Build and Test Your Own Network Configuration

Saeed Al-Haj, Padmalochan Bera, and Ehab Al-Shaer

University of North Carolina Charlotte, Charlotte NC 28223, USA
{salhaj,bpadmalo,ealshaer}@uncc.edu

Abstract. Access control policies play a critical role in the security of enterprise networks deployed with variety of policy-based devices (e.g., routers, firewalls, and IPSec). Usually, the security policies are configured in the network devices in a distributed fashion through sets of access control lists (ACL). However, the increasing complexity of access control configurations due to larger networks and longer policies makes configuration errors inevitable. Incorrect policy configuration makes the network vulnerable to different attacks and security breaches. In this paper, we present an imperative framework, namely, *ConfigLEGO*, that provides an open programming platform for building the network security configuration globally and analyzing it systematically. The *ConfigLEGO* engine uses Binary Decision Diagram (BDD) to build a Boolean model that represents the global system behaviors including all possible interaction between various components in extensible and scalable manner. Our tool also provides a C/C++ API as a software wrapper on top of the BDD engine to allow users in defining topology, configurations, and reachability, and then analyzing in various abstraction levels, without requiring knowledge of BDD representation or operations.

Keywords: Imperative analysis, BDDs, Formal methods, Network configuration.

1 Introduction

The extensive use of various network services and applications (e.g., telnet, ssh, http, etc.) for accessing network resources forces enterprise networks to deploy policy based security configurations. However, most of the enterprise networks face security threats due to incorrect policy configurations. Recent studies reveal that more than 62% of network failures today are due to security misconfiguration. These misconfigurations may cause major network failures such as reachability problems, security violations, and introducing vulnerabilities. An enterprise LAN consists of a set of network *domains* connected through various interface routers. The security policies of such networks are configured in the security devices (like, routers, firewalls, IPSec, etc.) through set of access control lists (ACLs) in a distributed manner. The global network configuration may contain several types of conflicts (redundancy, shadowing, spuriousness, etc.) in different levels (intra-policy, inter-policy) [1] which may violate the end-to-end

M. Rajarajan et al. (Eds.): SecureComm 2011, LNICST 96, pp. 522–532, 2012.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2012

security of the network. Moreover, there may exist several reachability problems depending on flow-, domain-, and network-level constraints. Thus, the major challenge to the network administrators/developers is to comprehensively build and analyze the security configurations in a flexible and efficient manner.

In this paper, we present an imperative framework, namely, *ConfigLEGO* that allows users to comprehensively specify, implement, and diagnose network configurations based on user requirements. *ConfigLEGO* internally uses an efficient binary decision diagram (BDD) structure to compactly represent the network configuration and further uses the configuration BDDs for analysis. On the other hand, *ConfigLEGO* provides a C/C++ programming interface that acts as a software wrapper on top of the BDD engine to selectively compose different BDDs for systematic evaluation. *ConfigLEGO* is named after the famous Lego toy. In a Lego toy, one can build a complex design from a set of basic components. *ConfigLEGO* provides all basic components to design a network, and allows the user to build and test his own network by putting all components together and by writing the queries he needs to check the validity of configuration properties. *ConfigLEGO* hides BDDs complexity and allows users to analyze the network without requiring previous knowledge about BDD representation or operations.

Compared to other declarative modeling languages/systems, *ConfigLEGO* is the first BDD-based engine that provides a generic C/C++ programming interface for configuration modeling, abstraction, and analysis in usable and scalable manner. Compared to other network management tools like COOLAID [2], *ConfigLEGO* provides libraries for comprehensively creating and analyzing network configurations based on user requirements.

In terms of diagnosability, *ConfigLEGO* can evaluate various configuration problems such as follows:

- Network reachability, intra- and inter-firewall misconfigurations, flow-level, path-level and domain-level network traffic reachability.
- Inferring configuration problems using a sequence of evaluation results.
- Analyzing network configuration and testing whether the current configuration meets the provided risk requirements.

The remaining sections of the paper are organized as follows: Section 2 describes the architecture of *ConfigLEGO*; Section 3 introduces different examples verified using *ConfigLEGO*. This section also describes the formalization of the examples in Boolean logic using efficient BDD representation; Section 4 discusses the evaluation and experimental results; Section 5 is the related work; Finally, conclusion and future work have been presented in Section 6.

2 Architecture of ConfigLEGO

The presented *ConfigLEGO* system (refer Figure 1) consists of two major modules: Internal Module and User Program Module. The Internal Module is the core of the system which has two components; *ConfigLEGO* Engine and the *ConfigLEGO* API. *ConfigLEGO* Engine is responsible for modeling the device

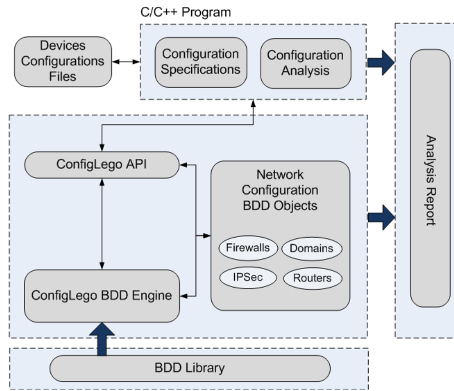

Fig. 1. *ConfigLEGO* System Architecture

configuration (behavior of each device such as, domains, routers, firewalls etc.), topology, and the network using efficient BDD representation. The engine builds a BDD for each device and each path in the network. It can also provide a BDD for the entire network depending on user requirement. A complete set of functions for managing internal module is defined in *ConfigLEGO* API. It provides an interface to the User Program Module to use *ConfigLEGO* system.

The user only needs a basic knowledge of the available functions in the API to write his/her program for analyzing/diagnosing a network.

2.1 ConfigLEGO Internal Module

ConfigLEGO parses the device configuration files and builds a BDD structure for each device. Then, it allows to define the links between the device BDDs and different access paths depending on the network topology specification. This network configuration can be formalized as a network access model.

Definition 1 [Network Access Model]: A network access model is defined as a 3-tuple $NG = \langle N, I, F \rangle$, where,

- N is a finite set of network devices. Network devices (N) can be of three types: NR- network routers; NF- firewalls and NE- end point devices (hosts/domains) represented as an IP address block. Each device is associated to several connecting interfaces which are identified by an IP address.
- $I \subseteq N \times N$ is a finite set of network links between devices, such that for every physical link between N_1 and N_2 there is a pair of lines or channels: $I_{12} = \langle N_1, N_2 \rangle$ and $I_{21} = \langle N_2, N_1 \rangle$.
- F is a finite set of access control lists (ACL) associated to different devices.

Modeling Network Configuration. Each device configuration in the network has to be modeled in BDD for modeling the network configuration. More details

about modeling devices configuration can be found in [3]. After modeling devices configuration, we model the complete network considering the topology and the combined effect of the routing and firewall rules. *ConfigLEGO* allows users to specify the topology using C/C++ programming constructs. It is represented as a formal network access model, $NG\langle N, I, F\rangle$ as described in the last section. Then, we formalize the combined effect of firewall rules along different access routes between source and destination. In this process, the notion of *Access Routes* and *Access Route Policy* have been introduced.

Definition 2 [Access Route]: An *Access Route* $AR_i^{(S,D)}$ is defined as a sequence of devices $(N_1, N_2, ..., N_k)$ from source S to destination D in the network where, each $\langle N_i, N_{i+1}\rangle \in I$ and D is reachable from S through $S, N_1, N_2, ..., D$. This corresponds to the physical topology of the network.

Definition 3 [Access Route Policy]: An *Access Route Policy* ($ARP^{(S,D)}$) between a source S and destination D is a combined model of the distributed policy rules along all possible access routes $(AR_1^{(S,D)}, ..., AR_n^{(S,D)})$ between a source S and a destination D. It is represented as a Boolean function:

$$ARP^{(S,D)} = (P_{AR_1^{(S,D)}} \vee P_{AR_2^{(S,D)}} \vee \ldots \vee P_{AR_n^{(S,D)}})$$

such that $P_{AR_i^{(S,D)}} = \bigwedge_{N \in AR_i} P_a^N$. This represents the logical access path.

Here, $P_{AR_i^{(S,D)}}$ (along the route AR_i) is represented as the conjunction of policies for all devices in that route. Then, we represent the complete access route policy $ARP^{(S,D)}$ between a source S and a destination D as disjunction of all $P_{AR_i^{(S,D)}}$ for access route AR_i. We describe the modeling of logical access paths between a specified pair of source and destination. However, depending on user requirement, *ConfigLEGO* is also capable of generating the combined network model considering all possible source and destination pairs in the network. For basic reachability analysis, the presented *ConfigLEGO* framework checks the conjunction of the BDDs along the access route as specified. On the other hand, for imperative analysis, it uses the sequence of reachability results. Section 3 shows the verification of such analysis with different examples.

ConfigLEGO Engine in Figure 1 utilizes the compact canonical format that BDDs provide to encode the device configuration file into a BDD representation. This will be used later by the *ConfigLEGO* API for providing a convenient interface to the user writing his own program. A partial set of the functions provided by *ConfigLEGO* API are shown in Table 1. The functions support three phases of network's design: (1) Components Installation, (2) Components Connection, and (3) Testing and Validation.

2.2 User Program Module

In this module, the user can provide network specifications in C/C++ programming language. The *ConfigLEGO* API supports user's program module by providing a set of functions that will be used by a user to construct a network. For

Table 1. ConfigLEGO API Functions

Definitions and Function Names	**Description**
Network N	To create a new network N
Firewall F("policy.txt")	creates a firewall that has policy defined in *policy.txt* text file
Router R("rtable.txt")	creates a router that has routing table defined in *rtable.txt* text file
IPSec G("policy.txt")	creates an IPSec device that has policy defined in *policy.txt* text file
Domain D("domain.txt")	creates a domain D that has an address and a network mask defined in *domain.txt* test file
Host H("host.txt")	creates a host H that has an address defined in *host.txt* test file
Rule r	creates a BDD representation for a firewall rule
link(C1, interface1, C2, interface2)	links components C1 and C2 through interface1 and interface2 respectively
buildDeviceBDD()	builds a BDD for each device in the network
buildGlobalBDD()	builds a BDD for the network
checkFlow(S, source-port, D, dest-port)	checks flows between source and destination using specified ports and returns a BDD that represents the computed flows
printFlows(B, n)	print the first n flows that satisfy the BDD B
getPathObjects(src, dst, vec, TYPE)	returns a vector *vec* of objects of type $TYPE$ along the path between source *src* and destination *dst*, TYPE can be FIREWALL, ROUTER, IPSEC, or IDS
policy()	returns the BDD representation for a firewall/router

a firewall, it contains the policy rules and two interfaces while it contains a routing table and up to 16 interfaces for a router. The configuration file is assigned logically to the proper device in the initialization statement in the program. To give a clear explanation how *ConfigLEGO* system works, we provide several code segments throughout the paper, where each segment solves a specific problem.

Any user program starts with initializing a network N stated as follows:

```
Network N;
```

Given a configuration file for each device, it can be added to the network *N*. A firewall $F1$ with configuration file *"f1.txt"* can be defined firewall as follows:

```
Firewall F1("f1.txt");
```

Other devices (routers, IPSec, etc.) can be added to the network similarly. After adding all components in the network *N*, the next step is to connect the components by introducing links between them. The connections between components are installed. *Link(. . .)* function is used to link two components as follows:

```
N.link(D1, ANY_IFACE, F1, 1);
```

Here, domain *D1* (any interface) is linked to firewall *F1* (interface 1).

After linking all components in network N, a BDD for the network and each device are generated by invoking the statements:

```
N.buildDeviceBDD(); N.buildGlobalBDD();
```

3 Verification Examples

In this section, we provide examples for showing the usability of *ConfigLEGO*. The examples are categorized as: Basic Analysis and Imperative Analysis.

3.1 Basic Analysis

ConfigLEGO can perform various security analysis, such as, reachability, intra-policy conflicts, and inter-policy conflicts. Due to the space limitation, we will show an example on reachability verification.

A traffic C is reachable to a distention node/domain D from a source node/domain S along an access route $\langle S, R_i, F_k, D\rangle$, iff the traffic is allowed by the routing table rule T_i^j [BDD for router R_i and port j] and the firewall policy F_k along that route. It can be formalized as follows:

$$reachable(C, S, D) : (C \Rightarrow \bigwedge_{(i,j)\in P} T_i^j) \wedge (C \Rightarrow \bigwedge_{(i,k)\in P} F_k).$$

ConfigLEGO checks the reachability by analyzing the BDDs for routers and firewalls along an access route between specified source and destination domain. This can be checked by the following statement:

```
T = N.checkFlow(src, src-port, dst, dst-port);
```

CheckFlow(...) returns a BDD, T, that represents the computed flows between a source *src* and a destination *dst* considering source and destination ports as provided in the function call. If the resultant BDD T is $bddfalse$, then there is no flow between source and destination. Flow computations are performed based on the *AccessRoute* and *AccessRoutePolicy* defined in section 2.1. *ConfigLEGO* can analyze the reachability between all source hosts and a single destinations or between all sources and all destinations by calling checkFlow(...) function inside a loop. The following example checks the reachability between all source hosts and a single destination D1:

```
// hSize is the number of hosts
int hSize = allHosts.size();BDD T, TC=TRUE;
for(i = 0; i < hSize; i++ ){
  T = N.checkFlow(*allHosts[i], ANY, D1, ANY);
  TC = TC | T;
  if( T != bddfalse )
    cout<<"Reachable from Host "<<i; }
```

Here, two BDDs, T and TC, are computed. T represents all flows from a host to a destination D1, and TC is the BDD for the complete representation from all hosts to the destination D1. An example of further analysis is to compare two hosts in term of the incoming traffic. Here, the BDD TC is the disjunction of all BDDs T, the operator | is overloaded to perform BDD "OR" operation.

3.2 Imperative Configurations Analysis

The *ConfigLEGO* system is capable of analyzing different imperative cases using sequence of evaluation steps which is one of the unique features of the system. The use of *loops* and *conditional statements* allows the users to comprehensively analyze these imperative queries.

Path Conflict Analysis for Firewalls. First, we introduce the formalization of *shadow-free* and *spurious-free* relations based on inter-policy firewall conflicts.

Lemma 1: Shadow-free and spurious-free are *transitive relations*. Assume S_a^i, S_a^j and S_a^k are upstream to downstream firewall policies in a path P, the following is always true:

$$[(\neg S_a^i \wedge S_a^j) = false] \bigwedge [(\neg S_a^j \wedge S_a^k) = false] \Rightarrow [(\neg S_a^i \wedge S_a^k) = false]$$

We formalize *Path Conflicts* using *path-shadowing* and *path-spuriousness.*

Definition 4 [Path Conflict]: Assuming S_a^i to S_a^n are the firewall policies from upstream to downstream in the path from x to y, a *path conflict(x,y)* between any two firewalls is represented as follows:

Path Shadowing(x,y):

$$\bigvee_{i=1..(n-1)\ and\ i \in path(x,y)} (\neg S_a^i \wedge S_a^{i+1}) \neq false$$

Path Spuriousness:

$$\bigvee_{i=1..(n-1)\ and\ i \in path(x,y)} (S_a^i \wedge \neg S_a^{i+1}) \neq false$$

ConfigLEGO checks this type of path conflicts as a sequence of steps (under a *loop*), where each step checks the conflicts between a pair of BDDs. The following code finds path shadowing between a source and a destination.

```
N.getPathObjects(src, dst, fwVec, FIREWALL);
for(i = 0 ; i < fwVec.size()-1 ; i++)
  if(!fwVec[i].policy() & fwVec[i+1].policy()){
    cout<<"Path Shadowing"; break; }
```

Here, getPathObjects(...) function returns a vector, $fwVec$, of all firewall objects between a source *src* and a destination *dst*. A pair of consecutive firewalls is checked for shadowing. The conflict is reported once found, the loop is stopped.

Reachability Requirement Verification. In large networks, some subnets are restricted to communicate with others, which is known as least privilege principle. For example, in a university network, student subnet is not allowed to use resources allocated for staff subnet. Network administrator can enforce least privilege by defining a reachability requirement matrix. Requirement matrix tells for each subnet which subnets are allowed to reach which signifies the soundness of the system. The soundness of a configuration can be defined as:

Definition 5 [Soundness]: a network configuration $\mathcal{C}$ is *sound* if, for all domains x and y, all possible paths from x to y are subset of the requirement matrix REQ. Formally, soundness can be defined as follows:

$$\forall x \forall y (reachable(x, y) \wedge src(x) \wedge dest(y)) \rightarrow REQ[x][y] = true$$

The following example verifies connection requirements between domains:

```
int domSize = allDomains.size();
int Req[domSize][domSize]; BDD T;
for( int i = 0; i < domSize; i++ )
 for( int j = 0; j < domSize; j++ )
  if( i != j){
   T=N.checkFlow(*allDomains[i], ANY, *allDomains[j], ANY);
   if(   (T != bddfalse && Req[i][j] == 0) ||
           (T == bddfalse && Req[i][j] == 1)   )
      cout<<"Reachability Violation";  }
```

Here, *Req* is the requirement matrix. If Req[i][j] is *ZERO*, then subnets i and j are not allowed to communicate.

4 Performance Evaluation

The various modules of *ConfigLEGO* framework have been implemented in C/C++ programming language using BuDDy2.2 package [4] and tested on a machine with a 1.8 GHz core 2 CPU and 2GB memory. Parsers have been developed for device configuration files. For evaluating imperative examples, *ConfigLEGO* analyzes a combination of device's BDDs (using loop and conditional statements) and infer about the configuration issues under consideration.

ConfigLEGO is evaluated with respect to time and space requirements. The framework has been tested under 100 different network configurations in more than 20 different test networks with up to 5000 nodes and 50 thousands of policy rules under each configuration. Table 2 shows the experimental results with different test cases. We have thoroughly analyzed the impact of network size and policy rules on network building time and configuration diagnosis time.

Impact of Network Size and Policy Rules on Space Requirement and Network Building Time: The space requirement basically covers the total BDD size for the network. *ConfigLEGO* framework creates a BDD for each

Table 2. Evaluation Results

Network Size	**Total No. of Rules**	**BDD Size (Mb)**	**Network Building Time (sec)**	**Configuration Analysis and Diagnosis Time (sec)**		
				Reachability	Flow analysis	Distributed Path Conflict
500	5000	1.6	0.665	0.235	0.65	0.37
1000	8500	3.2	1.325	0.43	1.32	1.33
1500	15000	4.6	1.95	0.65	1.89	3.2
2000	22500	6.3	2.67	0.885	2.5	5.32
3000	32500	9.65	3.92	1.38	3.78	11.27
4000	40125	12.7	5.12	1.82	5.25	21.5
5000	48755	15.8	6.34	2.33	6.52	32.12

network device by parsing the associated policy rule file. Thus, BDD size is linearly dependent on both network and policy size. Table 2 shows that space requirement lies within 15 MB for 5000 nodes and total of 50000 policy rules which is reasonably good for large networks.

Network model building time is almost linearly dependent on both network and policy rule size. It can be observed that this time lies within 7 seconds for 5000 nodes with 50000 policy rules.

Impact of Network Size and Policy Rules on Configuration Diagnosis Time: *ConfigLEGO* framework evaluates different configuration problems using Boolean satisfiability analysis of the network and device BDDs. The impact of network size on the evaluation time varies based on the problem complexity.

Reachability Analysis: ConfigLEGO checks the conjunction of all BDDs along a specified access. Thus, the reachability analysis time is linearly dependent on the number of nodes along that access path.

Flow Level Reachability: ConfigLEGO checks flow level reachability under a specific traffic flow C_k by evaluating BDDs for all routers (in a *loop*) and the destination firewall along the specified path P. For path level unreachability, *ConfigLEGO* analyzes all possible flows in a path P from node i to node j. Thus, the complexity of flow level reachability problem can be represented as $O(T_{pathreachability} * k)$, where, $T_{pathreachability}$ indicates the flow level reachability analysis time (for a specific flow) and k indicates the total flows. Table 2 shows the average flow level analysis time which lies within 6.5 seconds for 5000 nodes and 50000 policy rules.

The space and time requirement shows that the framework is scalable for large scale networks. The uniqueness of *ConfigLEGO* framework lies in comprehensive use of C/C++ programming language features and use of efficient BDD representation for systematically diagnosing different configuration problems.

5 Related Work

Researchers proposed different high level security policy languages and frameworks for automated management and modeling network configurations. *FLIP* [5],

is a high level conflict-free firewall policy language for enforcing access control based security and ensuring seamless configuration management. In *FLIP*, security policies are defined as high level service oriented goals, which can be translated automatically into access control rules. However, it limits in comprehensively specifying and analyzing the global network configuration with imperative queries. Chen et al. present a framework called COOLAID [2] for comprehensive management of network configurations by embedding knowledge bases from different network users. COOLAID uses a declarative framework for integrating explicit knowledge bases derived from low level network configurations and then reason about misconfigurations based on high level intents. However, this work does not provide the libraries for performing fine-grained security analysis and applications on top of the network configurations. This is an important requirement for providing diagnosability and provability of network configurations. Secondly, the scalability of network configuration management and automation have not been analyzed in these tools. Al-shaer et al. proposed a BDD based framework, ConfigChecker [3], for end-to-end verification of network reachability. Narain et al. [6] proposed a SAT-based approach for security configuration analysis. However, none of the earlier approaches provide an open interface for security configuration analysis.

The literature survey reveals the need of a framework that allows users to build the network configuration comprehensively as well as systematically analyze various configuration problems with imperative queries. Towards this goal, we develop the *ConfigLEGO* system exploiting the advantages of C/C++ programming language features and efficient BDD structure. The *expressiveness*, *composability*, and *reusability* features of *ConfigLEGO* allows user to comprehensively specify the network configuration and diagnosis requirements.

6 Conclusion and Future Work

In large scale networks, it is hard to find and debug misconfigurations and analyze security requirements manually. Thus, there is a need of an automated tool for finding and reasoning such misconfigurations in an abstract and comprehensive way. In this paper, we presented an imperative framework for comprehensively analyzing and diagnosing the network configurations. The framework is implemented in a tool called *ConfigLEGO*. It allows users to write C/C++ program that captures network specifications and implement the required analysis. This makes *ConfigLEGO* a convenient tool to use. For the purpose of analysis and diagnosis, *ConfigLEGO* uses an efficient BDD engine, this engine hides BDDs complexity and does not require previous knowledge about BDDs representation and operations. The efficiency of the framework has been evaluated rigorously with different network and policy rule sizes. In future, the *ConfigLEGO* framework can be extended for finer-grained security analysis like, role based access control and risk based policy configuration in enterprise networks.

References

1. Al-Shaer, E.S., Hamed, H.H.: Discovery of Policy Anomalies in Distributed Firewalls. In: Proceedings of IEEE INFOCOM 2004, Hong Kong, China, pp. 2605–2626 (March 2004)
2. Chen, X., Mao, Y., Mao, Z.M., Van der Merwe, J.: Declarative Configuration Management for Complex and Dynamic Networks. In: Proceedings of ACM CoNEXT (2010)
3. Al-Shaer, E., Marrero, W., El-Atway, A., AlBadani, K.: Network Configuration in a Box: Towards End-to-End Verification of Network Reachability and Security. In: Proceedings of ICNP 2009, Princeton, NY, USA, pp. 123–132 (2009)
4. Lind-Nielsen, J.: The BuDDy OBDD package, `http://sourceforge.net/projects/buddy/`
5. Zhang, B., Al-Shaer, E.S., Jagadeesan, R., Riely, J., Pitcher, C.: Specifications of A High-level Conflict-Free Firewall Policy Language for Multi-domain Networks. In: Proceedings of ACM SACMAT 2007, France, pp. 185–194 (June 2007)
6. Narain, S., Levin, G., Malik, S., Kaul, V.: Declarative Infrastructure Configuration Synthesis and Debugging. Journal of Network and Systems Management 16, 235–258 (2008)