
&& (p_0.SEQUENCE == p_1.SEQUENCE) && (p_0.EMAIL_FROM == p_1.EMAIL_FROM)
&& (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS))))
                                                        (VPN AccessRule_1)
```

Figure 6.11. VPN Access Rule_1)

Test in Verigraph
/*
* Test VPN <p/>
*
* | a | ——— | access | ——— | exit | ——— | b | <p/>
*
*/
Test case (packet directions):

1. a -> b: expected result: SAT final result: SAT;
2. b -> a: expected result: SAT final result: SAT;

6.6 IPv4-in-IPv6

With the rapid development of Internet, IPv4 is wearing away with every passing day, with the advent of time IPv6, it is an inevitable trend that IPv6 will replace

```
(( send(n_VpnExit,n_0,p_0) && (p_0.INNER_SRC == null)) ==> E(n_1, p_1 | (
recv(n_1,n_VpnExit,p_1) && (p_1.IP_SRC == accessIp) && (p_1.IP_DST == exitIp)
&& (p_1.ENCRYPTED == true) && (p_0.IP_DST == p_1.INNER_DEST)
&& (p_0.IP_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == null)
&& (p_0.ENCRYPTED == false) && isInternal(p_0.IP_SRC)
&& (p_0.PROTO == p_1.PROTO) && (p_0.ORIGIN == p_1.ORIGIN)
&& (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.BODY == p_1.BODY)
&& (p_0.SEQUENCE == p_1.SEQUENCE) && (p_0.EMAIL_FROM == p_1.EMAIL_FROM)
&& (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS))))
                                                    (VPN Exit  Rule_0)
```

Figure 6.12. VPN Exit Rule_0)

```
(( send(n_VpnExit,n_0,p_0) && !((p_0.INNER_SRC == null))) ==>  E(n_1, p_1 | (
recv(n_1,n_VpnExit,p_1) && !((p_1.ENCRYPTED == true)) && (p_1.INNER_SRC == null)
&& (p_1.INNER_DEST == null) && (p_0.IP_DST == accessIp) && (p_0.IP_SRC == exitIp)
&& (p_0.INNER_DEST == p_1.IP_DST) && (p_0.INNER_SRC == p_1.IP_SRC)
&& (p_0.ENCRYPTED == true) && isInternal(p_0.INNER_DEST)
&& (p_0.PROTO == p_1.PROTO) && (p_0.ORIGIN == p_1.ORIGIN)
&& (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.BODY == p_1.BODY)
&& (p_0.SEQUENCE == p_1.SEQUENCE) && (p_0.EMAIL_FROM == p_1.EMAIL_FROM)
&& (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS))))
                                                    (VPN Exit  Rule_1)
```

Figure 6.13. VPN Exit Rule_1)

IPv4. However, for some reasons, the process is long and tortuous. Therefore, in the current internet situation of coexistence of IPv4 and IPv6, it is of great significance to implement IPv4 in IPv6 Internet.

IPv4-in-IPv6 technology combines the tunnel technology and NAT technology and puts forward a more perfect IPv6 transition technology solutions. During the later stage of IPv4 to IPv6 transition, a large number of IPv6 networks have been deployed and isolated IPv4 sites may exist. You can create a tunnel on an IPv6 network to connect isolated IPv4 sites, which is similar to deploying the VPN on the IP network using tunnel technology. The tunnel connecting IPv4 isolated sites on the IPv6 network is called an IPv4 over IPv6 tunnel. This network function model needs the same VPN model's notion of exit and access gateways. As shown in (Figura 6.14), we call the two dual stack routers 'IPv4Exit' with an IP address 'exitIp' and ' IPv4Access' with an IP address 'accessIp'.

1. On the border device, the IPv4/IPv6 dual protocol stack is enabled and the IPv4 over IPv6 tunnel is configured.

2. After the border device **(IPv4Exit)** receives a packet not destined for the

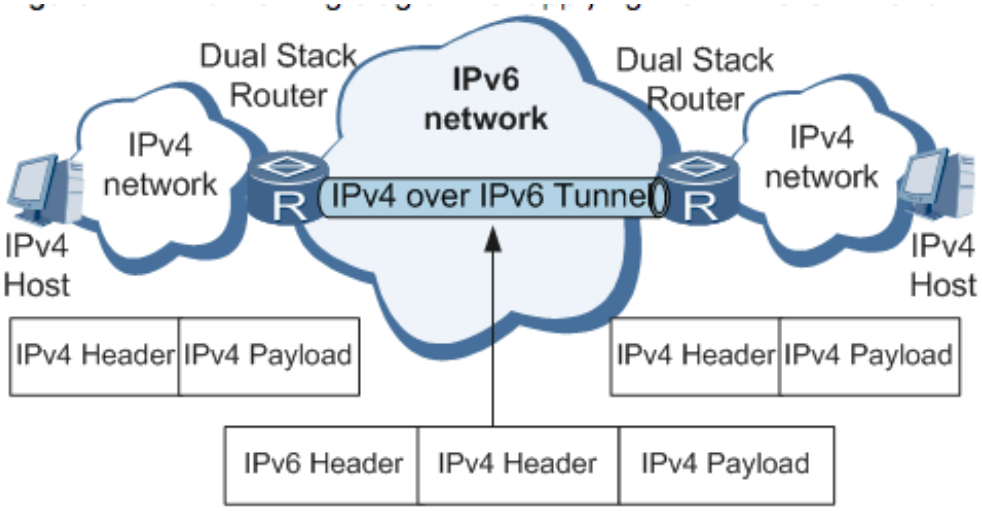

Figure 6.14. Networking diagram for applying the IPv4 over IPv6 tunnel)

device from the IPv4 network, the device appends an IPv6 header to the IPv4 packet and encapsulates the IPv4 packet as an IPv6 packet. During the process, the received IPv4 packet (p_1) is not encrypted ($!(p\_1.ENCRYPTED == true)$) and it has only one header, so its inner source and destination address are both null ($p\_1.INNER\_SRC == null)\&\&(p\_1.INNER\_DEST == null$). The sending packet (p_0) should have two packet header, the outer one is IPv6 header and the other is IPv4 header. The source and destination addresses of IPv4 header are equal to that of the received packet separately ($p\_0.INNER\_DEST == p\_1.IP\_DST)\&\&(p\_0.INNER\_SRC == p\_1.IP\_SRC$). Moreover, this IPv6 packet will be encrypted to be able to safely pass through the IPv6 network. The IPv6 header has a source IP equal to 'exitIp' and a destination IP equal to 'accessIp'. This model of the process is in Figure 6.15.

On the opposite direction, when IPv4Exit device receives a encrypted IPv6 packet with inner_src field not equal to null, its ip_src must be equal to 'accessIp' and its ip_dest must be equal to 'exitIp' ($p\_1.IP\_SRC == accessIp)\&\&(p\_1.IP\_DS$ $exitIp$). Then IPv4Exit will send an unencrypted IPv4 packet with empty inner_src and inner_dest to IPv4 network. This process is modeled in Figure 6.16.

3. On the IPv6 network, the encapsulated packet is transmitted to the remote border device (IPv4Access).

4. The remote border device **(IPv4Access)** decapsulates the packet, removes

the IPv6 header, and sends the decapsulated IPv4 packet to the IPv4 network. During the process, the received encrypted IPv6 packet (p_1) should have fixed IP addresses $(p\_1.IP\_SRC == exitIp)\&\&(p\_1.IP\_DST == accessIp)$. The inner source and destination address of the sending unencrypted packet (p_0) will be null $(p\_0.INNER\_SRC == null)\&\& (p\_0.INNER\_DEST == null)$, for all other packet fields, their values are same, the rules is in Figure 6.17

IPv4Access can also send an unencrypted IPv4 packet through an IPv6 network. The encapsulates process is similar with IPv4Exit, the difference is the IP addresses of sending packet p_0 $((p\_0.IP\_SRC == accessIp)\&\&(p\_0.IP\_DST == exitIp)$, as shown in Figure 6.18.

```
(( send(n_IPv4Exit,n_0,p_0) && !((p_0.INNER_SRC == null))) ==>
E(n_1, p_1 |
( recv(n_1,n_IPv4Exit,p_1) && !((p_1.ENCRYPTED == true)) && (p_1.INNER_SRC == null)
&& (p_1.INNER_DEST == null) && (p_0.IP_DST == accessIp) && (p_0.IP_SRC == exitIp)
&& (p_0.INNER_DEST == p_1.IP_DST) && (p_0.INNER_SRC == p_1.IP_SRC)
&& (p_0.ENCRYPTED == true) && isInternal(p_0.INNER_DEST) && (p_0.PROTO == p_1.PROTO)
&& (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY)
&& (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL)
&& (p_0.OPTIONS == p_1.OPTIONS))))
```

Figure 6.15. IPv4–In–IPv6 Exit Rule_0

```
(( send(n_IPv4Exit,n_0,p_0) && (p_0.INNER_SRC == null)) ==>
E(n_1, p_1 |
( recv(n_1,n_IPv4Exit,p_1) && (p_1.IP_SRC == accessIp) && (p_1.IP_DST == exitIp)
&& !((p_1.INNER_SRC == null)) && (p_0.IP_DST == p_1.INNER_DEST)
&& (p_0.IP_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == null)
&& (p_1.ENCRYPTED == true) && (p_0.ENCRYPTED == false)
&& isInternal(p_0.IP_SRC) && (p_0.PROTO == p_1.PROTO)
&& (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY)
&& (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL)
&& (p_0.OPTIONS == p_1.OPTIONS))))
```

Figure 6.16. IPv4–In–IPv6 Exit Rule_1

Test in Verigraph
/*
* Test IPv4 in IPv6 <p/>
*

```
(( send(n_IPv4Access,n_0,p_0) && (p_0.INNER_SRC == null)) ==>
E(n_1, p_1 |
        ( recv(n_1,n_IPv4Access,p_1) && (p_1.IP_SRC == exitIp)
&& (p_1.IP_DST == accessIp) && (p_1.ENCRYPTED == true)
&& (p_0.IP_DST == p_1.INNER_DEST) && (p_0.IP_SRC == p_1.INNER_SRC)
&& (p_0.INNER_DEST == null) && (p_0.ENCRYPTED == false)
&& isInternal(p_0.IP_DST) && (p_0.PROTO == p_1.PROTO)
&& (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY)
&& (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL)
&& (p_0.OPTIONS == p_1.OPTIONS))))
```

Figure 6.17. IPv4–In–IPv6 Access Rule_0

```
(( send(n_IPv4Access,n_0,p_0) && !((p_0.INNER_SRC == null))) ==>
E(n_1, p_1 |
        ( recv(n_1,n_IPv4Access,p_1) && !((p_1.ENCRYPTED == true))
&& (p_1.INNER_SRC == null) && (p_1.INNER_DEST == null) && (p_0.IP_SRC ==
accessIp) && (p_0.IP_DST == exitIp) && (p_0.INNER_DEST == p_1.IP_DST)
&& (p_0.INNER_SRC == p_1.IP_SRC) && (p_0.ENCRYPTED == true)
&& isInternal(p_0.INNER_SRC) && (p_0.PROTO == p_1.PROTO)
&& (p_0.ORIGIN == p_1.ORIGIN) && (p_0.ORIG_BODY == p_1.ORIG_BODY)
&& (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL)
&& (p_0.OPTIONS == p_1.OPTIONS))))
```

Figure 6.18. IPv4–In–IPv6 Access Rule_1

* | a | ——— | access | ——— | exit | ——— | b | <p/>
*
*/
Test case (packet directions):

1. a -> b: expected result: SAT final result: SAT;
2. b -> a: expected result: SAT final result: SAT;

6.7 MPLS

Multiprotocol Label Switching (MPLS)[7] is a technology used to rapidly forward data packets and combines the advantages of IP and ATM. It is designed to improve forwarding efficiency. Because IP forwarding is mostly performed by software, at least one longest matching lookup is performed for each hop forwarded,

and the complexity of the operation results in a slower forwarding speed. The main steps are as follows.

* **Step1:** The Ingress LER (Figure 6.19) receives the IP packet, analyzes the IP packet header and maps it to the FEC(Forwarding equivalence class: the same destination address or the same service class), and then adds the IP packet to the packet and sends the marked packet to the corresponding outgoing interface according to the LSP in the label forwarding table. During the process, we store the label in the 'options'field of the packet. It is an integer. Due to the label allocation depends on the IP header, my ingress model, will try to match the IP destination of the received packet, once an entry exists in the table, then the model will generate a random integer number called 'label'and assign it to the 'options'field of the packet (p_0.options==label), then forward it to other LSR. The process in java file is in the following code:

* **Step2:** After the LSR receives the packet with the tag, it only analyzes the tag header, does not care about the part above the tag header, searches the LSP according to the Label header, replaces the Label, and sends it to the corresponding outgoing interface. Due to the network chain is constructed by Verigraph in verification process, the forwarding path can not be decided by these LSRs in our model, so we will not model these LSRs. Their behavirs are just matching the inLabel (options) and modifying the outLabel (options) according to its routing tabel.

* **Step3:** The penultimate hop LSR receives the packet with the tag, finds the tag forwarding table, finds that the corresponding egress tag is an implicit null tag or an explicit null tag, and pops the tag and sends the IP packet to the last LSR. In this case, the 'options'field of the forwarding packet must be null (p_0.options==null).

* **Step4:** Performs Layer 3 routing on the last–hop Egress LER and forward packets based on the destination IP address of the packet.

```
1( send(n_IngressNode,n_0,p_0) ==>
2E(n_1, p_1 |
3( recv(n_1,n_IngressNode,p_1) && matchEnty(p_1.IP_DST) && (p_0.IP_SRC == p_1.IP_SRC)
4&& (p_0.IP_DST == p_1.IP_DST) && (p_0.PROTO == p_1.PROTO) && (p_0.ORIGIN == p_1.ORIGIN)
5&& (p_0.ORIG_BODY == p_1.ORIG_BODY) && (p_0.BODY == p_1.BODY) && (p_0.SEQUENCE == p_1.SEQUENCE)
6&& (p_0.EMAIL_FROM == p_1.EMAIL_FROM) && (p_0.URL == p_1.URL) && (p_0.OPTIONS == p_1.OPTIONS)
7&& (p_0.INNER_SRC == p_1.INNER_SRC) && (p_0.INNER_DEST == p_1.INNER_DEST)
8&& (p_0.ENCRYPTED == p_1.ENCRYPTED))))
```

Figure 6.19. Rule of Ingress node in MPLS network

Chapter 7

Conclusion

In conclusion, allows the interested actors to define the behavior of any VNF in a more developer- friendly Java-like fashion and allows the extraction of an abstract model from the Java code in order to verify the important network properties in a wider network environment, so I think it is possible to say that this project has reached its objectives and I hope it can be used in other wider works. It will be very helpful in terms of fault management, rapid upgrade, quickly meet market demand and reduce cost.

Chapter 8

Bibliography

Chapter 9

Appendix: Necessary Software Install Guide

The jar package that needs to be imported in this example project is in the following figure, and it can also be drowloaded here[**z3Jars**]:

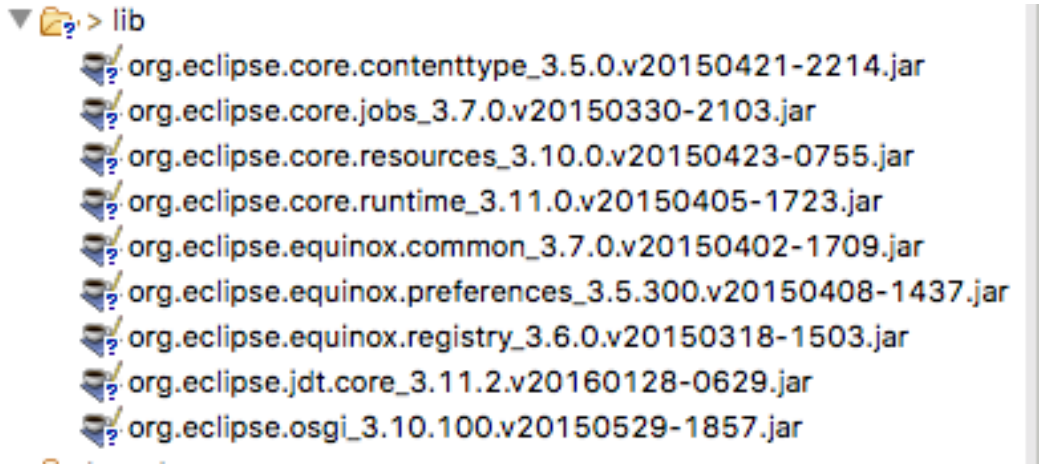

Figure 9.1. Necessary jar files for AST).

Bibliography

[1] *First-order logic*. URL: https://en.wikipedia.org/wiki/First-order_logic.

[2] *Implication in FOL*. URL: http://faculty.simpson.edu/lydia.sinapova/www/cmsc180/LN180_Johnsonbaugh-07/Overview_logic.htm.

[3] *Remote Authentication Dial In User Service (RADIUS)*. RFC 2865. IETF, June 2000. URL: https://tools.ietf.org/html/rfc2865.

[4] Athena Vakali Rajkumar Buyya Mukaddim Pathan. «Content Delivery Networks». In: (). ISSN: 1876-1100. URL: https://link.springer.com/book/10.1007/978-3-540-77887-5#about.

[5] *How content delivery networks (CDNs) work*. URL: https://www.nczonline.net/blog/2011/11/29/how-content-delivery-networks-cdns-work/.

[6] *Framework for Content Distribution Network Interconnection (CDNI)*. RFC 3466. IETF, Oct. 2015, pp. 10–37. URL: https://datatracker.ietf.org/doc/rfc7336/.

[7] Divya Kapil, Emmanuel S Pilli, and Ramesh C Joshi. «Live virtual machine migration techniques: Survey and research challenges». In: *Advance Computing Conference (IACC), 2013 IEEE 3rd International*. IEEE. 2013, pp. 963–969.