Policy-based Verification Method for Configurations of Large Network with Header-space Analyses

Toshio Tonouchi
System Platform Research Laboratories, NEC

***Abstract*— Configuration of network is getting complex because the network has been equipped with much functionality. Meanwhile, the network should satisfy many requirements for sophisticated multi-tenancy, high-level security and so on. For example, a flow which should be secure has to go through a firewall. However, it is difficult for an operator to verify whether the configuration in large network can satisfy these requirements. The verification takes a lot of time and a lot of human work. In addition, the human operator may inherently overlook an erroneous configuration. In this paper, we propose a policy language, which can specify the requirements. We also propose two implementation designs of the policy language. The one of the methods is estimated to verify the configuration of larg network.**

Keywords—Network Configration, Verification, Header Space Analysis, Policy

I. INTRODUCTION

Configuration of network is getting complex because network has been equipped with much functionality. For example, in a network having virtual networks behind a firewall, the configuration of the firewall and the virtual networks should be consistent. An inconsistent configuration may cause the unreachability among the firewall and virtual networks.

Meanwhile, the network should satisfy many requirements for sophisticated multi-tenant network, high-level security and so on. For example, flows for a customer should not share the same network with the different customers. A network operator has to verify whether the configurations satisfy the requirements, but it is difficult for an operator to manually verify it. It takes a lot of time and a lot of human work. As a result, the human operator may inherently overlook an erroneous configuration.

In this paper, we propose a policy language which can specify the requirements. We also propose a verification method, which can automatically verify whether the network configuration satisfies the requirements given in the policy specifications. The operator can specify requirements in the policy specifications. We also show the design of a verification tool, which reads the policy specifications, gets the configuration of the network elements, and verifies whether the given policy is satisfied in the network.

The contributions of this paper are follows:

- We defined a new policy language based on the well-known first-order predicate logic. We expect that a lot of operators are familiar to the policy language.
- We show two types of implementation designs of the verification tool by using the existing methods [1] [4]. Through performance estimations, we show the one of the methods is suitable for the verification of large network.

II. RELATED WORK

HSA [1] models a flow table as so-called a transfer function, and calculates the function for a header of flow at a switch port. A packet header consists of a list of attribute-value pairs. The value can be a set value. For example, the value of Attribute “ipv4_dest” can be “10.20.30.*”. In short, it can handle a bit sequence in which some bits may be wild cards. This header is called a header space.

In HSA, Transfer function $\tau_{p_i}: P \to H$ is defined for an input port $p_i \in P$. The domain of $\tau_p(p')$ is whole the header space goes from Port p to Port p' in a switch. $\tau_p(p')(h_p)$ means the header space at Output port p' of a flow incoming from Port p with Header Space h_p. We can also calculate the header space of a flow which goes through the several switches. For example, in the sequence of the switches in Fig. 1, we can calculate Header Space $\tau_{p_n}(p_o)$ at Port p_e from Port p_i with Header Space h_{p_i} can be $\tau_{p_n}(p_o) = \tau_{p_n}(p_o) \circ \cdots \circ \tau_{p_1}(p_2) \circ \tau_{p_i}(p_0)(h_{p_i})$.

The calculation time of HSA is the cube of flow entries for calculating the header spaces for whole switch ports in the network [4]. As a result, HSA cannot apply a large scale network.

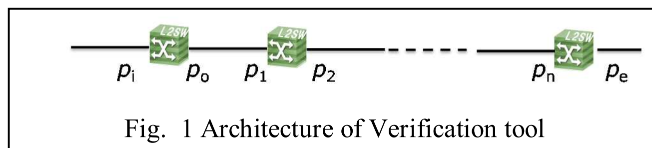

Fig. 1 Architecture of Verification tool

NetPlumber[2] uses HSA and verify the network configuration. It provides a policy language called FlowExp. The operator can specify requirements in FlowExp. NetPlumber checks, in start-up time and runtime, whether the network satisfies the requirement written in FlowExp.

As mentioned before HSA is not so fast for the runtime verification. NetPlumber uses an incremental checking approach in order to make verification time shorter. Firstly, before the runtime verification, it calculates header spaces and stores the calculation result into a so-called plumbing graph. The plumbing graph has the same topology with the physical network to be verified. NetPlumber uses these calculation results in the plumbing graph, and can firstly verify the

© Copyright IEICE – The 18th Asia-Pacific Network Operations and Management Symposium (APNOMS) 2016

network. Secondly, NetPlumber maintains the plumbing graph when a flow entry is added / modified / deleted. This is the reason why it is call an incremental checking approach.

The incremental checking approach is effective after the initial calculation has finished, but it is not effective when the verification right after the network is constructed. NetPlumber has to make a plumber graph for the newly constructed network, and it takes a lot of time.

B-HSA[4] has less powerful expression power than HSA, but it can calculate the header spaces of whole switch ports in the network faster than HSA. HSA calculates the header spaces of a flow, while B-HSA calculates the header spaces of the whole network at once. B-HSA can re-use the calculation results which were calculated in other flows. This is the reason why B-HSA is faster than HSA. However, B-HSA cannot calculate the header space of each flow. This is a disadvantage of B-HSA.

B-HSA can be expressed as two functions: $\beta_{in}: P \rightarrow H$ and $\beta_{out}: P \rightarrow H$. $\beta_{in}(p)$ represents the union set of whole header spaces at Port p of incoming flows. $\beta_{out}(p)$ represents the union set of whole header spaces at Port p of outgoing flows.

B-HSA does not have a policy language while HSA has a NetPlumber with FlowExp. In this work, we propose the designs of the policy language for B-HSA and the verification tool. It can verify network efficiently because B-HSA is used.

VeriFlow[3] is a verifier which uses a different calculation approach as well as NetPlumber. This verification tool is the proxy of the OpenFlow controller. It is placed in the control link between the controller and each OpenFlow switch. VeriFlow monitors OpenFlow packets, and detects an event of add / modify / delete of a flow entries. When it finds the event, it, then, verifies the network. Veriflow verifies the network fast, but it can be only used in OpenFlow network. Most of network has legacy network elements. We should verify the whole network including legacy network elements.

Frenetic [5] is a declarative specification language for OpenFlow controller. It can define the network configuration. The operator can compose elements of Frenetic specifications, and define the whole network. The operator can write a network configuration with this declarative specifications more easily and correctly than the procedure-based controller program, such as POX[7] , NOX[6], Flood Light[8] and so on.

NetKat[9] is a formal specification language for OpenFlow controller. The correctness of the language is mathematically proved. However, it is a different issue that a programming language is correctly designed from the configuration of the network satisfies the requirements.

III. Policy-based Verification Tool

A. Approach

The goal of this work is to verify whether the large-scale network satisfies the requirement written in the policy language or not. In order for a fast verification, we use B-HSA as well as HSA. The architecture is shown in Fig. 2. The inputs are a policy and, flow tables and the topology of the network. Translator translates a policy into a set of propositions. Checker calculates header spaces with B-HSA and HSA if necessary, and checks the propositions.

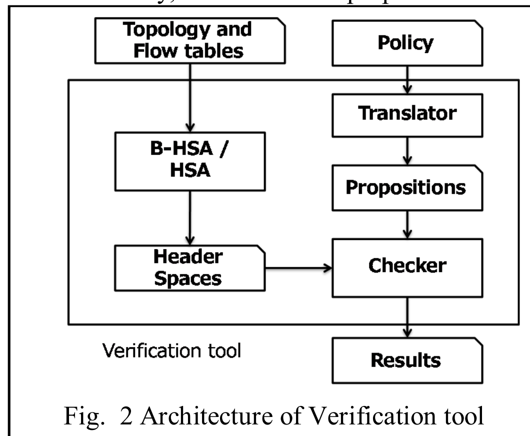

Fig. 2 Architecture of Verification tool

B. Examples of Policy

We give three examples of policy specifications.

- Example 1. Flow in subnet A should be with vlan_id=100. The network is shown in Fig. 3.

ports_in_ nw_A = (**port**("p_a1"), **port**("p_a2"));
forall *p* **in** *ports_in_ nw_A* {
  inhd(p) ⊂ [vlan_id=100],
  outhd(p) ⊂ [vlan_id=100];
}

"**inhd**(p)" means the header space at Port p.

- Example 2. Flows through Subnet A must go through the firewall in Fig. 3.

forall *p* **in** *ports_in_ nw_A* {
  iflow(p < **ports** < **port**("fw2_2")) ;
}

"**iflow**(p < **ports** < **port**("fw2_2"))" means that the incoming flow at Port p has to go through Port "fw2_2". "**ports**" means any path between Port p and Port "fw2_2".

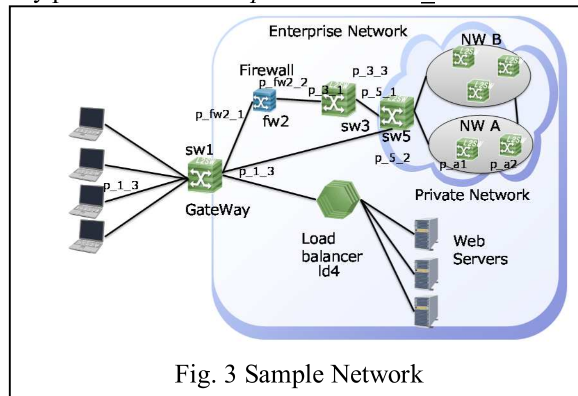

Fig. 3 Sample Network

- Example 3. A flow coming from the gateway in Fig. 4 goes through an SSL-VPN gateway or an IPSec-VPN gateway.

oflow(port(sw4_1) > **ports** >(sslvpn|IPvpn));

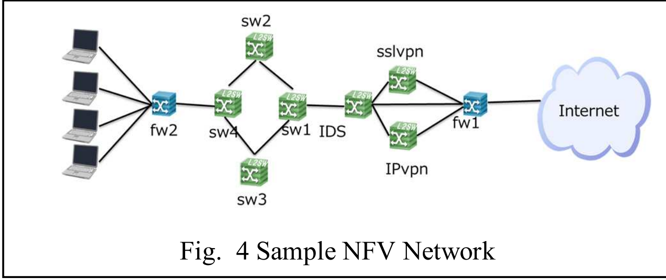

Fig. 4 Sample NFV Network

C. *Syntax of Policy Language*

We give the abstract syntax of the policy language in BNF.

```
/**** First-order predication ****/
[start] ::= ([dec] “;” | [pred] “;”)* ;
[pred] ::= “forall” [var] “in” [set_value] [pred]“;”
     | “exists” [var] “in” [hostSet] [pred]
     | “{“ [pred] “}” | [prop];
[prop] ::= [atomicProp] | [prop] “,” [prop] | [prop] “|”
      [prop]
     | “(“ [prop] “)”;
[dec] ::= [var] “=” [val] “;”
[val] ::= [direct_value] | [set_value];
[direct_value] ::= [header_val] | [port];

/**** Set ****/
[set_value] ::= [set_value] “&” [set_value]
     | [set_value] “+” [set_value]
     | “(“ [port](“,” [port])* “)” ;

/**** Proposition ****/
[atomicProp] ::= [exp] “⊂” [exp]
     | iflow([flow]) | oflow([flow]) ;
[exp] ::= [header_val] | [var] ;
[header_val] ::= inhd([port]) | outhd([port])
     | [header]
[iflow] ::= [port] (“<” [portexp])* “<” [port];
[oflow] ::= [port] (“>” [portexp] )* “<” [port];
[header] ::= “ [“ (attr_name “=” value)* “]”
[port] ::= [var] | port “(“ str “)” | ports
```

D. *Semantic of Policy Language*

We give the semantic of the policy language. Non-terminal symbols from [*start*] to [prop] represent the first-order logic, and, therefore, the semantic is well known and clear. Non-terminal symbol of [set_value] represents the set theory, and the semantics is also clear.

We have to show, in this paper, the semantics of [atomicProp] and **inhd** and **outhd** functions.

- **inhd**(p) is the union of header spaces of whole flows which go into Port p.
- **outhd**(p) is the union of header spaces of whole flows which go out of Port p.
- [exp] in “[exp] ⊂ [exp]” can be [header_val]. “[exp] ⊂ [exp]” is, therefore, represented as “[header_val$_1$] ⊂ [header_val$_2$]”. This holds if and only if each attribute value in [header_val$_1$] is a subset of the attribute value of [header_val$_2$].
  We consider that “[attr$_1$=v$_1$]” has attribute other than “attr$_1$”, whose values are universal sets. Therefore, “[attr$_1$=v$_1$] ⊂ [attr$_1$=v$_1$, attr$_2$=v$_2$]” is true because the formula is considered as “[attr$_1$=v$_1$, attr$_2$=any]⊂ [attr$_1$=v$_1$, attr$_2$=v$_2$]”.
- **iflow**($p1 < p2 < \ldots < p3$) holds if and only if the flow which reaches Port *p1* goes through *p3*, …, *p2*, and *p1*. For example it is false if some flow from *p1* does not go through *p2*.
- **oflow**($p1 > p2 > \ldots > p3$) is true if and only if any flow from *p1* goes through *p2*, …, and *p3*.
- “*p1* < **ports** < *p2*” and “*p2* > **ports** > *p1*” represent any flow from *p2* to *p1*.

E. *Implementation Designs of Policy Language*

We show the implementation designs of the policy language. Implementation of the part of the first order logic and the part of the set theory is clear. The implementation of “[exp] ⊂ [exp]” is also clear. We do not mention the implementation designs of these features in this paper.

We have to show the implementation designs of **inhd**, **outhd**, **iflow**, and **oflow**.

- **oflow**($p_0>p_1>p_2> \ldots>p_n>p_e$), which we call Prop. A, holds if only if $\mathbf{dom}\left(\tau_{p_0}(p_1)\right) \supseteq \mathbf{outhd}(p_o)$ and $0 < \forall i < n.\mathbf{dom}\left(\tau_{p_{i+1}}(p_{i+2})\right) \supseteq \tau_{p_{i-1}}(p_i) \circ \cdots \circ \tau_{p_1}(p_2)(\mathbf{outhd}(p_o))$, which we call Prop. B, holds, where $p_e = p_{n+1}$ and **dom**(*f*) is the domain of Function *f*.
  - We firstly prove that Prop. A → Prop. B. From Prop. A, this flow goes through p_{i+1} and p_{i+2} for any *i*. Then, $\forall i.\mathbf{dom}\left(\tau_{p_{i+1}}(p_{i+2})\right) \supseteq h_{p_{i+1}}$ holds. Notice that we can calculate Header Space h_{p_i} as $h_{p_{i+1}} = h_{p_i} = \tau_{p_{i-1}}(p_i) \circ \cdots \circ \tau_{p_1}(p_2)(\mathbf{outhd}(p_o))$. Then Prop. A holds.
  - Finally, we show Prop. B → Porp. A. For any *i*, the flow with Header Space $h_{p_i} = h_{p_{i+1}}$ goes through p_{i+1} and p_{i+2} because $\forall i.\mathbf{dom}\left(\tau_{p_{i+1}}(p_{i+2})\right) \supseteq h_{p_{i+1}}$ from Prop. B. Then, the flow with $\mathbf{outhd}(p_o)$ goes through $p_0>p_1>p_2> \ldots>p_n>p_e$. Then Prop. A holds if Prop. B holds.
- **iflow**($p_0<p_1<p_2<\ldots<p_n<p_e$) holds if only $0 < \forall i < n.\mathbf{dom}\left({\tau^{-1}}_{p_{i+1}}(p_{i+2})\right) \supseteq {\tau^{-1}}_{p_{i-1}}(p_i) \circ \cdots \circ {\tau^{-1}}_{p_1}(p_2)(\mathbf{inhd}(p_o)$. ${\tau^{-1}}_{p_{i+1}}(p_i)$ is the inverse function of $\tau_{p_i}(p_{i+1})$. In short, ${\tau^{-1}}_{p_{i+1}}(p_i) \circ \tau_{p_i}(p_{i+1})\left(h_{p_i}\right) = h_{p_i}$ and $\tau_{p_i}(p_{i+1}) \circ {\tau^{-1}}_{p_{i+1}}(p_i)\left(h_{p_{i+1}}\right) = h_{p_{i+1}}$.
- We propose two implementation designs of **outhd** and **inhd**. Firstly we implement them as B-HAS functions. [Design 1]. In short, **outhd**(p) ::= $\beta_{out}(p)$ and **inhd**(p) ::= $\beta_{in}(p)$. We calculate the header spaces of all ports in the network before the policies are checked.

- Second one uses HSA transfer function [Design 2]. **outhd**(p) ::= $\bigcup_{q \in \mathbf{out_paths}(p)} \tau_{q(i-1)}(q(i)) \circ \cdots \circ \tau_{q(0)}(q(1))(\mathbf{outhd}(p))$, where **out_paths**$(p)$ is the all flows from Port p, and $q(i)$ is i-th link of the flow q. **inhd**(p) ::= $\bigcup_{q \in \mathbf{in_paths}(p)} \tau^{-1}{}_{q(i-1)}(q(i)) \circ \cdots \circ \tau_{q(0)}(q(1))(\mathbf{inhd}(p))$, where **in_paths**$(p)$ is the all flows to Port p.

F. Performance Estimations of Design 1 and Design 2

We estimated the verification times of these two methods. In Fig.5 and Fig. 6 in our previous work [4], we estimated approximate curves of HSA and B-HSA The x-axis of these curves is the number of the flow entries in the network. The y-axis is the calculation time of the header spaces of the all ports in the network.

- B-HSA $b(\mathrm{x}) = 3 * 10^{-1}x^2 + 0.077x - 13.267$ [sec.]
- HSA $h(\mathrm{x}) = 3 * 10^{-8}x^3 - 0.0001x^2 + 0.2837x - 45.2333$ [sec.]

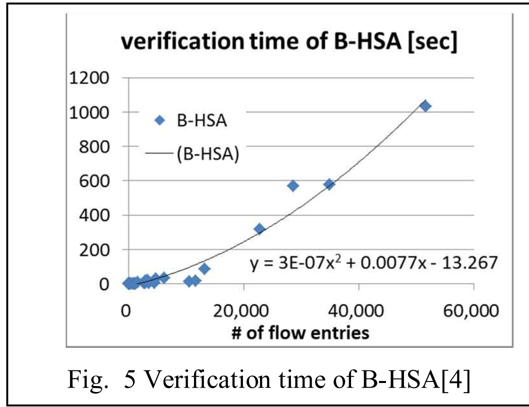

Fig. 5 Verification time of B-HSA[4]

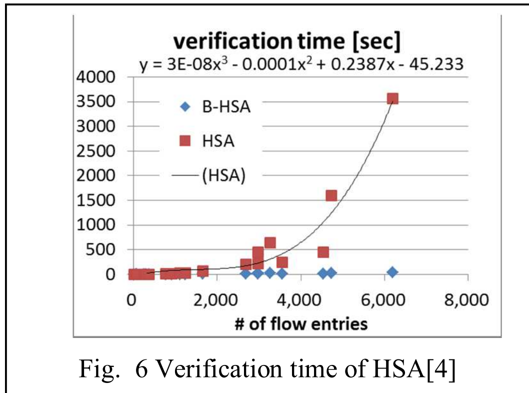

Fig. 6 Verification time of HSA[4]

In this estimation, the network is assumed to consist of s switches, each of which has 24 ports and 500 flow entries.

- In Design 1, it takes $b(500s)$ [sec] for calculation of all header spaces. The calculation time of **iflow** and **oflow** are estimated small in comparison with $b(500s)$. We approximate the verification time as $b(500s)$.
- In Design 2, the calculation time of **inhd**(p) and **outhd**(p) is estimated as $h(500s)/(24s)$ [sec] because the network has $24s$ ports and the calculation time of only one port is estimated $h(500s)/(24s)$.

Fig. 7 shows graphs of the estimated verification times of Design 1 (B-HSA) and Design 2 (HSA) under the 50 servers and the 100 servers. The x-axis shows the number of ports used in a policy to be verified. The y-axis is logarithmic of the verification times of B-HSA and HSA. The verification times are not so different when the ports in the policy are a few, but that of Design 2 is getting rapidly worse when many ports are verified.

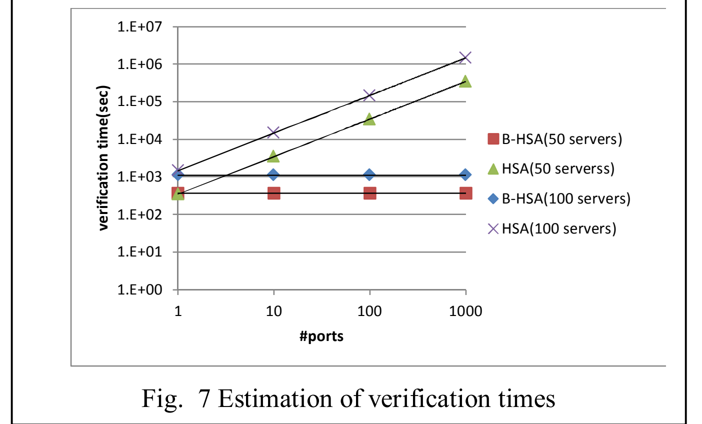

Fig. 7 Estimation of verification times

IV. Conclusion

In this paper, we proposed a policy language and the implementation designs of a policy verification tool used in the network construction time. The policy is estimated to be able to verify a large network. In future, we will implement the tool, and evaluate the usefulness and performance of the proposed methods.

References

[1] Peyman Kazemian, George Varghese, and Nick McKeown. "Header space analysis: static checking for networks". In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12). USENIX Association,

[2] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte,. "Real time network policy checking using header space analysis". In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), pages 99-111, 2013.

[3] A. Khurshid, W. Zhou, M. Caesar, P. B. Godfrey, " VeriFlow: Verifying Network-Wide Invariants in Real Time," ACM SIGCOMM Computer Communication Review, Vol. 42, No. 4, pp. 467-472, Sep., 2012.

[4] Toshio Tonouchi, Satoshi Yamazaki, Yutaka Yakuwa and Nobuyuki Tomizawa, "A fast method of verifying network routing with back-trace header space analysis, IM2015

[5] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," in Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011 (M. M. T. Chakravarty, Z. Hu, and O. Danvy, eds.), pp. 279{291, ACM, 2011.

[6] "NOXRepo." http://www.noxrepo.org/.

[7] "About POX | NOXRepo." http://www.noxrepo.org/pox/about-pox/

[8] "Floodlight Project." http://www.projectfloodlight.org/projects/.

[9] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D.Walker, "Netkat: Semantic foundations for networks," in Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, (New York, NY, USA), pp. 113-126, ACM, 2014.