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.

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:

$$\begin{aligned} Packet = \{&src, dst, origin, origin\_body, body, \\ &seq, proto, emailFrom, url, options\} \end{aligned} \tag{1a}$$

- *src* and *dst* are the source and destination addresses of the current 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.

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_dst, packet, time_t)*: the *send* function returns a boolean and represents the sending action performed by a source node (*node_src*) towards a destination node (*node_dst*) of a packet (*packet*) at certain time (*time_t*);
- *Bool recv(node_src, node_dst, packet, time_t)*: this function returns a boolean and models a destination node (*node_dst*) that has received a packet (*packet*) at time *time_t* 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:

$$\begin{aligned}(send(n_0, n_1, p_0, t_0)) \implies & (n_0 \neq n_1 \wedge p_0.src \neq p_0.dst \wedge \\ & sport(p_0) \geq 0 \wedge sport(p_0) < MAX\_PORT \wedge \\ & dport(p_0) \geq 0 \wedge dport(p_0) < MAX\_PORT \wedge \\ & t_0 > 0), \quad \forall n_0, p_0, t_0\end{aligned} \tag{2a}$$

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

$$
\begin{aligned}
(recv(n_0, n_1, p_0, t_0)) \implies & (n_0 \neq n_1 \wedge p_0.src \neq p_0.dst \wedge \\
& sport(p_0) \geq 0 \wedge sport(p_0) < MAX\_PORT \wedge \\
& dport(p_0) \geq 0 \wedge dport(p_0) < MAX\_PORT \wedge \\
& \exists(t_1) | (send(n_0, n_1, p_0, t_1) \wedge t_1 < t_0 \wedge \\
& t_1 > 0 \wedge t_0 > 0), \quad \forall n_0, p_0, t_0
\end{aligned}
\tag{3a}
$$

$p_0.dst$). The source and destination ports must also be defined in a valid range of values and the sending time must be a positive value. In Formula 3a, we can find the conditions for receiving a packet (similar to the formula for sending), but 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 ca verify the reachability between the *src* and *dest* nodes in presence of a set of middleboxes thanks to the following formula:

$$
\begin{aligned}
\exists(n_0, n_1, p_0, p_1, t_0, t_1) \mid & (send(src, n_1, p_1, t_1) \wedge nodeHasAddr(src, p_1.src) \wedge \\
& p_1.origin == src \wedge nodeHasAddr(dest, p_1.dst) \wedge \\
& recv(n_0, dest, p_0, t_0) \wedge nodeHasAddr(dest, p_0.dst) \\
& \wedge p_1.origin == p_0.origin)
\end{aligned}
\tag{4a}
$$

Here we are modelling the case of a source node (src) that is sending a packet (p_1) to a destination node dst ($send(src, n_1, p_1, t_1) \wedge nodeHasAddr(dest, p_1.dst)$), 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 ($p_1.origin == p_0.origin$).

Please note that Formula 4 can be substituted by a simplified version thanks to the condition expressed in formula 3; the simpler formula is the following:

$$
\begin{aligned}
\exists(n_0, p_0, t_0) \mid & (recv(n_0, dest, p_0, t_0) \wedge nodeHasAddr(dest, p_0.dst) \wedge \\
& p_0.origin == src)
\end{aligned}
\tag{5a}
$$

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 and Learning 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 6a): *(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 6b) must have the end-host address as destination.

$$(send(end\_host, n_0, p_0, t_0)) \implies (nodeHasAddr(end\_host, p_0.src) \wedge p_0.origin == end\_host \wedge p_0.origin\_body == p_0.body) \quad \forall(n_0, p_0, t_0) \tag{6a}$$

$$(recv(n_0, end\_host, p_0, t_0)) \implies (nodeHasAddr(end\_host, p_0.dest)), \quad \forall(n_0, p_0, t_0) \tag{6b}$$

Figure 1: End-host model.

At this version of VeriGraph, the tool does not support end-host configurations, as well as the other end-host-based models (i.e., servers and clients). In other words, we cannot 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.). It could be useful to extend VeriGraph to support end-host configurations.

Mail Server model A mail server is a complex form of end-host. In fact, this kind of server can generate only *POP3_RESPONSE* or *SMTP_RESPONSE* messages addressed to a mail client (Formula 7a), but the type of the response depends on the previous messages. In particular, a *POP3_RESPONSE* packet is sent only if previously a *POP3_REQUEST* was received (Formula 8a) and similarly for the *SMTP_RESPONSE* packets (Formula 8b). A mail server is also modelled for receiving just *POP3_REQUEST* or *SMTP_REQUEST* messages (Formula 8c).

$$(send(mail\_server, n_0, p_0, t_0)) \implies (nodeHasAddr(mail\_server, p_0.src) \wedge p_0.origin = mail\_server \wedge p_0.origin\_body = p_0.body \wedge (p_0.proto = POP3\_RESPONSE \vee p_0.proto = SMTP\_RESPONSE) \wedge p_0.dst = ip\_mail\_client \wedge p_0.emailFrom = mail\_server), \quad \forall(n_0, p_0, t_0) \tag{7a}$$

(7b)

$$(send(mail\_server, n_0, p_0, t_0) \land p_0.proto = POP3\_RESPONSE) \implies \exists(p_1, t_1)|recv(n_0, mail\_server, p_1, t_1) \land p_1.proto = POP3\_REQUEST), \forall(n_0, p_0, t_0) \quad (8a)$$

$$(send(mail\_server, n_0, p_0, t_0) \land p_0.proto = SMTP\_RESPONSE) \implies \exists(p_1, t_1)|recv(n_0, mail\_server, p_1, t_1) \land p_1.proto = SMTP\_REQUEST), \forall(n_0, p_0, t_0) \quad (8b)$$

$$(recv(n_0, mail\_server, p_0, t_0)) \implies (nodeHasAddr(mail\_server, p_0.dst) (p_0.proto = POP3\_REQUEST \lor p_0.proto = SMTP\_REQUEST)), \forall(n_0, p_0, t_0) \quad (8c)$$

Figure 2: Mail Server model.

Mail client model A mail client is a particular kind of end-host (Formulas 9a and 9b). This node is modelled so that it can send *POP3_REQUEST* or *SMTP_REQUEST* messages only. Those messages must have a mail server address as destination ($p_0.dst = ip\_mail\_server$) and the *emailFrom* field must indicate the mail client node itself (Formula 9a).

$$(send(mail\_client, n_0, p_0, t_0)) \implies (nodeHasAddr(mail\_client, p_0.src) \land p_0.origin = mail\_client \land p_0.origin\_body = p_0.body \land (p_0.proto = POP3\_REQUEST \lor p_0.proto = SMTP\_REQUEST) \land p_0.dst = ip\_mail\_server \land p_0.emailFrom = mail\_client), \forall(n_0, p_0, t_0) \quad (9a)$$

$$(recv(n_0, mail\_client, p_0, t_0)) \implies (nodeHasAddr(mail\_client, p_0.dst)), \forall(n_0, p_0, t_0) \quad (9b)$$

Figure 3: Mail Client model.

Web Client model The web client is based on the end-host model (Formulas 10a and 10b), but it can generate only *HTTP_REQUEST* packets with a web server address as destination (Formula 10a).

$$(send(web\_client, n_0, p_0, t_0)) \implies (nodeHasAddr(web\_client, p_0.src) \land p_0.origin = web\_client \land p_0.origin\_body = p_0.body \land p_0.proto = HTTP\_REQUEST \land p_0.dst = ip\_web\_server), \forall(n_0, p_0, t_0) \quad (10a)$$

$$(recv(n_0, web\_client, p_0, t_0)) \implies (nodeHasAddr(web\_client, p_0.dst)), \forall(n_0, p_0, t_0) \quad (10b)$$

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 11a). The sent and received packets must refer to the same *url* field ($p_1.url = p_0.url$):

$$
\begin{aligned}
&(send(web\_server, n_0, p_0, t_0)) \implies (nodeHasAddr(web\_server, p_0.src) \wedge \\
&\quad p_0.origin = web\_server \wedge p_0.origin\_body = p_0.body \wedge \\
&\quad p_0.proto = HTTP\_RESPONSE \wedge \exists (p_1, t_1) | (t_1 < t_0 \\
&\quad recv(n_0, web\_server, p_1, t_1) \wedge p_1.url = p_0.url \wedge \\
&\quad p_1.proto = HTTP\_REQUEST \wedge p_0.dst = p_1.src \wedge p_0.src = p_1.dst)), \\
&\quad \forall (n_0, p_0, t_0)
\end{aligned} \tag{11a}
$$

$$
\begin{aligned}
&(recv(n_0, mail\_server, p_0, t_0)) \implies (nodeHasAdrr(web\_server, p_0.dst)), \\
&\quad \forall (n_0, p_0, t_0)
\end{aligned} \tag{11b}
$$

Figure 5: Web Server model.

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 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 12a, an anti-spam rejects any message containing a black listed email address (that are set during the creation of the VNF chain model). On the other hand, Formula 12b is needed in order to state that a `POP3_REQUEST` message is forwarded only after having received it in a previous time instant.

$$
\begin{aligned}
&(send(anti\_spam, n_0, p_0, t_0) \wedge p_0.protocol = POP3\_RESPONSE) \implies \\
&\qquad \neg isInBlackList(p_0.emailFrom) \wedge \exists (n_1, t_1) \mid (t_1 < t_0 \\
&\qquad \wedge recv(n_1, anti\_spam, p_0, t_1)) \\
&\qquad \forall n_0, p_0, t_0
\end{aligned} \tag{12a}
$$

$$
\begin{aligned}
&(send(anti\_spam, n_0, p_0, t_0) \wedge p_0.protocol = POP3\_REQUEST) \implies \\
&\qquad \exists (n_1, t_1) \mid (t_1 < t_0 \wedge recv(n_1, anti\_spam, p_0, t_1)) \\
&\qquad \forall n_0, p_0, t_0
\end{aligned} \tag{12b}
$$

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 13a. Otherwise, let us suppose that the blacklist contains two elements (*BlackList=[mail1, mail2]*), VeriGraph will build the Formula 13b. In this case, VeriGraph is imposing that the *inInBlackList()* function returns *TRUE* if either ($emailFrom == mail1$) or ($emailFrom == mail2$) are *TRUE*.

$$(isInBlackList(emailFrom) == False), \forall emailFrom \tag{13a}$$

$$(isInBlackList(emailFrom) == (emailFrom == mail1) \vee (emailFrom == mail2)), \forall emailFrom \tag{13b}$$

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 14a among its constraint to verify. Here the *isPrivateAddress()* function returns *TRUE* if both ($address == nodeA\_addr$) and ($address == nodeB\_addr$) are *TRUE*.

$$(isPrivateAddress(address) == (address == nodeA\_addr \wedge address == nodeB\_addr)), \forall address \tag{14a}$$

In details, the NAT behaviour is modelled by two formulas. Formula 15a 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 ($\neg isPrivateAddress(p_0.dst)$), 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).

$$\begin{aligned}
&(send(nat, n_0, p_0, t_0) \wedge \neg isPrivateAddress(p_0.dst)) \implies p_0.src = ip_nat \\
&\quad \wedge \exists (n_1, p_1, t_1) \mid (t_1 < t_0 \wedge recv(n_1, nat, p_1, t_1) \wedge isPrivateAddress(p_1.src) \\
&\quad \wedge p_1.origin = p_0.origin \wedge p_1.dst = p_0.dst \wedge p_1.seq_no = p_0.seq_no \\
&\quad \wedge p_1.proto = p_0.proto \wedge p_1.emailFrom = p_0.emailFrom \wedge p_1.url = p_0.url), \\
&\quad \forall (n_0, p_0, t_0)
\end{aligned} \tag{15a}$$

$$\begin{aligned}
&(send(nat, n_0, p_0, t_0) \wedge isPrivateAddress(p_0.dst)) \implies \neg isPrivateAddress(p_0.src) \\
&\quad \wedge \exists (n_1, p_1, t_1) \mid (t_1 < t_0 \wedge recv(n_1, nat, p_1, t_1) \wedge \neg isPrivateAddress(p_1.src) \\
&\quad \wedge p_1.dst = ip_nat \wedge p_1.src = p_0.src \wedge p_1.origin = p_0.origin \\
&\quad \wedge p_1.seq_no = p_0.seq_no \wedge p_1.proto = p_0.proto \wedge p_1.emailFrom = p_0.emailFrom \\
&\quad \wedge p_1.url = p_0.url) \wedge \exists (n_2, p_2, t_2) \mid (t_2 < t_1 \wedge recv(n_2, nat, p_2, t_2) \\
&\quad \wedge isPrivateAddress(p_2.src) \wedge p_2.dst = p_1.src \wedge p_2.dst = p_0.src \\
&\quad \wedge p_2.src = p_0.dest), \forall (n_0, p_0, t_0)
\end{aligned} \tag{15b}$$

Figure 7: NAT model.

On the other hand, the traffic from the external network to the private is modelled by Formula 15b. In this case, if the NAT is sending a packet to an internal address ($isPrivateAddress(p_0.dst)$), 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 15b by imposing that p_1 must be preceded by another packet p_2 $((t_2 < t_1 \wedge recv(n_2, nat, p_2, t_2))$, 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 three formulas (Formulas 17a, 17b and 17c), 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 14a) 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 16a.

$$(isInternal(node) == (node == nodeA \wedge node == nodeB)), \forall node \tag{16a}$$

This model was designed to work with web end-hosts (i.e., web client and server). In details, formula 17a 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 $(p_0.proto = HTTP\_REQ)$ and received from an internal node, which cannot be served by the cache $(\neg isInCache(p_0.url, t_0))$, otherwise the request would have not been forwarded towards the external network.

$$\begin{aligned}(send(cache, n_0, p_0, t_0) \wedge \neg isInternal(n_0)) \implies \neg isInCache(p_0.url, t_0) \\ \wedge p_0.proto = HTTP\_REQ \wedge \exists (t_1, n_1) \mid (t_1 < t_0 \\ \wedge isInternalNode(n_1) \wedge recv(n_1, cache, p_0, t_1)), \\ \forall (n_0, p_0, t_0)\end{aligned} \tag{17a}$$

$$\begin{aligned}(send(cache, n_0, p_0, t_0) \wedge isInternal(n_0)) \implies isInCache(p_0.url, t_0) \\ \wedge p_0.proto = HTTP\_RESP \wedge p_0.src = p_1.dst \wedge p_0.dst = p_1.src \wedge \\ \wedge \exists (p_1, t_1) \mid (t_1 < t_0 \wedge p_1.proto = HTTP\_REQ \\ \wedge p_1.url = p_0.url \wedge recv(n_0, cache, p_1, t_1)), \\ \forall (n_0, p_0, t_0)\end{aligned} \tag{17b}$$

$$\begin{aligned}isInCache(u_0, t_0) \implies \exists (t_1, t_2, p_1, p_2, n_1, n_1) \mid (t_1 < t_2 \wedge t_1 < t_0 \wedge t_2 < t_0 \\ \wedge recv(n_1, cache, p_1, t_1) \wedge recv(n_2, cache, p_2, t_2) \wedge p_1.proto = HTTP\_REQ \\ \wedge p_1.url = u_0 \wedge p_2.proto = HTTP\_RESP \wedge p_2.url = u_0 \wedge isInternal(n_2)) \\ \forall (u_0, t_0)\end{aligned} \tag{17c}$$

Figure 8: Web cache model.

Formula 17b 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 received from the internal network is a `HTTP_REQUEST` and the target URL is the same as the response $(p_1.protocol = HTTP\_REQ \wedge p_1.url = p_0.url)$.

The final formula (Formula 17c) expresses a constraint that the *isInCache()* function must respect. In particular, we state that a given URL (u_0) is in

cache at time t_0 if (and only if) a request packet was received at time t_1 (where $t_1 < t_0$) for that URL and a subsequent packet was received at time t_2 (where $t_2 < t_0 \wedge t_2 > t_1$) carrying the corresponding `HTTP_RESPONSE`.

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 though the Formula 18a, when the ACL list contains two entries, like for example $ACL = [< src_1, dst_1 >, < src_2, dst_2 >]$.

$$\begin{aligned}(acl\_func(a,b) == ((a == src_1 \wedge b == dst_1) \vee \\ (a == src_2 \wedge b == dst_2))), \forall a, b\end{aligned} \tag{18a}$$

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.

$$\begin{aligned}(send(fw, n_0, p_0, t_0)) \implies (\exists (t_1, n_1) | t_1 < t_0 \wedge recv(n_1, fw, p_0, t_1) \wedge \\ \neg acl\_func(p_0.src, p_0.dst)), \\ \forall (n_0, p_0, t_0)\end{aligned} \tag{19a}$$

Figure 9: ACL Firewall model

Learning firewall model The learning firewall behaviour follows the fundamental principals of network forwarding: if the firewall is sending a packet, this implies that the function has previously received that packet (Formula 20a). However the forwarding decisions of a learning firewall depend on an internal Access Control List: each ACL entry specifies a pair of addresses allowed to exchange packets and are configured through the same mechanism defined for the ACL firewall model (Formula 18a). The ACL entries define also the direction of the allowed traffic flows, which is, for instance, if the ACL list contains an entry like $< nodeA\_addr, nodeB\_addr >$, the learning firewall will forward packet from node A to B, but no the vice-versa. The traffic in the opposite direction (i.e., from B to A) is allowed if and only if node A has previously opened a connection toward B (that is A has sent a packet to B - Formula 20b):

$$(send(fw, n_0, p_0, t_0)) \implies (\exists (t_1, n_1) |\ t_1 < t_0 \wedge recv(n_1, fw, p_0, t_1)), \forall (n_0, p_0, t_0) \tag{20a}$$

$$\begin{aligned}(send(fw, n_0, p_0, t_0) \wedge \neg acl\_func(p_0.src, p_0.dst)) \implies (\exists (n_1, t_1) \mid send(fw, n_1, p_1, t_1) \wedge \\ t_1 + 1 < t_0 \wedge acl\_func(p_1.src, p_1.dst) \wedge p_0.src = p_1.dst \wedge p_0.dst = p_1.src \wedge \\ p_0.src\_port = p_1.dst\_port \wedge p_0.dst\_port = p_1.src\_port), \\ \forall (n_0, p_0, t_0)\end{aligned} \tag{20b}$$

Figure 10: Learning Firewall model

1.2.1 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 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;
- **Web Cache**: the list of network nodes that belong to the internal network;
- **Anti-spam**: the set of blacklist email addresses;
- **ACL and Leaning firewalls**: a set of $< source, destination >$ pair of addresses, which are allowed (Learning firewall) or not (ACL firewall) to communicate between each other;
- **End-host, Mail Client/Server, Web CLient/Server**: at this version of VeriGraph, the tool does not support end-host configurations. This means that we cannot specify which traffic flow end-hosts send without changing their models (e.g., a client can generate packet with specific port number, destination address etc.). It could be useful to extend end-host model in this direction.

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:

```
1  { "nodes": [ {
2        "id": "mail-cliet",
3        "description": "traffic flow specification",
4        "configuration": ["ip_server", "ip_client",
            "25"]
5          },
6      { "id": "nat",
7        "description": "internal address",
8        "configuration": ["ip_client1", "ip_client2"]
9      },
10     { "id": "fw",
11       "description": "acl entries",
12       "configuration": [
13        { "val1": "ip_client1", "val2": "ip_client3" },
14        { "val1": "ip_client2", "val2": "ip_client3" }
15        ]
16     },
17     { "id": "antispam",
```

[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.

```
18        "description": "bad emailFrom values",
19        "configuration": ["2"]
20      },
21      { "id": "mail-server",
22      "description": "traffic flow specification",
23      "configuration": ["ip_server", "ip_client", "25"]
24      }
25     ]
26 }
```