0.1 VERIGRAPH

VeriGraph is a verification tool based on the state of the art SMT solver, Z3. This tool aims at performing the VNF graph verification, through an intensive modelling activity. VeriGraph includes models of both the whole network and forwarding behaviours of each involved VNF. Those models are expressed by means of first-order logic formulas and allow to verify if reachability properties hold in the network (i.e., source node can reach a destination node passing through a set of network functions). In order to exhaustively verify all the possible service chains available in a given VNF graph, VeriGraph exploits an external tool, namely Neo4JManager, to extract all the paths from the source node to the destination node. Then, VeriGraph is able to generate a model for each extracted chain. Using the VNF chain model, the verification tool can perform a complete verification step. In detail, VeriGraph (and, in turn, Z3) looks for some assignment of appropriate values to uninterpreted functions and constant symbols that compose the VNF chain formulas, in order to make the chain formulas satisfiable. Here we present how network and VNFs in the catalog are modelled in VeriGraph, that is the set of the main formulas that represent the behaviours of both network and VNFs.

0.1.1 NETWORK MODEL

VeriGraph models the network as a set of network nodes that send and receive packets. Each packet has a static structure, which is:

$$\begin{aligned} Packet = \{&src, dest, inner\_src, inner\_dest, origin, origin\_body, body, \\ &seq, proto, emailFrom, url, options, encrypted\} \end{aligned} \tag{1a}$$

- *src* and *dest* are the source and destination addresses of the current packet;
- *inner_src* and *inner_dest* are the ultimate source and destination addresses of the encapsulated IPv4 packet;
- *origin* represents the network node that has originally created the packet;
- *origin_body* takes trace of the original content body of the packet, while *body* is the current body (that could be modified traversing the chain);
- *seq* is the sequence number of this packet;
- *proto* represents the protocol type and it can assume as values: *HTTP_REQUEST, HTTP_RESPONSE, POP3_REQUEST, POP3_RESPONSE, SMTP_REQUEST, SMTP_RESPONSE*. This set of values must be extended when the VNF catalog is enriched with new functions that use other protocols (e.g., DNS server, VPN gateways and other);

- *emailFrom* states for the email address that has sent a POP3 or SMTP message and it is modelled as an integer;
- *url* states for the web content of a HTTP message and it is modelled as an integer;
- *options* are the options values for the current packet.
- *encrypted* represents that if this field is set then the packet is encapsulated in other packet.

The structure of the packet needs to be extended in case of VNF specific fields are used by new models. When there is not explicit constraints imposed on which values a packet filed can assume, Z3 is allowed to assign any values to those fields. This is because Z3 looks for those values that satisfy all the conditions imposed by the formulas to verify. For instance, let us consider that VeriGraph has to check if a web server is reachable from a web client. Both network nodes are modelled so that they can send and receive only HTTP messages (i.e., $p.proto == HTTP\_RESPONSE \vee p.proto == HTTP\_REQUEST$) and no conditions are imposed on other fields. Hence Z3 can assign any value to, for instance, the *url* field, because this is not directly involved into the formulas for verifying the web server and client connection.
VeriGraph library provides also a set of functions for retrieving some useful information. All of these functions are uninterpreted functions supported by Z3, which means that they do not have any a priori interpretation, like usual programming language. Uninterpreted functions allow any interpretation that is consistent with the constraints over the function. Some functions supported by VeriGraph are:

- *Bool nodeHasAddress(node, address)*, which checks if *address* is an address associated to *node*;
- *Node addrToNode(address)*, which returns the node associated to the passed *address*;
- *Int sport(packet)* and *Int dport(packet)* return respectively the source and destination ports of *packet*.

The main functions that model operational behaviours in a network, provided by VeriGraph, are:

- *Bool send(node_src, node_dest, packet)*: the *send* function returns a boolean and represents the sending action performed by a source node (*node_src*) towards a destination node (*node_dest*) of a packet (*packet*);
- *Bool recv(node_src, node_dest, packet)*: this function returns a boolean and models a destination node (*node_dest*) that has received a packet (*packet*) from a source node (*node_src*).

Hence the *send* and *recv* functions (as well the other previously defined) are the means to impose conditions for describing how network and VNFs operate. In particular, VeriGraph has already a set of conditions imposed on those two functions in order to model the fundamentals principals of a correct forwarding behaviour. The formulas of such conditions are:

$$\begin{aligned} send(n_0, n_1, p_0) \implies & (n_0 \neq n_1 \wedge p_0.src \neq p_0.dest \wedge \\ & sport(p_0) \geq 0 \wedge sport(p_0) < MAX\_PORT \wedge \\ & dport(p_0) \geq 0 \wedge dport(p_0) < MAX\_PORT \wedge), \forall n_0, p_0 \end{aligned} \tag{2a}$$

$$recv(n_0, n_1, p_0) \implies send(n_0, n_1, p_0), \quad \forall n_0, p_0 \tag{3a}$$

Formula 2a states that the source and destination nodes (n_0 and n_1) must be different, as well source and destination addresses in the packet ($p_0.src$ and $p_0.dest$). The source and destination ports must also be defined in a valid range of values. In Formula 3a, we can find the conditions for receiving a packet. In this case we have to consider an additional constrain: if a packet is received by a node (n_1), this implies that this packet was previously sent to that node.
Finally, it is possible that source and destination nodes may be no directly connected, but they can exchange traffic through a set of functions. These functions process and potentially can modify received packets before forwarding them toward the final destination (e.g., NATs modify IP addresses).
In order to verify the correctness of reachability properties in presence of such functions, we have to assume that the original sent packet could be different from the received one. Hence VeriGraph can verify the reachability between the *src* and *dest* nodes in presence of a set of middleboxes thanks to the following formula:

$$\begin{aligned} \exists(n_0, n_1, p_0, p_1) \mid & (send(src, n_1, p_1) \wedge nodeHasAddr(src, p_1.src) \wedge \\ & nodeHasAddr(dest, p_1.dest) \wedge \\ & recv(n_0, dest, p_0) \wedge nodeHasAddr(dest, p_0.dest) \\ & \wedge p_1.origin == p_0.origin) \end{aligned} \tag{4a}$$

Here we are modeling the case of a source node (src) that is sending a packet (p_1) to a destination node ($dest$) ($send(src, n_1, p_1)$), of which we want to check the reachability between each other. The sent packet is delivered to the first chain node (n_1) that connect source and destination nodes and must also have the address of the *src* node as source address ($nodeHasAddr(src, p_1.src)$). As we have already explained, the destination node may receive a different packet from the one sent, because VNFs could modify the sent packet in its trip towards the destination. Thus we have to impose that the destination node receives a new packet (p_0) from the last chain node (n_0): the received packet must have the address of the *dest* node as destination, but it must have anyway the same origin of the sent packet ($p_1.origin == p_0.origin$).

0.1.2 VNF MODEL

The VNF catalogue supported by VeriGraph is composed of several network function models, which are: End-host, Mail Server/Client, Web Client/Server, Anti-spam, NAT, Web Cache, ACL firewall. Here we describe the formulas that model the functional behaviour of each function in catalog.

End-host model An end-host is a network node which sends packets towards a destination and receives packets from a source. The sent packets must satisfy some conditions (Formula 5a): *(i)* the end-host address is the source address; *(ii) origin* is the end-host itself; *(iii) origin_body* and *body* must be equal. The received packet (Formula 5b) must have the end-host address as destination.

$$\begin{aligned}(send(end\_host, n_0, p_0)) \implies\ & (nodeHasAddr(end\_host, p_0.src) \wedge \\ & p_0.origin == end\_host \wedge p_0.origin\_body == p_0.body) \wedge \\ & predicatesOnPktFields, \quad \forall(n_0, p_0)\end{aligned} \tag{5a}$$

$$\begin{aligned}(recv(n_0, end\_host, p_0)) \implies\ & (nodeHasAddr(end\_host, p_0.dest)), \\ & \forall(n_0, p_0)\end{aligned} \tag{5b}$$

Figure 1: End-host model.

This is a basic version of an endpoint in the service graph, which can be configured to behave as end-host-based model (i.e., servers and clients). VeriGraph support end-host configurations to specify which traffic flow an end-host sends, without changing its model (e.g., a client can generate packet with specific port number, destination address etc.). Initially *predicatesOnPktFields* is set to true and depending on the packet model configured by the user, this predicate will be appended with the assigned fields of the packet.

Mail Server model A mail server is a complex form of end-host which can function as POP3 or SMTP server. In fact, this kind of server can generate only *POP3_RESPONSE* or *SMTP_RESPONSE* messages addressed to a mail client, but the type of the response depends on the type of the mail server. For instance POP3 mail server is modelled for sending only *POP3_RESPONSE* (Formula 6a) and receiving just *POP3_REQUEST* messages (Formula 6b). In particular, a packet from a mail server is sent only if previously a *POP3_REQUEST* was received (Formula 6c). The same applies for an SMTP server. These set of formulas will be appended to the solver depending on the type of the end-host used in a test scenario.

$$(send(end\_host, n_0, p_0)) \implies (p_0.proto = POP3\_RESPONSE), \forall(n_0, p_0) \tag{6a}$$

$$(recv(n_0, end\_host, p_0)) \implies (p_0.proto = POP3\_REQUEST), \forall(n_0, p_0) \tag{6b}$$

$$(send(end\_host, n_0, p_0) \implies \exists(p_1)|recv(n_0, end\_host, p_1) \land (p_1.proto = POP3\_REQUEST), \forall(n_0, p_0) \tag{6c}$$

Figure 2: POP3 Mail Server model.

Mail Client model A mail client is a particular kind of end-host and similarly to a Mail Server it can behave as POP3 or SMTP client. This node is modelled so that it can receive *POP3_RESPONSE* or *SMTP_RESPONSE* messages only. In the case where the end-host set to behave as POP3 mail client, Formula 7a is appended to the available set of formulas in an end-host.

$$(recv(n_0, end\_host, p_0)) \implies (p_0.proto = POP3\_RESPONSE) \land p_0.src = ip\_pop3\_mail\_server, \forall(n_0, p_0) \tag{7a}$$

Figure 3: POP3 Mail Client model.

Web Client model The web client is an extension of the end-host model. This node is modelled so that if it receive *HTTP_RESPONSE* messages there must be a packet sent by this node whose destination address is the source address of the received packet (Formula 8a).

$$(recv(n_0, end\_host, p_0)) \implies (p_0.proto = HTTP\_RESPONSE) \land \exists(p_1)|send(end\_host, n_0, p_1) \land (p_0.src == p_1.dest), \forall(n_0, p_0) \tag{8a}$$

Figure 4: Web Client model.

Web Server model The web server model is built to send *HTTP_RESPONSE* packets only if the server has previously received a *HTTP_REQUEST* packet (Formula 9a). The sent and received packets must refer to the same *url* field ($p_1.url = p_0.url$) and the received packet protocol must be a $HTTP\_REQUEST$ (Formula 9b).

Anti-Spam model An anti-spam function was modelled to drop packets from blacklisted mail clients and servers. In fact, the anti-spam behaviour was based

$$
\begin{aligned}
&(send(end\_host, n_0, p_0)) \implies (p_0.proto = HTTP\_RESPONSE) \wedge \\
&\quad \exists(p_1) | recv(n_0, end\_host, p_1) \wedge p_1.url = p_0.url \wedge \\
&\quad p_0.dest = p_1.src, \forall(n_0, p_0)
\end{aligned} \tag{9a}
$$

$$
(recv(n_0, end\_host, p_0)) \implies (p_0.proto = HTTP\_REQUEST), \forall(n_0, p_0) \tag{9b}
$$

Figure 5: Web Server model.

on the assumption that each client interested in receiving a new message addressed to it, sends a `POP3_REQUEST` to the mail server in order to retrieve the message content. The server, in turn, replies with a `POP3_RESPONSE` which contains a special field (*emailFrom*) representing the message sender. The process of sending an email is similarly modelled through SMTP request and response messages. As evident from Formula 10a, an anti-spam rejects any message containing a black listed email address (that are set during the creation of the VNF chain model). However, according to Formula 10b the packet that does not involve mail protocol is forwarded only after having received it.

$$
\begin{aligned}
(send(anti\_spam, n_0, p_0) \wedge (p_0.protocol = POP3\_RESPONSE \vee \\
& p_0.protocol = POP3\_REQUEST)) \implies \\
& \neg isInBlackList(p_0.emailFrom) \\
& \forall n_0, p_0
\end{aligned} \tag{10a}
$$

$$
\begin{aligned}
(send(anti\_spam, n_0, p_0) \implies & \exists(n_1) \mid recv(n_1, anti\_spam, p_0)) \\
& \forall n_0, p_0
\end{aligned} \tag{10b}
$$

Figure 6: Anti-spam model.

The set of blacklist email addresses is configured through a public function (e.g., *parseConfiguration()*), which gives an interpretation to the *isInBlackList()* uninterpreted function. In particular if the blacklist is empty, VeriGraph will build a constraint like Formula 11a. Otherwise, let us suppose that the blacklist contains two elements (*BlackList=[mail1, mail2]*), VeriGraph will build the Formula 11b. In this case, VeriGraph is imposing that the *inInBlackList()* function returns *TRUE* if ($emailFrom == mail1$) or ($emailFrom == mail2$) is *TRUE*.

$$
(isInBlackList(emailFrom) == False), \forall emailFrom \tag{11a}
$$

$$
\begin{aligned}
(isInBlackList(emailFrom) == (emailFrom == mail1) \vee \\
& (emailFrom == mail2)), \forall emailFrom
\end{aligned} \tag{11b}
$$

NAT model A different type of function is a NAT, which needs the notion of internal and external networks. This kind of information is modelled by means of a function (*isPrivateAddress*) that checks if an address is registered as private or not. Private addresses are configured when the VNF chain model is initialized. As example of configuration, let us suppose that end-hosts *nodeA* and *nodeB* are internal nodes, hence VeriGraph must add Formula 12a among its constraint to verify. Here the *isPrivateAddress()* function returns *TRUE* if (*address* == *nodeA_addr*) or (*address* == *nodeB_addr*) is *TRUE*.

$$
\begin{aligned}
(isPrivateAddress(address) == (address == nodeA\_addr \lor \\
address == nodeB\_addr)), \forall address
\end{aligned} \tag{12a}
$$

In details, the NAT behaviour is modelled by two formulas. Formula 13a states for an internal node which initiates a communication with an external node. In this case, the NAT sends a packet (p_0) to an external address ($\neg isPrivateAddress(p_0.dest)$), if and only if it has previously received a packet (p_1) from an internal node ($isPrivateAddress(p_1.src)$). The received and sent packets must be equal for all fields, except for the *src*, which must be equal to the NAT public address (*ip_nat*).

$$
\begin{aligned}
&(send(nat, n_0, p_0) \land \neg isPrivateAddress(p_0.dest)) \implies p_0.src = ip\_nat \\
&\quad \land \exists (n_1, p_1) \mid recv(n_1, nat, p_1) \land isPrivateAddress(p_1.src) \\
&\quad \land p_1.origin = p_0.origin \land p_1.dest = p_0.dest \land p_1.seq\_no = p_0.seq\_no \\
&\quad \land p_1.proto = p_0.proto \land p_1.emailFrom = p_0.emailFrom \land p_1.url = p_0.url), \\
&\quad \forall (n_0, p_0)
\end{aligned} \tag{13a}
$$

$$
\begin{aligned}
&(send(nat, n_0, p_0) \land isPrivateAddress(p_0.dest)) \implies \neg isPrivateAddress(p_0.src) \\
&\quad \land \exists (n_1, p_1) \mid (recv(n_1, nat, p_1) \land \neg isPrivateAddress(p_1.src) \\
&\quad \land p_1.dest = ip\_nat \land p_1.src = p_0.src \land p_1.origin = p_0.origin \\
&\quad \land p_1.seq\_no = p_0.seq\_no \land p_1.proto = p_0.proto \land p_1.emailFrom = p_0.emailFrom \\
&\quad \land p_1.url = p_0.url) \land \exists (n_2, p_2) \mid recv(n_2, nat, p_2) \\
&\quad \land isPrivateAddress(p_2.src) \land p_2.dest = p_1.src \\
&\quad \land p_2.src = p_0.dest), \forall (n_0, p_0)
\end{aligned} \tag{13b}
$$

Figure 7: NAT model.

On the other hand, the traffic from the external network to the private is modelled by Formula 13b. In this case, if the NAT is sending a packet to an internal address ($isPrivateAddress(p_0.dest)$), this packet ($p_0$) must have an external address as its source ($\neg isPrivateAddress(p_0.src)$). Moreover, p_0 must be preceded by another packet (p_1), which is, in turn, received by the NAT and it is equal to p_0 for all the other fields. It is worth noting that, generally, a communication between internal and external nodes cannot be started by the external node in presence of a NAT. As a consequence, this condition is expressed in the Formula 13b by imposing that p_1 must be preceded by another packet p_2 ($recv(n_2, nat, p_2)$, sent to the NAT from an internal node ($isPrivateAddress(p_2.src)$).

Web Cache model A simple version of web cache can be modelled with five formulas (Fig. 8), where we have a notion of internal addresses (*isInternal* function), which are configured when the chain model is created. VeriGraph follows a similar approach to the NAT model (Formula 12a) for configuring the internal nodes. For instance, if the internal network is composed of two nodes *nodeA* and *nodeB*, VeriGraph will give an interpretation to the *isInternal* function by means of Formula 14a.

$$\begin{aligned}(isInternal(node) == (node == nodeA \vee \\ node == nodeB)), \forall node\end{aligned} \tag{14a}$$

This model was designed to work with web end-hosts (i.e., web client and server). In details, formula 15a states that: a packet sent from the cache to a node belonging to the external network ($\neg isInternal(n_0)$), implies a previous HTTP request packet ($p_0.proto = HTTP\_REQ$) is received from an internal node, which cannot be served by the cache ($\neg isInCache(p_0.url)$), otherwise the request would have not been forwarded towards the external network.

$$\begin{aligned}(send(cache, n_0, p_0) \wedge \neg isInternal(n_0)) \implies \neg isInCache(p_0.url) \\ \wedge p_0.proto = HTTP\_REQ \wedge \exists(n_1) \mid \\ isInternalNode(n_1) \wedge recv(n_1, cache, p_0)), \\ \forall(n_0, p_0)\end{aligned} \tag{15a}$$

$$\begin{aligned}(send(cache, n_0, p_0) \wedge isInternal(n_0)) \implies isInCache(p_0.url) \\ \wedge p_0.proto = HTTP\_RESP \wedge p_0.dest = p_1.src \wedge \\ \wedge \exists(p_1) \mid p_1.url = p_0.url \wedge \\ recv(n_0, cache, p_1)), \forall(n_0, p_0)\end{aligned} \tag{15b}$$

$$\begin{aligned}(recv(n_0, cache, p_0) \wedge isInternal(n_0)) \implies \\ p_0.proto = HTTP\_REQ, \forall(n_0, p_0)\end{aligned} \tag{15c}$$

$$\begin{aligned}(recv(n_0, cache, p_0) \wedge \neg isInternal(n_0)) \implies \\ p_0.proto = HTTP\_RESP \wedge p_1.dest = p_0.src \wedge \\ \exists(p_1) \mid send(cache, n_0, p_1)), \forall(n_0, p_0)\end{aligned} \tag{15d}$$

$$\begin{aligned}isInCache(u_0) \implies \exists(p_1, n_1) \mid \\ recv(n_1, cache, p_1) \wedge p_1.url = u_0 \wedge \neg isInternal(n_1)) \\ \forall(u_0)\end{aligned} \tag{15e}$$

Figure 8: Web cache model.

Formula 15b states that a packet sent from the cache to the internal network contains a `HTTP_RESPONSE` for an URL which was in cache when the request has been received. We also state that the packet's target URL received from the

internal network is the same as the response ($p_1.url = p_0.url$). This is followed by Formula 15c, which states the received packet from the internal node must be a `HTTP_REQUEST`. On the other hand a packet received from the external node must be a `HTTP_RESPONSE` message and there must be another packet sent from the cache to that node(Formula 15d).
The final formula (Formula 15e) expresses a constraint that the *isInCache()* function must respect. In particular, we state that a given URL (u_0) is in cache if (and only if) a request packet was received for that URL from the external network.

ACL firewall model An ACL firewall is a simple firewall that drops packets based on its internal Access Control List (ACL), configured when the chain model is initialized. In particular the ACL list is managed through the uninterpreted function $acl\_func()$. A possible interpretation is given by VeriGraph through the Formula 16a, when the ACL list contains two entries, like for example $ACL = [< src_1, dest_1 >, < src_2, dest_2 >]$.

$$\begin{aligned}(acl\_func(a,b) == ((a == src_1 \wedge b == dest_1) \vee \\ (a == src_2 \wedge b == dest_2))), \forall a, b\end{aligned} \tag{16a}$$

Hence, if an ACL firewall sends a packet, this implies that the firewall has previously received a packet of which the source and destination address are not contained in the ACL list.

$$\begin{aligned}(send(fw, n_0, p_0)) \implies &(\exists(n_1)|recv(n_1, fw, p_0)\wedge \\ &\neg acl\_func(p_0.src, p_0.dest)), \\ &\forall(n_0, p_0)\end{aligned} \tag{17a}$$

Figure 9: ACL Firewall model

Field modifier Field modifier function is in charge of simple task - packet field modification. In other words, this network function let's to forward the packets by changing the fields available in the List 1a except the ones shown in Formula 18a. $predicatesOnPktFields$ is set to true and depending on modifications introduced by the user on the packet fields, this predicate will be appended with the updated fields of the packet.

IDS Intrusion detection system (IDS) function monitors a network for malicious activity or policy violations. We model the simplest IDS network function that also acts like an intrusion prevention system function which is best compared to a firewall. In general, this function performs the similar reasoning as in the case of Anti-spam model, where in this case blacklist contains the set of "suspicious" strings that the body of the packet may carry. Let us suppose that the

$$
\begin{aligned}
(send(modifier, n_0, p_0)) \implies &(\exists (n_1, p_1) | recv(n_1, modifier, p_1) \land predicatesOnPktFields \land \\
&p_1.encrypted = p_0.encrypted \land p_1.origin = p_0.origin \land \\
&\land p_1.inner\_src = p_0.inner\_src \land p_1.inner\_dest = p_0.inner\_dest \land \\
&p_1.orig\_body = p_0.orig\_body \land p_1.src = p_0.src, \\
&\forall (n_0, p_0)
\end{aligned} \tag{18a}
$$

Figure 10: Field modifier model

blacklist contains two elements (*BlackList=[keylogger, Brutus]*), VeriGraph will build the Formula 19a. In this case, VeriGraph is imposing that the *inInBlackList()* function returns *TRUE* if (*body* == *keylogger*) or (*body* == *Brutus*) is *TRUE*.

$$
\begin{aligned}
(isInBlackList(body) == (body == keylogger) \lor \\
(body == Brutus)), \forall body
\end{aligned} \tag{19a}
$$

Formula 20a states that IDS forwards a packet whose protocol is `HTTP_REQUEST` or `HTTP_RESPONSE` only if this packet is received and does not contain blacklisted string in the body of the packet.

$$
\begin{aligned}
(send(ids, n_0, p_0) \land (p_0.protocol = HTTP\_RES \lor \\
&p_0.protocol = HTTP\_REQ)) \implies \\
&\exists (n_1) \mid recv(n_1, ids, p_0) \land \neg isInBlackList(p_0.body), \\
&\forall n_0, p_0
\end{aligned} \tag{20a}
$$

$$
\begin{aligned}
(send(ids, n_0, p_0) \implies \\
&(p_0.protocol = HTTP\_RES \lor p_0.protocol = HTTP\_REQ) \\
&(nodeHasAddr(ids, p_0.src) \\
&\forall n_0, p_0
\end{aligned} \tag{20b}
$$

Figure 11: Anti-spam model.

VPN access VPN access function enables a user to establish virtual network connection and exchange private encrypted messages. This network function model needs the same NAT model's notion of internal and external networks. This kind of information is modelled by means of a function (*isPrivateAddress*). This network node sends packets towards a VPN exit function or receive from it. If the inner source address ($p_0.inner\_src$) of the encapsulated packet is not equal to null, the packets are sent only if a packet (p_1) was received. More importantly the source/destination address of that packet must be equal to the inner source/destination of the packet being sent, the *encrypted* field is set to false and the rest of the fields must be equal. In this scenario VPN access

$$
\begin{aligned}
&(send(access, n_0, p_0) \land p_0.inner\_src = null) \implies isPrivateAddress(p_0.dest)\\
&\quad p_0.encrypted \neq true \land \exists(n_1, p_1) \mid recv(n_1, access, p_1) \land p_1.src = vpnExitIp\\
&\quad \land p_1.encrypted = true \land p_1.dest = vpnAccessIp \land p_1.inner\_src = p_0.src\\
&\quad \land p_1.inner\_dest = p_0.dest \land p_1.seq = p_0.seq \land p_1.body = p_0.body\\
&\quad \land p_1.proto = p_0.proto \land p_1.emailFrom = p_0.emailFrom \land p_1.url = p_0.url),\\
&\quad \land p_1.origin = p_0.origin \land p_1.options = p_0.options \land p_1.origin\_body = p_0.origin\_body,\\
&\quad \forall(n_0, p_0)
\end{aligned} \tag{21a}
$$

Figure 12: VPN access model.

$$
\begin{aligned}
&(send(access, n_0, p_0) \land p_0.inner\_src \neq null) \implies isPrivateAddress(p_0.inner\_src)\\
&\quad \land p_0.src = vpnAccessIp \land p_0.dest = vpnExitIp \land p_0.inner\_dest \neq vpnAccessIp\\
&\quad \land p_0.encrypted = true \land \exists(n_1, p_1) \mid recv(n_1, access, p_1) \land p_1.src = p_0.inner\_src\\
&\quad \land p_1.encrypted \neq true \land p_1.dest = p_0.inner\_dest \land p_1.inner\_src = null\\
&\quad \land p_1.inner\_dest = null \land p_1.seq = p_0.seq \land p_1.body = p_0.body\\
&\quad \land p_1.proto = p_0.proto \land p_1.emailFrom = p_0.emailFrom \land p_1.url = p_0.url)\\
&\quad \land p_1.origin = p_0.origin \land p_1.options = p_0.options \land p_1.origin\_body = p_0.origin\_body,\\
&\quad \forall(n_0, p_0)
\end{aligned} \tag{22a}
$$

Figure 13: VPN access model.

function encloses a packet within another, that has IP source of *vpnAccessIp* and destination of *vpnExitIp*. However, the inner destination address needs to be different then *vpnAccessIp* and finally the *encrypted* field needs to be set to true, since the data is first encrypted and encapsulated before it is sent to the remote VPN exit server (Formula 21a).
In order to send a packet in an opposite direction, the packet's destination address needs to belong to an internal network ($isPrivateAddress(p_0.inner\_src)$) and the content must be decrypted. Moreover, there must be a packet (p_1) received whose source address is equal to a *vpnExitIP* and destination address of this network node. Apart from this, the packet needs to be decapsulated ($p_1.dest = p_0.inner\_dest \land p_1.src = p_0.inner\_src$) by copying all the other fields of the received packet, as it was shown in Formula 22a.

VPN exit VPN exit function is used in parallel with VPN access function, whose models are identical except in this case a packet (p_0) is sent to an internal network only if there is a packet sent by VPN access function ($p_1.src = vpnAccessIp$) and the packet being sent from this network function towards to the VPN access needs to have *vpnAccessIp* on its destination address field ($p_0.dest = vpnAccessIp$). It is evident from the Fig. 14 all the other fields of the packet being sent must be equal to the packet that was received.

$$
\begin{aligned}
&(send(exit, n_0, p_0) \wedge p_0.inner\_src = null) \implies isPrivateAddress(p_0.src)\\
&\quad p_0.encrypted \neq true \wedge \exists(n_1, p_1) \mid recv(n_1, access, p_1) \wedge p_1.src = vpnAccessIp\\
&\quad \wedge p_1.encrypted = true \wedge p_1.dest = vpnExitIp \wedge p_1.inner\_src = p_0.src\\
&\quad \wedge p_1.inner\_dest = p_0.dest \wedge p_1.seq = p_0.seq \wedge p_1.body = p_0.body\\
&\quad \wedge p_1.proto = p_0.proto \wedge p_1.emailFrom = p_0.emailFrom \wedge p_1.url = p_0.url),\\
&\quad \wedge p_1.origin = p_0.origin \wedge p_1.options = p_0.options \wedge p_1.origin\_body = p_0.origin\_body,\\
&\quad \forall(n_0, p_0)
\end{aligned} \tag{23a}
$$

$$
\begin{aligned}
&(send(exit, n_0, p_0) \wedge p_0.inner\_src \neq null) \implies isPrivateAddress(p_0.inner\_src)\\
&\quad \wedge p_0.src = vpnExitIp \wedge p_0.dest = vpnAccessIp \wedge p_0.inner\_dest \neq vpnExitIp\\
&\quad \wedge p_0.encrypted = true \wedge \exists(n_1, p_1) \mid recv(n_1, access, p_1) \wedge p_1.src = p_0.inner\_src\\
&\quad \wedge p_1.encrypted \neq true \wedge p_1.dest = p_0.inner\_dest \wedge p_1.inner\_src = null\\
&\quad \wedge p_1.inner\_dest = null \wedge p_1.seq = p_0.seq \wedge p_1.body = p_0.body\\
&\quad \wedge p_1.proto = p_0.proto \wedge p_1.emailFrom = p_0.emailFrom \wedge p_1.url = p_0.url)\\
&\quad \wedge p_1.origin = p_0.origin \wedge p_1.options = p_0.options \wedge p_1.origin\_body = p_0.origin\_body,\\
&\quad \forall(n_0, p_0)
\end{aligned} \tag{23b}
$$

Figure 14: VPN exit model.

VNF configurations

The semantic of the configuration parameters passed to VeriGraph depends on the VNF type. Having described the models of the VNFs supported by VeriGraph and how to configure them in Section 0.1.2, we briefly recap what we expect as input for each VNF in the catalogue:

- **NAT**: a set of private addresses that represent the hosts in the internal network;
- **VPN access**: IP addresses of VPN access and exit network functions;
- **VPN exit**: IP addresses of VPN access and exit network functions;
- **IDS**: the set of blacklist strings;
- **Web Cache**: the list of network nodes that belong to the internal network;
- **Anti-spam**: the set of blacklist email addresses;
- **ACL**: a set of $< source, destination >$ pair of addresses, which are not allowed to communicate between each other;
- **End-host**: end-host configurations in the form of Packet Model. This means that we specify which traffic flow end-hosts send (e.g., a client can generate packet with specific port number, destination address etc.).

Depending on the type of the end-host (Mail Client/Server, Web CLient/Server) corresponding additional formulas for that type are appended.

- **Field Modifier**: a Packet Model containing the fields needs to be modified;

To better understand the information that VeriGraph needs to create the verification scenario, we show an example of JSON file [1] that contains the configurations of each VNF involved in a generic chain, where we have included also the end-hosts configuration to have a complete understanding of the network scenario:

```
{ "nodes": [ {
       "id": "mail-cliet",
       "description": "traffic flow specification",
       "configuration": ["ip_server", "ip_client", "25"]
  },
    { "id": "nat",
      "description": "internal address",
      "configuration": ["ip_client1", "ip_client2"]
    },
    { "id": "fw",
      "description": "acl entries",
      "configuration": [
       { "val1": "ip_client1", "val2": "ip_client3" },
       { "val1": "ip_client2", "val2": "ip_client3" }
       ]
    },
    { "id": "antispam",
      "description": "bad emailFrom values",
      "configuration": ["2"]
    },
    { "id": "mail-server",
    "description": "traffic flow specification",
    "configuration": ["ip_server", "ip_client", "25"]
    }
   ]
}
```

[1]Note that this is note the actual implementation of how VeriGraph supports the function configuration, but a simple example to clarify what VeriGraph expects.