Towards an Efficient Verification Approach on Network Configuration

Khalid Elbadawi
School of Computing
DePaul University
Chicago, IL 60604
Email: badawi@cdm.depaul.edu

Yongning Tang
School of Information Technology
Illinois State University
Normal, IL 61790
Email: ytang@ilstu.edu

James Yu
School of Computing
DePaul University
Chicago, IL 60604
Email: jyu@cdm.depaul.edu

***Abstract*—This paper presents our new design and implementation of a configuration verification system called ConfVS. With the increasing complexity of network configuration, verifying network behavior has become a highly time-consuming and error-prone process. Much research effort has been made to tackle this challenge. In this paper, we propose a formalization scheme based on binary decision diagram to model the entire network behavior specified by diverse configuration requirements (e.g., security policies, routing policies, and address translation rules), and design a set of algorithms to efficiently verify the compliance of network behavior to the requirements. Our experiments show that ConfVS can validate thousands of network devices configured by millions rules with ten times improved efficiency when compared to several well-known existing solutions.**

I. Introduction

Modern networks are designed and deployed to satisfy a wide variety of competing goals related to different network requirements, such as security vs. availability, and performance vs. manageability [1]. These high–level goals are realized through a complex chain of configuration commands that may include the configuration of thousands of access control rules from different policies such as packet filtering rules, routing policies, and address translation. Maintaining high–level goals is difficult because configurations are continuously revised due to policy changes and new devices and services.

Several surveys show that misconfiguration is the most critical threat to network operations [2], [3]. Analysis from IT experts states that human error is responsible for 50 to 80 percent of network outages [4]. In addition, a local configuration change or network failure may even inadvertently result in a global impact on existing network services. For example, when a network link goes down, the network may converge into a new topology that has not been envisioned by network administrators. This may indirectly cause illegitimate traffic bypass security protection or legitimate packets accidentally blocked. Thus, timely configuration verification is a highly desirable network service to enhance network reliability and manageability.

It is a significant challenge to verify in a timely fashion the configurations of a network system which may consist of hundreds or thousands of different devices with possibly hundreds of configuration rules on each device [5]. There is a growing consensus on the need of formal models to conduct configuration verification in order to validate enforced network policies. As such, many researchers have proposed formal models of network security and reachability. The work in [6]–[8] has focused on verifying firewall policies, while the works in [9], [10] have focused on detecting routing misconfiguration. Global verification has been discussed in [11]–[15].

This paper presents a configuration verification system called ConfVS, which provides instant configuration diagnoses for reachability and security compliance. ConfVS models the whole network as a directed graph, where each vertex in the graph represents a unidirectional *connection* between two adjacent devices. This abstraction allows us to build efficient algorithms to verify end-to-end reachability and security policies. We formalize the behavior of each policy rule using an efficient Boolean expression represented in Binary Decision Diagram (BDD).

Configuration verification should cover both *consistency* verification and *behavioral* verification. Consistency verification is related to the correctness of configuration data, while behavioral verification is about the dynamics of IP packets passing through a network where the packets are forwarded, blocked, delayed, or logged. This paper is considering only behavioral verification in terms of reachability and security compliance. Consistency verification has been addressed in our previous work [16].

This paper is structured as follows. We first describe our new formalization for firewall, routing and NAT policies in Section II. Then, we present an algorithm to compute network reachability in Section III. Section IV discusses our evaluation framework and results. Finally, we conclude our work in Section V.

II. Policy Formalization

This section presents a new set of formalization schemes for network verification analysis of security and reachability. One common function on most network devices is data forwarding, from an incoming interface to an outgoing interface based on specific *Forwarding Control List* (FCL). For example, FCL on a router is its routing entries in the routing table. We focus on three types of FCLs: filtering FCL, routing FCL and Network Address Translation (NAT) FCL. The combined effect of these FCLs will determine flow manipulation and network behavior.

Our objective is to study the effect (i.e. the actions) of these FCLs on IP packets passing through the devices

We use BDD to represent FCLs since BDD models a complex *set* as a single Boolean expression. Let us denote the Boolean expression of a set X as F_X, which is generated from a number of conjunctive Boolean variables $t_i \in \mathbb{B}$, $i = 1 \cdots n$ where $\mathbb{B} \in \{0, 1\}$. For example, if $X = \{5\}$, which is a set of a single element, then $F_X = t_1 \wedge \neg t_2 \wedge t_3$. The only assignment that makes F_X true is $1, 0, 1$ for t_1, t_2, and t_3, respectively. In addition, If X and Y are two different sets, then $X \cup Y$ and $X \cap Y$ can be modeled as $F_X \vee F_Y$ and $F_X \wedge F_Y$, respectively, in BDD.

Now, let $\mathcal{A}$ be an FCL that contains a set of rules, where each rule r has a constraint set c over certain IP packet header fields along with an action a to be taken when a network packet matches the constraints in c. The packet header fields include IP protocol, destination address, source address, destination port, and source port. Note that the constraint c represents a set of IP packets and we define this set as a *flow* (denoted as π).

The Boolean expression of an FCL depends on its *type*. However, the Boolean expression of an FCL rule is unified and generated as follows. Let $r = \langle c_{proto}, c_{dstip}, c_{srcip}, c_{dstport}, c_{srcport}, a\rangle$ be a rule where c_x is a constraint over the field x. The Boolean expression of the rule r, denoted as F_r, is $F_a \wedge F_c$ where $F_c = F_{proto} \wedge F_{dstip} \wedge F_{srcip} \wedge F_{dstport} \wedge F_{srcport}$. The Boolean expression of F_a depends on the FCL type.

We introduce three functions to manipulate Boolean expressions, which are the *projection* function, its *inverse* function, and the *mask* function. The *projection* function is defined as $\Psi : F_\pi \times P \rightarrow F_a$, where F_π, P, and F_a are Boolean expressions for a flow, an FCL policy and an action, respectively. Ψ is used to determine the set of actions applied to a flow π. The *inverse* function, which is defined as $\Psi^{-1} : F_a \times P \rightarrow F_\pi$, is used to extract the flow given a set of actions. For example, if P is a firewall policy, $\Psi^{-1}(false, P)$ returns a Boolean expression that represents the set of denied packets (since we model the actions *accept* and *deny* as true and false, respectively).

The *mask* function is defined as $\tau : F \times h \rightarrow F'$ that takes two Boolean expressions F and h as arguments, and returns a new Boolean expression F' by resetting F at location defined in h using an existential quantifier operator. For example, if a flow is any packet from 10.1.2.0/24 to 10.1.23.0/24, applying the τ function of $h = F_{dstip} = 1$ to the flow will create a new flow of any packet from 10.1.2.0/24 to any destination. In the following, we show how to construct the formal expressions for a filtering policy, a routing policy, and a NAT policy.

1) Firewall policy construction: The Boolean expression of a firewall rule represents the set of packets that can *pass* through the associated interface. Constructing a firewall FCL is very crucial since it requires all IP header information of an IP packet. To build its BDD expression with N rules, we use the following recursive equation:

$$T(i) = \begin{cases} a_i \rightarrow T(i+1) \vee F_r^i, & T(i+1) \wedge \neg F_r^i \\ & \text{if } 1 \leq i < N \\ a_i \rightarrow F_r^i,\ false & i = N \end{cases} \tag{1}$$

where $a \rightarrow b, c$ represents if–then–else operator (If a is true, then b, otherwise c). The Boolean expression, say P, is then obtained by computing $T(1)$, and its effect on a flow π can be expressed as

$$P \wedge F_\pi \tag{2}$$

Recall that $P \wedge F_\pi$ is equivalent to '*the set of accepted packets* $\cap\ \pi$'.

2) Routing Policy construction: Unlike firewalls, routers do not select next hop based on the order of routing rules, but would rather find the longest prefix match. In addition, most routers support load-balancing when multiple paths exist. This implies multiple routing rules can be applied to the same flow with their corresponding actions.

Each routing rule in the routing FCL is composed of a single matching field, which is the destination IP address, along with an action determined by the outgoing interface and the IP address of the next hop. Therefore, we construct the Boolean expression of a routing policy as follows. For each network prefix of length m, we first construct P_m and P_m^c such that

$$P_m = \bigvee_{k \in I_m} F_r^k \text{ and } P_m^c = \bigvee_{k \in I_m} F_c^k$$

where I_m represents the set of indexes for rules that have destination prefix of length m and F_c^k represents the Boolean expression of the rule constraint without the action field. Recall that $F_r^k = F_c^k \wedge F_a^k$. Thus, $F_c^k = F_{dstip}^k$ and $F_a^k = F_n^k$ where n is the outgoing interface number of the k^{th} rule. After that, we construct the routing policy, P, using the following recursive equation:

$$T_m = P_m \vee (\neg P_m^c \wedge T_{m-1}) \quad 1 \leq m \leq 32, \tag{3}$$

where $P = T_{32}$ and $T_0 = P_0$ represents the default gateway rule.

For routing verification, we would like to focus on whether or not a flow π can reach a specific device. In other words, if flow π passes through a router device, what flow will be routed to the outgoing interface n? We answer this query using the following expression:

$$F_\pi \wedge \Psi^{-1}(F_n, P) \tag{4}$$

3) NAT policy construction: Network Address Translation (NAT) is used by a device (e.g., firewall or router) deployed between a private and public network. There are three forms of NAT FCLs: static NAT FCL, dynamic NAT FCL, and overloading FAT (also known as PAT) FCL. In static NAT FCL, each rule translates a given source IP address to another pre-allocated source IP address. In dynamic NAT FCL, each rule translates a group of source IP addresses to another pre-allocated group of IP addresses. In overloading NAT FCL,

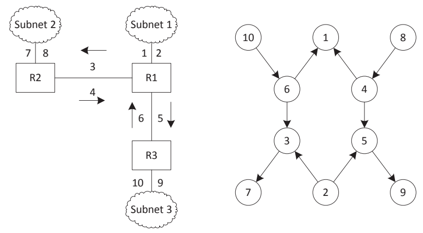

Fig. 1. A Simple network topology and its equivalent in graph model

each rule translates a group of source IP addresses to a single pre-allocated IP address but with different source ports.

In our analysis, we do not differentiate between static and dynamic NAT policies. Because Boolean expressions can represent a single value or a set of values, both NAT FCLs have the same semantic.

Basically, NAT FCL is composed of a constraint over the source header field of an IP packet along with an action that determines the new source address. If a flow that is traversing from a private to public network matches the constraint, the source IP address of each packet in the flow will be translated. Otherwise, the flow will be *passed* without translation. The device that supports NAT service also keeps a reverse translation for the flow that is traversing from public to private network. Modeling NAT FCL requires extra bits (up to 48 bits) to express the rule's action. This may explode in size for certain expressions. To resolve this problem, we break an FCL rule ($c \rightsquigarrow a$) into two rules: $c \rightsquigarrow i$ and $a \rightsquigarrow i$ where i is the rule index. This solution is valid because the number of rules in NAT FCL is very small. In our implementation, we set the maximum number of rules per FCL to 256 rules.

Let $\mathcal{A}_c$ be the FCL that represents $c \to i$ and $\mathcal{A}_a$ be the FCL that represents $a \to i$. Let P_c and P_a represent the Boolean expressions for $\mathcal{A}_c$ and $\mathcal{A}_a$, respectively. We calculate P_c and P_a as:

$$P_c = \bigvee_{i}^{K} F_i \wedge F_c^i \quad \text{and} \quad P_a = \bigvee_{i}^{K} F_i \wedge F_a^i,$$

where K is the number of rules. Now, let π be a flow that traverses from private to public network and passes through a static or dynamic NAT service. Let $P = \Psi(true, P_c)$ be the set of all packets that will be translated. The effect of NAT service on the flow is expressed as:

$$T(P \wedge F_\pi) \vee (\neg P \wedge F_\pi), \tag{5}$$

where $\neg P \wedge F_\pi$ is the set of packets that passes NAT service without translation, and $P_c \wedge F_\pi$ is the set of packets that has been translated under the transformation function T, and $T(x) = \tau(x, F_{srcip}) \wedge \Psi^{-1}(\Psi(x, P_c), P_a)$. In case of PAT, we replace F_{srcip} with the expression $F_{srcip} \wedge F_{srcport}$.

While constructing the Boolean expressions of P_c and P_a, we also construct $\acute{P_c}$ and $\acute{P_a}$ to represent the reverse translation when a flow is traversing from public back to private network. A constraint over the source IP address encoded in P_c and P_a becomes the constraint over the destination IP address encoded in $\acute{P_c}$ and $\acute{P_a}$, respectively. The inverse transformation function is defined as $T^{-1}(x) = \tau(x, F^{dstip}) \wedge \Psi^{-1}(\Psi(x, \acute{P_a}), \acute{P_c})$.

Algorithm 1 Updating Connection Expression

```
Input: Dev1, Inf1, Dev2, Inf2
Output: $F_\pi$
1: $F_\pi \leftarrow true$ /* i.e. any possible flow */
   /* processing outgoing interface */
2: if Dev1 has Routing then $F_\pi \leftarrow$ Apply Eqn 4
3: if Dev1 has NAT and Inf1 is external then $F_\pi \leftarrow$ Apply Eqn 5
4: if Dev1 has Filtering at Inf1 then $F_\pi \leftarrow$ Apply Eqn 2
   /* processing incoming interface */
5: if Dev2 has Filtering at Inf2 then $F_\pi \leftarrow$ Apply Eqn 2
6: if Dev2 has NAT and Inf2 is internal then $F_\pi \leftarrow$ Apply Eqn 5
7: return $F_\pi$
```

Algorithm 2 Computing $\mathcal{G}$ of vertex $v \in V$

```
Input: The graph $G = \langle V, E \rangle$ and vertex $v \in V$
1: reset all vertices in $V$ that are involved in a cycle as not visited
2: $\mathcal{G}_v$ = compute(v)
Procedure: compute($v$)
1: if $v$ is not visited then
2:     mark $v$ as visited
3:     if $v \in S$ then
4:         $\mathcal{G}_v \leftarrow \mathcal{F}_v$
5:     else
6:         $X \leftarrow$ false
7:         for all $u \in V$ such that $[u, v] \in E$ do
8:             compute($u$)
9:             $X \leftarrow X \vee \mathcal{G}_u$
10:        end for
11:        $\mathcal{G}_v \leftarrow \mathcal{F}_v \wedge X$
12:    end if
13: end if
```

III. Network Behavior Verification

We model a network topology as a directed graph $G = \langle V, E \rangle$, where V is the set of vertices and E is the set of edges. We also model each link in the network topology as one or two unidirectional *connection(s)* based on the link type, and assign each connection a universal identification number (id). These connections constitute the vertices of graph G. Therefore, a vertex $v \in V$ represents a unidirectional connection, and is labelled by its connection id. A directed edge $[u, v]$ in E is constructed when both vertices u and v are sharing a common device d such that the device d has an incoming interface from connection u and an outgoing interface to connection v. Consequently, the set V is divided into three disjoint sets: source vertices S, sink vertices T and intermediate vertices M. Note that S and T represent end-to-end connections in a network topology. Therefore, for each end host there is at least one connection v in S, one connection u in T, and $v \neq u$. Figure 1 shows a network topology and its representation in our graph model. The numbers beside links represent connection IDs. For the horizontal link, the upper ID has a direction from right to left, while the lower ID has a direction from left to right. For each vertical link, the ID on right side has a direction from up to bottom, while the ID on the left side has a direction from bottom to up.

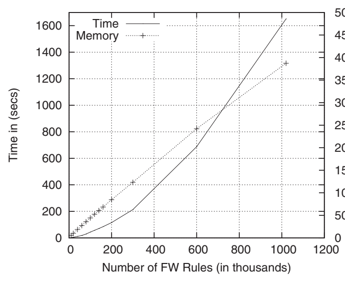

Fig. 2. Time/Memory required to construct FW BDDs

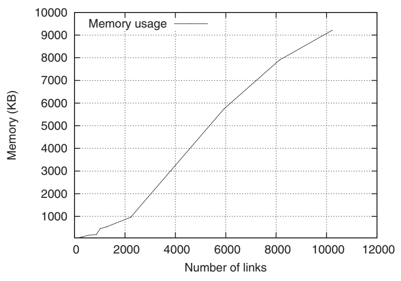

Fig. 3. Memory required to construct graph model

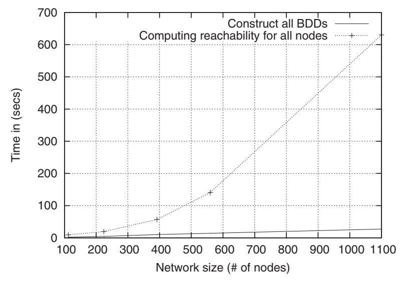

Fig. 4. ConfVS performance in terms of time

For each connection, we invoke Algorithm 1 to compute its Boolean expression $\mathcal{F}$. We follow the order of operations based on specific network devices.

Let $\mathcal{G}_v$ be a Boolean expression that represents the aggregated effect when a flow traverses from source connections towards vertex $v \in V$. We can obtain $\mathcal{G}_v$ by computing the disjunction of all paths from the source vertices to the vertex v, in which each path is computed by the conjunction of all connection's Boolean expressions that construct the path. For example, $\mathcal{G}_6 = \mathcal{F}_6 \wedge \mathcal{F}_{10}$ and $\mathcal{G}_3 = \mathcal{F}_3 \wedge (\mathcal{G}_2 \vee \mathcal{G}_6)$. Algorithm 2 describes a depth first search algorithm to compute $\mathcal{G}_v$ where $v \in V$. Using this algorithm, we can find, for example, what type of packets can reach S2 starting from S3. This query can be expressed as

$$\mathcal{G}_{10} \wedge \mathcal{G}_7.$$

The running time of Algorithm 2 is $O(|V|)$, ignoring the cost of BDD operations.

Computing $\mathcal{G}$ for all nodes requires applying Algorithm 2 for each sink vertex. The algorithm ensures that $\mathcal{G}_v$ is computed only once unless v is included in a cycle. In the case of cyclic graph, we reset those vertices that are included in some cycle by marking them as unvisited whenever we invoke `compute()`. Ignoring the cost of BDD operations, the running time over all sink vertices using Algorithm 2 is $O(|V| \times |T|)$. Note that identifying loops in a graph can be achieved in a linear time.

IV. Implementation and Evaluation

We implemented our model using the Erlang language along with C and C++ languages for connecting Erlang system with BDD package. Erlang [17] supports in-memory high-performance database to store millions of ConfVS predicates [18]. For BDD, we model FCL constraint using 104 Boolean variables and FCL action using 8 Boolean variables. We implemented BDD expressions using BuDDy package. In BuDDy, the BDD operation running time is linear to the size of the resulted BDD expression.

We evaluate the performance of our model in terms of memory and time complexities. We use the network topology generator tool introduced in [13] to generate a random network topology along with a set of random configuration files for routers, NATs and firewalls.

We run our evaluation on a computer with Intel Core 2 Duo CPU 2.13GHz and 4GB of RAM. We generated 15 different scenarios with different network sizes. Five scenarios have been configured to generate huge set of complex firewall policies. The remaining scenarios are used to evaluate ConfVS. The average number of nodes is 600 with 10,000 links. The average total number of FCL rules is 100,000 rules per network.

We first show the performance of using Equation 1. Firewall policy is considered the most complex and costly in BDD construction than other policies since it involves all packet header information. Figure 2 illustrates the time and memory required to build firewall rules. The time complexity increases quadratically (yet close to linear) while space complexity increases linearly. Note that our formalization scheme takes only 115 seconds to process 200,000 rules, while the previois work presented in [13] requires 1,100 seconds. Also, our scheme is scalable to construct more than one million firewall rules without exploding the BDD memory size.

Figure 3 shows the impact of the number of links in a network on the space complexity of the ConfVS system (excluding the space complexity in BDD package). We can see that the space required to build the graph model is almost linear.

The running time of our proposed algorithms is depicted in Figure 4. As shown, the running time to build all connection's BDDs increases linearly while the running time to build all $\mathcal{G}$'s expressions increases quadratically. As discussed before, the running time to compute reachability for all sink vertices is $O(n^2)$. For a network of size 1,100 nodes, the overall building time is approximately 618 seconds.

V. Conclusion

We present a new formalization scheme, along with its implementation in ConfVS, to address the issue of network configuraiton verification in a large and complex network environment where its behavior is constantly changing. ConfVS is a graph-based model and uses Binary decision diagram (BDD) to formally capture the network behavior. Our experiment results show that ConfVS can handle thousands of FCL rules more efficiently than the previous work. Moreover, ConfVS performs incremental computation to provide an instant answer to the compliance of all verification rules.

REFERENCES

[1] D. K. Smetters and R. E. Grinter, “Moving from the design of usable security technologies to the design of useful secure applications,” in *Proceedings of the 2002 workshop on New security paradigms*, ser. NSPW ’02. New York, NY, USA: ACM, 2002, pp. 82–89.

[2] Z. Kerravala, “As the value of enterprise networks escalates, so does the need for configuration management,” *The Yankee Group*, Jan. 2004.

[3] Arbor Networks Inc., “Worldwide infrastructure security report, volume vi,” http://www.arbornetworks.com/report, 2010.

[4] What’s Behind Network Downtime?, “Whitepaper by juniper networks,” http://www-05.ibm.com/uk/juniper/pdf/200249.pdf, May 2008.

[5] R. Bush, O. Maennel, M. Roughan, and S. Uhlig, “Internet optometry: assessing the broken glasses in internet reachability,” in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, ser. IMC ’09. New York, NY, USA: ACM, 2009, pp. 242–253.

[6] H. Hamed, E. Al-Shaer, and W. Marrero, “Modeling and verification of ipsec and vpn security policies,” in *Proceedings of the 13TH IEEE International Conference on Network Protocols (ICNP’05)*, Washington, DC, USA, 2005, pp. 259–278.

[7] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, “Fireman: a toolkit for firewall modeling and analysis,” in *Security and Privacy, 2006 IEEE Symposium on*, may 2006.

[8] M. G. Gouda and A. X. Liu, “Structured firewall design,” *Comput. Netw.*, vol. 51, pp. 1106–1120, March 2007.

[9] N. Feamster and H. Balakrishnan, “Detecting bgp configuration faults with static analysis,” in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI’05. USENIX Association, 2005, pp. 43–56.

[10] F. Wang, J. Qiu, L. Gao, and J. Wang, “On understanding transient interdomain routing failures,” *IEEE/ACM Trans. Netw.*, vol. 17, no. 3, pp. 740–751, Jun. 2009.

[11] G. Xie *et al.*, “On static reachability analysis of ip networks,” in *IEEE INFOCOM*, vol. 3, 2005, pp. 2170–2183.

[12] A. R. Khakpour and A. X. Liu, “Quantifying and querying network reachability,” *Distributed Computing Systems, International Conference on*, vol. 0, pp. 817–826, 2010.

[13] E. Al-Shaer, W. Marrero, A. El-Atawy, and K. Elbadawi, “Network configuration in a box: towards end-to-end verification of network reachability and security,” in *Proceedings of 17th International Conference on Network Protocols (ICNP)*. IEEE, Oct. 2009, pp. 123–132.

[14] S. Narain, G. Levin, S. Malik, and V. Kaul, “Declarative infrastructure configuration synthesis and debugging,” *J. Netw. Syst. Manage.*, vol. 16, no. 3, pp. 235–258, Sep. 2008.

[15] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, “Debugging the data plane with anteater,” in *Proceedings of the ACM SIGCOMM 2011 conference*, ser. SIGCOMM ’11. New York, NY, USA: ACM, 2011, pp. 290–301.

[16] K. Elbadawi and J. Yu, “High level abstraction modeling for network configuration validation,” in *GLOBECOM 2010, 2010 IEEE Global Telecommunications Conferenc*, Dec. 2010.

[17] J. Armstrong, R. Virding, and M. Williams, *Concurrent Programming in Erlang*, 2nd ed. Prentice Hall International (UK), 1996.

[18] S. L. Fritchie, “A study of erlang ets table implementations and performance,” in *Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, ser. ERLANG ’03. New York, NY, USA: ACM, 2003, pp. 43–55.