Implementation of a Query Language for Attack Graph Analysis

A Thesis submitted in partial fulfillment of the requirements for degree of

Master of Computer Application

Department of Computer Science and Engineering
Jadavpur University, Kolkata

By

Dipayan Ghosh

Examination Roll No: MCA196018
Class Roll No: 001610503022
Registration No: 137329 of 2016-17

Under the guidance of

Mridul Sankar Barik

Assistant Professor
Department of Computer Science & Engineering
Jadavpur University
Kolkata – 700 032

May, 2019

Department of Computer Science and Engineering,
Faculty of Engineering and Technology,
Jadavpur University, Kolkata 700032

Certificate of Approval

This is to certify that the thesis entitled "**Implementation of a Query Language for Attack Graph Analysis**" is a bona-fide record of work carried out by Dipayan Ghosh in partial fulfillment of the requirements for the award of the degree of Master of Computer Application, in the Department of Computer Science and Engineering, Jadavpur University during the period December 2018 to May 2019. It is understood that by this approval, the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein, but approve the thesis only for the purpose for which it is submitted.

Examiners:

____________________________ ____________________________

(Signature of the Examiner) (Signature of the Supervisor)

To whom it may concern

This is to certify that the work in this thesis entitled **"Implementation of a Query Language for Attack Graph Analysis"** has been satisfactorily completed by **Dipayan Ghosh.** It is a bona-fide piece of work carried out under my supervision and guidance for partial fulfillment of the requirements for the awarding of the Master of Computer Application degree of the Department of Computer Science & Engineering, Faculty of Engineering & Technology, Jadavpur University, during the academic year 2019-20.

Mridul Sankar Barik
Department of
Computer Science & Engineering
Jadavpur University

Countersigned:

Prof. Mahantapas Kundu	**Prof. Chiranjib Bhattacharjee**
Head of the Department	**Dean**
Computer Science & Engineering	**Faculty of Engineering & Technology**
Jadavpur University	**Jadavpur University**

Declaration of Originality and Compliance of Academic Ethics

I hereby declare that this thesis contains literature survey and original research work by the undersigned candidate, as part of my Master of Computer Application studies.

All information in this document has been obtained and presented in accordance with academic rules and ethical conduct.

I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name: **Dipayan Ghosh**

Exam Roll Number: MCA196018

Thesis Title: **Implementation of a Query Language for Attack Graph Analysis**

Signature with date:

Acknowledgement

With my sincere respect and gratitude, I would like to thank my thesis guide **Sri Mridul Sankar Barik** for his continuous support for this thesis work, for his patience, motivation and enthusiasm. His guidance helped me a lot throughout the duration of the work. His valuable suggestions inspired me a lot. I feel deeply honored that I got the opportunity to work under his guidance.

I would also wish to thank Prof. Mahantapas Kundu, Head of the Department of Computer Science & Engineering, Jadavpur University for providing me all the facilities and for his support to the activities of this project.

I would like to thank my classmate friend, Kallol Chowdhury for sharing his knowledge and experience with me and also their immense support and co-operation.

Last, but not the least, I would like to thank all my family members and my classmates of Master of Computer Application batch of 2016-2019, for their co-operation and support. Their wealth of experience has been a source of strength for me throughout the duration of my work.

Dipayan Ghosh
Exam Roll No: MCA196018
Registration No: 137329 of 2016-17
Master of Computer Application

Contents

List of Figures

List of Tables

Abstract

Attack graphs constitute a powerful security tool aimed at modeling the many ways in which an attacker may compromise different assets in a network. Attack graphs are most used by System / Security administrators to determine how vulnerable their systems are and also to determine what security measures to deploy inorder to defend their systems. Our thesis provides an abstraction for the need of query language for attack graph generation and analysis based on the network vulnerabilities. The proposed query language focuses to help the security analysts to formulate new types of analysis tasks in the form of queries, without requiring knowledge of any specialized graph algorithms and/or graph extraction methods as it hides the underlying data model. We have designed its implementation in such a way that it can be used on top of any kind of data store i.e. relational database, graph database etc. The generation of graph is done considering the network information is granular.

Chapter 1

Introduction

Our thesis aims to implement a Domain Specific Query Language (DSQL). By definition a domain-specific language is a computer language specialized to a particular application domain. Introducing the Attack Graph Query Language (AGQL) (please refer the Figure 1-1), a DSQL which is targeted towards facilitating both attack graph generation and attack graph based security analysis thereby hiding the underlying data model. This chapter introduces the interpreter, thereby following with motivation and thesis layout.

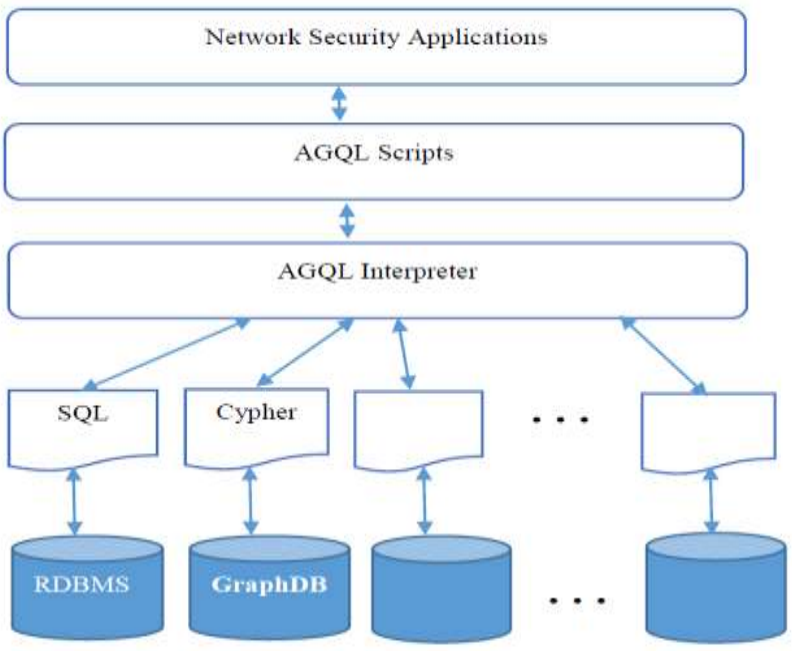


Figure 1-1 Attack Graph Query Language

1.1 Motivation

Through our research work we intend to provide a higher level of abstraction of network security state (represented as attack graph) to the System / Security administrators, where they can use a flexible declarative query language AGQL for specifying the security relevant information needed for the analysis tasks, on the fly. To the best of our knowledge, among the query languages been developed so far in the domain of network security, CyQL [12] is the only one which uses such attack graph model as an abstraction of the underlying network security state. Our language can give a good competition as we provide more flexibility in the underlying data models and even in the language itself.

1.2 Thesis Layout

The thesis is organized as follows. After the completion of this Chapter 1 which consists of sections - Motivation, thesis layout, we would find Chapter 2 with some past works i.e. a literature survey is done. In Chapter 3, we would basically start to understand the basics of attack graph with examples. Next, we will begin with our query language AGQL in Chapter 4. In this Chapter we would learn the syntax of the language with some examples and also deal with the need for developing this query language. Then in Chapter 5, we will learn about the RDBMS Backend Design. Subsequently in Chapter 6, we will learn about the Graph Database Backend Design. Chapter 7 deals with the Implementation, which is the main part of the thesis and it has sections consisting of the flowchart of the Interpreter, ER Diagram, Class Diagram, and algorithms of the queries. It also has a brief explanation about the data structures used, error messages, exception handling, assumptions taken, features of the interpreter. Chapter 8 is all about the attack graph generation using TVA Approach. Chapter 9 depicts a brief on the AGQL editor that is developed using JFrame. This shall be improvised more in later stages. Finally, Chapter 10 concludes with the real life usage of this language and thereby comments on the future work that are can make the language more useful in day to day life. Appendix A is the clear picture on the concrete syntax that is used in our interpreter. Appendix B is the sample code that is taken as input and was executed by the interpreter.

Chapter 2

Literature Survey

Attack graph is a model of knowledge about the inter-dependency between vulnerabilities and the connectivity in network. By regarding a collection of security –related conditions as a state and each exploit as a transition between these states, an attack graph can be generated by searching forward from the initial state or backwards from a given goal state. This chapter discusses on the few past works related to Attack graphs and compiler/interpreter published by researchers.

Jiawei Han et al. [1] have designed a data mining query language (DMQL) and has presented it step-by-step. They have designed inorder to provide necessary primitives for data mining engines to work in. It has preferred to use flexible GUIs to interact with a data miner for fruitful and convenient data mining. DMQL is said to serve as a "core language" for data mining system implementations, on top of which various kinds of GUIs should be developed for effective data mining. Its prime functionalities are Data Collection, Presentation of data mining results, Manipulation of data mining primitives, Interactive multi-level mining and other miscellaneous information.

Chunying Wang et al. [2] have proposed their approach and software tool for vulnerability analysis and security level assessment of computer networks, intended for implementation at various stages of a life cycle of computer networks. They have associated probabilities with transition in the model inorder to be able to find the reachability probability of various states in an Attack Model and termed as 'Probabilistic Model'. Their model is also capable in distinguishing the leaf nodes marked to be 'states' as white – nodes depending on the fact that the node is not detected to be an attacker by their intruder detection system or black – nodes which has sounded the alarm.

Sushil Jajodia et al. [3] has used the TVA tool for analyzing vulnerability to network attacks. Their overall architecture has three main components: (1) a knowledge base of modeled exploits, (2) a description of a network of interest, and (3) a specification of an attack scenario (attacker target, initial attack control, and network configuration changes). The TVA analysis engine merges these three components and then discovers attack paths (exploit combinations) based on the merged model. They build the dependency graph through a multi-step process. But there is a limit in accomplishing with network hardening and intrusion detection, particularly in the face of novel attacks with the need to offer services and effectiveness.

Paul Ammann et al. [4] has assumed of monotonicity to obtain a concise, scalable graph-based representation for encoding attack trees. It has two implications. (1) The precondition of an exploit, once satisfied, never becomes unsatisfied (attributes may start in the 'not satisfied' state and become transformed, via some exploit, to the 'satisfied' state, but never the reverse). (2) The monotonicity assumption requires that the negation operator is not used to express the precondition of any exploit. They assumed that the preconditions of an exploit are conjoined; this assumption entails no loss of generality, since disjunctions could easily be handled by splitting the exploit into multiple exploits. They also assumed that post conditions are conjoined, since it simplifies their algorithms.

In [5] authors have given a simple interpretation of metric for their attack graph. That is, the individual score $p(e)$ is the probability that any attacker can, and will execute e during an attack, given that all the preconditions are already satisfied. Equivalently, among all attackers that attempt to compromise the given network during any given time period, $p(e)$ is the fraction of attackers that can, and will execute e. This interpretation of individual scores has considered two factors in determining the individual score $p(e)$, namely, whether an attacker has the skills and resources to execute e and whether he/she will choose to do so. The interpretation of individual scores also provides a natural semantics to the cumulative scores. That is, $P(e)$ or $P(c)$ stands for the likelihood, or the fraction of, attackers who will successfully exploit e or satisfy c in the given network. Such a likelihood or fraction that a corresponding resource will be compromised during an attack is clearly relevant in analyzing the security of a network or in hardening the network for better security.

The Research paper [6] described about the specifications of the CQL language for continuous queries over data streams, including a formal abstract semantics on which CQL is based. Most of the CQL language is operational in the STREAM prototype data stream management system, including the “Linear Road” benchmark used as examples throughout the paper. The paper also describes the structure and implementation of query execution plans for CQL in the STREAM system. The have presented the structure of CQL’s query execution plans as well as details of the most important components like: operators, inter-operator queues, synopses, and sharing of components among multiple operators and queries.

Naggen (Network Attack Graph GENerator) [7], a security tool aimed at the generation and visualisation of core graphs. Naggen is composed of three main building blocks: (1) Naggen Shell, a command-line interface that allows to configure and control the generation process, (2) Naggen Core, responsible for the analysis and graph generation processes, and (3) Naggen Display, which contains different visualization mechanisms to display the generated attack graphs. The three potential security applications using Naggen are: (1) network monitoring and hardening, (2) security perimeters, and (3) forensic investigations. The proposed approach relies in identifying the main attack avenues towards specific network targets by performing a structural summarization process over the input network. The process essentially summarizes alternative routes between any two directly connected nodes and only keeps those routes than cannot be summarised into any other link in the graph. As a result, the obtained graphs presents simpler structures. The theoretical and practical results due to compactness of core attack graphs, or simply, core graphs strongly suggest that their approach can widely support the actual use of attack graphs in practice and therefore, having a significant impact in network security terms.

In the proposed work of [8], they have presented the data manipulation facility for a structures English Query Language (SEQUEL) which can be used for accessing data in an integrated relational database. Without resorting to the concepts of bound variables and quantifiers SQEUEL identifies a set of simple operations on tabular structures, which have been shown to be of equivalent power to the first order predicate calculus. A SEQUEL user is presented with a consistent set of English templates which reflect how people use tables to obtain information. Moreover, the SEQUEL user is able to compose the basic templates in a structured manner inorder to form more complex queries. SEQUEL is intended as a data base sublanguage for both the professional programmer and the more infrequent database user.

The authors of [9] has presented a method for vulnerability analysis of computer networks. The method is based on attack graphs which represent attack states and the transitions between them. Edges are directed arcs that connect one system state to the next. Edges represent a change of state caused by a *single step.* Templates list the required conditions for state transitions. The graph is generated by matching the current (node) state against a library of templates, choosing only the templates that apply to the current state. The attack graph can be used to identify unique attack paths that are most likely to succeed, or to simulate various attacks. The dynamic aspect of the graph gives unique insight into the behavior of the system. The graph analysis tool goes beyond tools which check a “laundry list” of services or conditions that are enabled on a particular machine. The graph can show new attack paths which can by-pass the static fortifications of a typical security system. They developed a robust attack graph tool with a graphical interface. The tool is designed to be fairly easy to use and to link to vulnerability and configuration databases from commercial tools. They plan to evaluate the utility of this approach for testing intrusion detection systems and/or for identifying where an attacker might next move once an intruder has been detected within a network. The attack graph could also be the basis for identifying the most cost effective set and placement of defenses.

Data extraction, conversion, transformation, and integration are all well-understood database problems, where solutions rely on a query language, either relational (SQL) or object-oriented (OQL). **Authors of [10]** have implemented such an interpreter for XML-QL (a query language for XML) using the unordered data model. XML-QL has features like declarative; ‘relational complete’- in particular, it can express joins; simple enough that known database techniques for query optimization, cost estimation, and query rewriting could be extended to XML-QL; extracts data from existing XML documents and construct new XML documents; supports both ordered and unordered views on an XML document. XML-QL supports querying, constructing, transforming, and integrating XML data. They have drawn comparison with other query languages for semi-structured data.

In paper [11], they have described a new query language for information retrieval in XML documents. They have presented the new query language XIRQL which integrates most from logic-based probabilistic IR-related features like weighting and ranking, relevance-oriented search, datatypes with vague predicates, and semantic relativism and have described the concepts that are necessary in order to arrive at a consistent model for XML retrieval. For processing XIRQL queries, they have described a path algebra, which also serves as a starting point for query optimization. In parallel, XIRQL can be extended to include the data-centric features from XQuery. They have implemented a first prototype retrieval engine that accepts XIRQL queries, transforms them into a path algebra expression and then processes this expression.

Authors of [12] have described CyGraph, a system for improving network security posture, maintaining situational awareness in the face of cyberattacks, and focusing on protection of mission-critical assets. CyGraph adopts a unified graph-based cybersecurity model relevant to potential and actual cyberattacks, defenses, and mission impacts. It captures incremental attack vulnerability, security events, and mission dependencies within a network environment, builds a predictive model of possible attack paths and critical vulnerabilities, and correlates events to known vulnerability paths. It also includes dependencies among mission requirements and network assets, for analysis in the context of mission assurance. Their resulting knowledge graph captures the complex relationships among entities in the cybersecurity domain. CyGraph brings together isolated data and events into an overall picture for decision support and situational awareness. It prioritizes exposed vulnerabilities, mapped to potential threats, in the context of mission-critical assets. In the face of actual attacks, it correlates intrusion alerts to known vulnerability paths and suggests best courses of action for responding to attacks. For post attack forensics, it shows vulnerable paths that may warrant deeper inspection. CyGraph also supports CyQL (CyGraph Query Language), a domain specific query language for expressing graph patterns of interest, with interactive visualization of query results. CyGraph supports the separation of graph models into interdependent layers. For time-dependent graph models, it provides dynamic visualization of evolving graph state. CyGraph also integrates with third-party tools like Neo4j for visualizing graph state changes (e.g., driven by simulations).Furthermore, it has capabilities for synthesizing graph models with particular statistical properties.

In [13] the authors have proposed for a vulnerability analysis tool. Their end-to-end framework model used in the analysis is able to automatically integrate formal vulnerability specifications from the bug-reporting community and is be able to scale to networks with thousands of machines. Their reasoning system that conducts multihost, multistage vulnerability analysis on a network. Their analysis algorithm is divided into two phases: attack simulation and policy checking. MulVAL adopts Datalog as the modeling language for the elements in the analysis (bug specification, configuration description, reasoning rules, operating-system permission and privilege model, etc.). We easily leverage existing vulnerability-database and scanning tools by expressing their output in Datalog and feeding it to our MulVAL reasoning engine. Once the information is collected, the analysis is performed in seconds for networks with thousands of machines. They implemented their framework on the Red Hat Linux platform. Their framework can reason about 84% of the Red Hat bugs reported in OVAL, a formal vulnerability definition language. They tested their tool on a real network with hundreds of users. The tool detected a policy violation caused by software vulnerabilities and the system administrators took remediation measures.

Chapter 3

Overview of Attack Graph

Attack graphs have important applications in protecting critical resources in networks against sophisticated multi-step intrusions. Currently, analyses of attack graphs largely depend on proprietary implementations of specialized algorithms. This chapter is to clear the basic understanding of the underlying concepts of Attack graph and their types.

3.1 Attack Graph

Attack graphs depict ways in which an adversary can exploit vulnerabilities to break into a system. System administrators analyze attack graphs to understand where their system's weaknesses lie and to help decide which security measures will be effective to deploy. They are mathematical abstractions of the details of possible attacks against a network. The current analysis of attack graphs requires an algorithm to be developed and implemented, causing a delay in the availability of Analysis. Such a delay is usually unacceptable because the needs for analyzing attack graphs may change rapidly in defending against network intrusions. Attack graph, being first introduced in 1998, is a modeling tool which is based on the casual relationship between different vulnerability exploits. An Attack Graph enumerates all possible attack paths that an attacker can follow to intrude into a target network and compromise its critical resources.

3.2 Nodes

A Node is a data or record. Nodes in an attack graph (please refer the Figure 3-1), represent possible system state during execution of an attack and edges represent a change of state, caused by a single action of the attacker. Nodes are of two types: Exploit nodes and Security Condition nodes.

3.2.1 Exploit Nodes

Nodes that are drawn as ovals represent exploits and are labelled with corresponding vulnerabilities. These nodes represent attacks (exploitation of certain vulnerabilities). We can see from the figure 3-1 that (h2, h1, sadmind_bof), (h3, h1, sadmind_bof), (h1, h2, sadmind_bof), (h3, h2, sadmind_bof) with oval shaped border are all exploits.

3.2.2 Security Condition Nodes

Nodes particularly without any boundary represent different security conditions, i.e. network condition, attacker capability etc. These nodes represent either the pre-condition or the post-condition of an attack. We can see that the labeled nodes without borders like (h1, sadmind_service), (h3, user_privilege), (h1, user_privilege), (h2, sadmind_service), (h2, user_privilege) are all security conditions in the figure 3-1 given below.

Example of an attack graph:
The simplified version of the left-hand side figure (Figure 3-1) is given in the right-hand side (refer Figure 3-2), where the exploits are given as triplets and security conditions as duplets. The left-hand side of below figure (refer figure 3-1) shows a simple attack graph as our running example. The attack graph indicates that by exploiting a buffer overflow vulnerability in the Sadmind service (Nessus ID 11841), an attacker can gain the privilege of using a remote machine. The right-hand side shows a simplified version where '*x*' denotes the existence of the vulnerability, '*y*' denotes the user privilege & '*A*' denotes the exploitation of that vulnerability. The attack graph shows an attacker having user privilege on host 3 can exploit the vulnerability on hosts 1 and 2 and obtain user privilege on the hosts. Please note that after an attacker has obtained user privilege on host 1, he/she can then exploit host 2 from either host 3 or host 1.

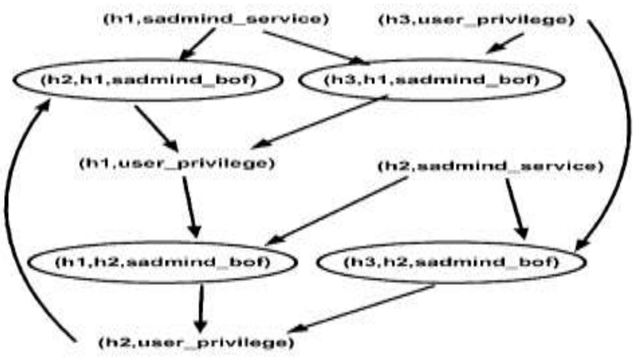


Figure 3-1 A running example of an Attack Graph

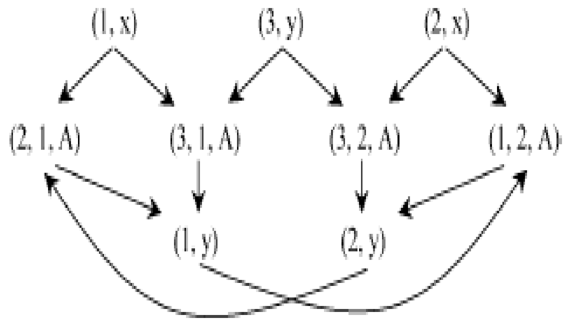


Figure 3-2 Simplified Version

The two types of edges in an attack graph have different semantics. The require relation is regarded as conjunctive, whereas the imply relation is disjunctive. More specifically, an exploit cannot be realized until all of its required conditions have been satisfied (different variations of an exploit that require different sets of conditions should be regarded as different exploits), but a condition can be satisfied by one of the realized exploits that imply that condition. Another important perspective is that conditions can be divided into initial conditions (those not implied by any exploit) and intermediate conditions. The main reason for such a distinction is that initial conditions can be independently disabled to harden a network, whereas intermediate conditions cannot be without first removing the exploit implying them.

3.3 Types of Attack Graph

There are different models of attack graphs which are been used by many researchers i.e. the state enumeration graph, exploit dependency graph, logical attack graph, multiple prerequisite attack graph, etc. Attack graphs that we shall adopt in our work is based on the notion of exploit dependency graphs. The above figure (Figure 3-1) is an example of exploit dependency graph.

Chapter 4

Attack Graph Query Language

AGQL is a Domain Specific Language (DSL) to facilitate modeling of input data necessary for generation of attack graphs. In this chapter we shall be dealing with the structure of different AGQL queries through few examples. The minimum hardware and software requirements for running is taken to be 1 GB of RAM, 50GB of free space in Hard Disk, Intel Pentium 2.8 GHz processor, Linux/Windows Operating System, PostgreSQL, Neo4j, javacc, java installed in the respective machine.

4.1 Need for a Query Language

AGQL has query constructs to help in attack graph based network security analysis. AGQL provides a higher level abstraction where, the user need not bother about underlying data management system, be it relational, graph data, etc. By convention, design of a domain specific language AGQL begins by defining the syntax of the queries that is conceived solely based on the use cases in the application domain.

4.2 Query

A query is a request for information from a database. Many database systems requires the user to make requests for information in the form of a stylized query that must be written in a special query language, which is the most complex method because it forces to learn a specialized language, but it is also the most powerful. AGQL is one such initiative. AGQL queries can be categorized into four categories: data definition queries, data manipulation queries, transaction control queries and data control queries (please refer the Figure 4-1).

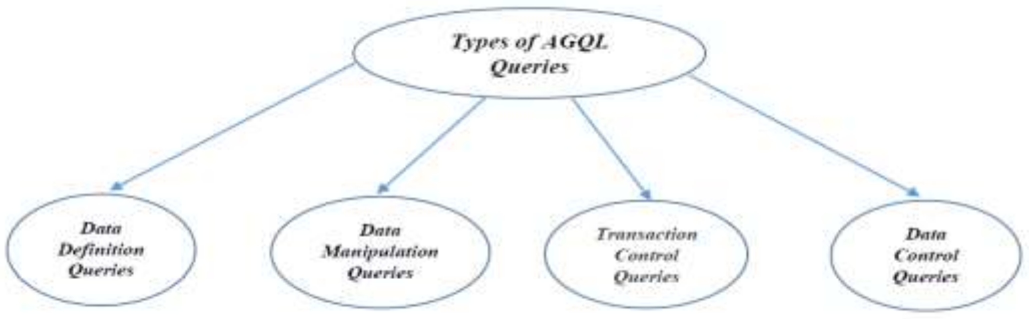


Figure 4-1 Types of AGQL Queries

4.2.1 Data Definition Language

These AGQL queries are used for creating, modifying, and dropping the structure of database objects. The query falling under this category are described below:

4.2.1.1 Entity Type Definition

Entity type define query constructs are used as data definition queries. It facilitate definition of entity types. This allows us to define an entity type. Hence for this query we need to provide name of the entity type and an attribute definition list.
Few examples for defining the entity types in our query language:

1. define entity-type host (name: string, ipAddr: string, macAddr: string, os:string)
2. define entity-type network-domain (name:string, netAddr:string, subnetMask:string)
3. define entity-type service (name:string, protocol:string, portNo:int, swName:string, swVer:string)
4. define entity-type vulnerability (name:string, cveId:string)
5. define entity-type privilege (privType:string)
6. define entity-type reachability (rchType:string)
7. define entity-type firewall (name:string, ifCount:int, ifIpAddr:string, ifSubnetMask:string)
8. define entity-type gateway (name:string, ifIpAddr:string, ifSubnetMask:string)

4.2.1.2 Relation Type Definition

Relation type define query constructs are used as data definition queries. It facilitate definition of relation types. This allows us to define a relation type between two set of entities. Hence for this query we need to provide name of the relation type and an attribute definition list and one set each for source entities and target entities respectively.
Few examples for defining the relation types in our query language:

1. define relation-type memberOf (since: time) between {host, firewall}, {network-domain}
2. define relation-type connects (ifId: int) between {gateway}, {network-domain}
3. define relation-type atHost () between {service, privilege}, {host, gateway}
4. define relation-type accessTo () between {reachability} , {service}
5. define relation-type accessBy () between {reachability} , {host, network-domain}
6. define relation-type hasVuln () between {service} , {vulnerability}

4.2.1.3 Security Condition Type Definition

Security condition type define query constructs are used as data definition queries. It facilitate definition of security condition types. This allows us to define a security condition type. Hence for this query we need to provide name of the security condition type, an attribute definition list and a corresponding security condition types with its entities.
Few examples for defining the security condition types in our query language:

1. define security-condition-type Reachability (name: string, since: time) accessBy {network-domain, host} accessTo {service}
2. define security-condition-type Privilege (name: string, since: time, privType: string) atHost {host, gateway}

4.2.1.4 Exploit Type Definition

Exploit type define query constructs are used as data definition queries. It facilitate definition of exploit types. This allows us exploit to define a type. Hence for this query we need to provide name of the exploit type and an attribute definition list.
An examples for defining exploit types in our query language:

1. define exploit-type bofExploit (name: string, category: string) CVE("CVE-2008-0106")
   precond {privilege (privType: "user"), reachability (rchType: "xyz")}
   postcond {privilege (privType:"user")}

4.2.1.5 Unique Constraint Definition

Unique key constraint is a type of constraints that the every instances of entities must conform to. This query constructs specify the attribute(s) that uniquely identifies instances of a particular entity type.
Few examples for declaring unique key constraints in our query language:

1. define unique host (ipAddr)
2. define unique vulnerability (cveId)

4.2.1.6 Cardinality Constraint Definition

Cardinality constraint is a type of constraints that every instances of entities must conform to. This query constructs specify the way in which two entities may be related.
Few examples for declaring cardinality constraints in our query language:

1. define cardinality memberOf n:1
2. define cardinality hasVuln n:n

4.2.1.7 Drop

The DROP query is used to drop or delete a definition. Dropping of a defined type will drop all database objects inside it.
Few examples for drop query in our language:

1. drop entity-type (host, network-domain, service, vulnerability, firewall, gateway, privilege, reachability)
2. drop relation-type (memberOf, connect, atHost, accessTo, accessBy, hasVuln, runAt)
3. drop security-condition-type(reachability, privilege)
4. drop exploit-type(bofExploit)

4.2.2 Data Manipulation Language

These AGQL queries are used for storing, retrieving, modifying, and deleting data. The query falling under this category are described below:

4.2.2.1 Entity Creation

Entity create query constructs are used as data manipulation queries. It allows us to create an instance of entities. The query creates entities of the entity type already defined. Hence for this query we need to provide name of the entity and the attribute value list which should match with attribute definition list of the entity type that is defined.

Few examples for creating entities in our query language:

1. create entity host (name:"h1", ipAddr:"192.168.148.3", macAddr:"xx:xx:xx:xx:xx:xx", os: "Ubuntu")
2. create entity host (name:"h2", ipAddr:"192.168.148.5", macAddr:"xx:xx:xx:xx:xx:xx", os: "Win7")
3. create entity host (name:"h3", ipAddr:"192.168.148.3", macAddr:"xx:xx:xx:xx:xx:xx", os: "Win7")
4. create entity domain (name:"dom1", netAddr:"192.168.148.1", SubnetMask: "255.255.255.0")
5. create entity domain (name:"dom2", netAddr:"192.168.148.1", SubnetMask: "255.255.255.0")
6. create entity network-domain (name:"nd1", netAddr:"192.168.148.3", SubnetMask: "255.255.255.0")
7. create entity network-domain (name:"nd2", netAddr:"192.168.148.4", SubnetMask: "255.255.255.0")
8. create entity firewall (name:"fw1", ifCount:2, ifIpAddr:"192.168.150.1", ifSubnetMask: "255.255.255.0")
9. create entity firewall (name:"fw2", ifCount:2, ifIpAddr:"192.168.148.1", ifSubnetMask: "255.255.255.0")
10. create entity firewall (name:"fw3", ifCount:2, ifIpAddr:"192.168.148.1, 192.162.148.1", ifSubnetMask:"255.255.255.0, 255.255.255.0")
11. create entity host (hostname:"h1", ipAddr:"192.128.148.3", macAddr:"00:98:CC:0C:BA:12", os:"Win7")
12. create entity host (name:"h5", ipAddr:"193.168.148.3", macAddr:" 30:98:CC:0C:BA:12")
13. create entity service (name:"httpd1", protocol:"ftp", portNo:3, swName:"vsc", swVer:"1.1.2")

4.2.2.2 Relation Creation

Relation create query constructs are used as data manipulation queries. It allows us to create an instance of relations. The query creates relations of the relation type already defined. Hence for this query we need to provide name of the relation and the attribute value list which should match with attribute definition list of the relation type that is defined.
Few examples for creating relations in our query language:

1. create relation x:host (x.ipAddr == "192.168.148.3") memberOf (since:"10:11:34") nd1:network-domain (nd1.netAddr == "192.168.148.3")
2. create relation h1:host (h1.ipAddr == "192.168.148.3") memberOf (since:"10:11:34",ifId: 0,name:"mmm") nd1:network-domain (nd1.netAddr == "192.168.148.3")
3. create relation h1:host (h1.ipAddr == "192.168.148.5") memberOf (since:"10:11:34") nd1:network-domain (nd1.netAddr == "192.168.148.3")
4. create relation h1:host (h1.ipAddr == "192.168.148.3") memberOf (since:"10:11:34") nd1:network-domain (nd1.netAddr == "192.168.148.3")
5. create relation fw1:firewall (fw1.name == "fw1") memberOf (since:"10:11:34") nd1:network-domain (nd1.netAddr == "192.168.148.3")
6. create relation fw1:firewall (fw1.name == "fw2") memberOf (since:"10:11:34") nd1:network-domain (nd1.netAddr == "192.168.148.4")

7. create relation fw1:firewall (fw1.name == "fw1") memberOf (since:"10:11:34") nd1:network-domain (nd1.netAddr == "192.168.148.4")
8. create relation fw1:firewall (fw1.name == "fw1") memberOf (since:"10:11:34") nd1:network-domain (nd1.netAddr == "192.168.148.3")

4.2.2.3 Security Condition Creation

Security Condition create query constructs are used as data manipulation queries. It allows us to create an instance of security conditions. The query creates security conditions of the security conditions type already defined. Hence for this query we need to provide name of the security condition and the attribute value list which should match with attribute definition list of the security condition type that is defined.
Few examples for creating relations in our query language:

1. create security-condition reachability (name:"httpd31", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.3")} accessTo {y:service (y.name == "httpd1")}
2. create security-condition reachability (name:"httpd32", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.3")} accessTo {y:service (y.name == "httpd2")}
3. create security-condition reachability (name:"httpd12", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.1")} accessTo {y:service (y.name == "httpd2")}
4. create security-condition reachability (name:"httpd21", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.2")} accessTo {y:service (y.name == "httpd1")}
5. create security-condition privilege (name:"user3", privType: "user") atHost {x:host (x.ipAddr == "192.168.148.3")}
6. create security-condition privilege (name:"user1", privType: "user") atHost {x:host (x.ipAddr == "192.168.148.1")}
7. create security-condition privilege (name:"user2", privType: "user") atHost {x:host (x.ipAddr == "192.168.148.2")}

4.2.2.4 Exploit Creation

Exploit create query constructs are used as data manipulation queries. It allows us to create an instance of exploits. The query creates exploits of the exploit type already defined. Hence for this query we need to provide name of the exploit and the attribute value list which should match with attribute definition list of the exploit type that is defined.
Few examples for creating exploits in our query language:

1. create exploit bofExploit(name:"bofExp31", category:"remote")
   precond {x:privilege (x.name == "user3"), y:reachability (y.name == "httpd31")}
   postcond {x:privilege (x.name == "user1")}
2. create exploit bofExploit(name:"bofExp32", category:"remote")
   precond {x:privilege (x.name == "user3"), y:reachability (y.name == "httpd32")}
   postcond {x:privilege (x.name == "user2")}

4.2.2.5 Select

AGQL SELECT query is used to query or retrieve data regarding different network components from a table in the database. These query may retrieve information from specified columns or from all of the columns in the table or from multiple tables.
Few examples for select query in our language:

1. select x:host where (x.ipAddr=="192.168.148.3")

2. select x:host where (x.ipAddr=="192.168.148.3") and (x.name=="h1")
3. select x:host where (x.name=="h1") or (x.os== "win7")
4. select x:host where (x.ipAddr=="192.168.148.5") or (x.ipAddr=="192.168.148.1")
5. select x:host where (x.ipAddr=="192.168.148.3") and (x.host_Id >= 3) and (x.name == "h3")
6. select x:host where (x.ipAddr=="192.168.148.3") and (x.name == “h2”) or (x.name == "h3")
7. select x:host where (x.name == "h1") or (x:host memberOf nd1:network-domain)
8. select x:host where (x.ipAddr=="192.168.148.3") or (x:host memberOf nd1:network-domain)
9. select x:host where (x.os== "win7") and (x:host memberOf nd1:network-domain)
10. select x:network-domain where (x.name == "nd1")

4.2.2.6 Delete

AGQL DELETE query is used to delete tuples from relational database.
Few examples for delete query in our language:

1. delete x:host where (x.host_Id >= 3)
2. delete x:host where (x.ipAddr == "192.168.128.1")
3. delete x:host where (x.ipAddr=="192.168.148.3")
4. delete x:host where (x.name=="h1")
5. delete x:host where (x.os=="Win7”)
6. delete x:network-domain where (x.name == "nd1")

4.2.3 Transaction Control Language

These AGQL queries are used for managing changes affecting the data and might have functionality like to save the changes, to roll back the changes, to create points within the groups of transactions in which to roll back, to place a name on a transaction, which we will soon introduce.

4.2.4 Data Control Language

These AGQL queries are used for providing security to database objects. These queries might have functionality like provide access or privileges on the database objects to the users and enforce database security in a multiple user database environment, which we will soon introduce.

4.3 Operator

An operator is a reserved word or a character used primarily in an AGQL statements. Two types of Operators are widely used in our language: Relational Operator and Logical Operator.

4.3.1 Relational Operator

Operators are used to specify conditions in an AGQL query and to serve as conjunctions for multiple conditions in a query.

- == Checks if the value of two operands are equal or not, if yes then condition becomes true. Example: (x.ipAddr == "123.111.103.3") is not true.
- > Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. Example: (x.ipAddr > y.ipAddr) is not true.
- < Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. Example: (x.ipAddr < y.ipAddr) is true.
- >= Checks if the value of left operand is greater than or equal to the value of right operand, if yes; condition becomes true. Example: (x.ipAddr >= y.ipAddr) is not true.
- <= Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true Example: (x.ipAddr <= y.ipAddr) is true.

4.3.2 Logical Operator

AGQL Logical Operators are used in a query and has some special functionalities:
Here is a list of all the logical operators available in AGQL.
Operator Description

- AND: The AND operator allows the existence of multiple conditions in an SQL query's WHERE clause.
- BETWEEN: The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
- OR: The OR operator is used to combine multiple conditions in an SQL query's WHERE clause.
- UNIQUE: The UNIQUE operator searches every tuple of a specified table for uniqueness (no duplicates).

4.4 Clause

AGQL Clauses was designed to help the programmers and IT professionals. The AGQL WHERE clause is used to specify a condition while fetching the data from a single table or by joining with multiple tables. If the given condition is satisfied, then only it returns a specific value from the table. You should use the WHERE clause to filter the records and fetching only the necessary records. The WHERE clause is not only used in the SELECT statement, but it is also used in the DELETE statement, etc. Similarly, there are some more clauses which are BETWEEN, TO, etc.

4.5 Predicate

In AGQL, Predicates defines a logical condition being applied to rows in a table. AGQL predicates are found on the tail end of clauses. It is an expression that evaluates to TRUE, FALSE, or UNKNOWN.

4.5.1 Relationship Predicate

The predicate that is responsible for whether a relationship holds between two instances. For example: (x:host memberOf nd1:network-domain) is a relationship predicate for the relation type 'memberOf'. We have used this in our select queries and many other queries which uses or checks the existence of an instance or definition itself.

4.5.2 Attribute Predicate

The predicate that is responsible for whether value for an attribute of an entity instance satisfies some relation. For example: (x.host_Id >= 3), (x.ipAddr == "192.168.128.1") are attribute predicates for the entity type 'host'. We have used this in our select queries and many other queries which uses or checks the existence of an instance or definition itself.

4.5.3 Path Predicate

The predicate that is responsible for whether a network path exists between two hosts or not. This predicate is so far planned to be used inside the queries and is yet to introduce in our language as an extend characteristics to use.

4.6 Data Type

As of now, we have considered three data types string, int, time in our query language.

4.5.1 String

This data type in AGQL stores a set of characters within double quotes. It is not restricted with any length size.

4.5.1 Integer

This data type in AGQL holds decimal value that are not in fraction (i.e. a whole number). Its range is -2^{32} to $2^{32}-1$ or -2^{64} to $2^{64}-1$ Bytes depending on the processor of the computer.

4.5.1 Time

This data type holds a set of digits that are of String type in a specified format – <hh:mm:ss>, where hh means 'hours hand time', mm means 'minute hand time', ss means 'second hand time'.

4.7 Variable

As we know a data type is always associated with a variable in definition or every variable must be defined by a proper data type given above. The variable is said to have a valid if it holds the following conditions:

- Starting letter of the variable is any capital English letter alphabet or small English letter alphabet.
- Rest letters of the variable can be any combination of capital English letter alphabet, small English letter alphabet, numeric digits or hyphen (-) or underscore (_).

Chapter 5

RDBMS Backend Design

An attack graph describes all possible sequences of exploits an attacker can follow to advance an intrusion. The attack graph is generated from those inputs as relational views. Also typical analyses of the attack graph can be realized as relational queries against the views. This approach eliminates the needs for developing a proprietary algorithm for each different analysis, because an analysis is now simply a relational query. We have PostgreSQL for our RDBMS Backend Design. This chapter presents a relational model for representing necessary inputs including network configuration and domain knowledge.

5.1 Table Schema

The table schema for AGQL are written below, which consists of name, description, identity column, attributes/ fields and the complete description of the attributes.

5.1.1 Entity Type

Every information about the related query for entity definition will be checked and is responsible for storing the entity definitions in Entity Type Def Table Schema.

Name		ENTITY_TYPE_DEF				
Description		Stores Entity Type Definitions				
Identity Column		{ Entity_Type_Id }				
Attributes/Fields		{ Entity_Type_Name, Entity_Type_Attr_Def_List, Entity_Type_Unique_Attr_List }				
Sl.	**Field Name**	**Description**	**Size**	**Type**	**Optional/ Mandatory?**	**Constraints**
1	Entity_Type_Id	ID of the Entity Type	4B	INTEGER	Mandatory	Primary Key
2	Entity_Type_Name	Name of the Entity Type	∞	TEXT	Mandatory	
3	Entity_Type_Attr_Def_List	Attribute Definition List of the Entity Type	∞	TEXT	Mandatory	
4	Entity_Type_Unique_Attr_List	Attribute List for defining Unique Constraint(s) of the Entity Type	∞	TEXT	Optional	

Table 5-1 Table Schema for Entity Type Definition

5.1.2 Relation Type

Every information about the related query for relation definition will be checked and is responsible for storing the relation definitions in Relation Type Def Table Schema.

Name		RELATION_TYPE_DEF				
Description		Stores Relation Type Definitions				
Identity Column		{ Relation_Type_Id }				
Attributes/Fields		{ Relation_Type_Name, From_Entity_Type_Name, To_Entity_Type_Name, Relation_Type_Attr_Def_List , Relation_Type_Cardinality_Constraint }				
S l.	**Field Name**	**Description**	**Size**	**Type**	**Optional/ Mandatory?**	**Constraint s**
1	Relation_Type_Id	ID of the Relation Type	4B	INTEGER	Mandatory	Primary Key
2	Relation_Type_Name	Name of the Relation Type	∞	TEXT	Mandatory	
3	From_Entity_Type_Name	Source Entity Type Name	∞	TEXT	Mandatory	
4	To_Entity_Type_Name	Target Entity Type Name	∞	TEXT	Mandatory	
5	Relation_Type_Attr_Def_List	Attribute Definition List of the Relation Type	∞	TEXT	Mandatory	
6	Relation_Type_Cardinality_ Constraint	Attribute List for defining Cardinality Constraint(s) of the Relation Type	∞	INTEGER : INTEGER	Optional	

Table 5-2 Table Schema for Relation Type Definition

5.1.3 Security Condition Type

Every information about the related query for security condition type definition will be checked and is responsible for storing in Security Condition Type Def Table Schema.

Name		SECURITY_CONDITION_TYPE_DEF				
Description		Stores Security Condition Type Definitions				
Identity Column		{ Security_Condition_Type_Id }				
Attributes/Fields		{ Security_Condition_Type_Name, Security_Condition_Type_Attr_Def_List, Entity_Relation_List }				
S l.	**Field Name**	**Description**	**Size**	**Type**	**Optional/ Mandatory?**	**Constraint s**
1	Security_Condition_Type_Id	ID of the Security Condition Type	4B	INTEGER	Mandatory	Primary Key
2	Security_Condition_Type_N ame	Name of the Security Condition Type	∞	TEXT	Mandatory	
3	Security_Condition_Type_At tr_Def_List	Attribute Definition List of the Security Condition Type	∞	TEXT	Mandatory	
4	Entity_Relation_List	List of Security Conditions consisting of Entity Names and Relation Type Names	∞	TEXT	Mandatory	

Table 5-3 Table Schema for Security Condition Type Definition

5.1.4 Exploit Type

Before creating any exploit we must define it. Every information about the related query for exploit type definition will be checked and is responsible for storing the exploit type definitions in Exploit Type Def Table Schema.

Name		EXPLOIT_TYPE_DEF				
Description		Stores Exploit Type Definitions				
Identity Column		{ Exploit_Type_Id }				
Attributes/Fields		{ Exploit_Type_Name, CVE(s), Exploit_Type_Attr_Def_List, Precond_Entity_Type_Name, Postcond_Entity_Type_Name }				
Sl.	**Field Name**	**Description**	**Size**	**Type**	**Optional/ Mandatory?**	**Constraints**
1	Exploit_Type_Id	ID of the Exploit Type	4B	INTEGER	Mandatory	Primary Key
2	Exploit_Type_Name	Name of the Exploit Type	∞	TEXT	Mandatory	
3	CVE(s)	List of CVE(Id's)	∞	TEXT	Mandatory	
4	Exploit_Type_Attr_Def_List	Attribute Definition List of the Exploit Type	∞	TEXT	Mandatory	
5	Precond_Entity_Type_Name	List of Pre Conditions consisting of Entity Type Name(s)	∞	TEXT	Mandatory	
6	Postcond_Entity_Type_Name	List of Post Conditions consisting of Entity Type Name(s)	∞	TEXT	Mandatory	

Table 5-4 Table Schema for Exploit Type Definition

Chapter 6

GraphDB Backend Design

Attack graphs can help to harden a network at the least cost through finding critical vulnerabilities whose removal can prevent potential attacks. By providing the context of attacks, an attack graph can reveal threats in a more meaningful way compared to isolated vulnerabilities. Attack graphs are used to monitor and predict intrusions for real-time attack responses. Attack graphs may also be used as a basis for designing network security metrics. The delay is usually unacceptable due to rapidly changing needs in defending against network intrusions. This chapter basically focuses on the concept that is required to develop the backend design using Graph database. We have taken the popular open source software for graph database i.e. Neo4j [14] for storing graph information.

6.1 Concept

In our approach we have planned to model network configuration information as graph data [17] also and the domain knowledge is encoded as graph patterns. Graph queries are used to look for existence of such patterns over the graph data representing network configuration information. Results of those queries provide information about which vulnerabilities can be exploited based on the present network configuration information [18]. The exploitation of such vulnerabilities may generate new network conditions which we also model as graph patterns i.e. set of new nodes and edges, which are added to the existing graph data. The graph model consists of set of nodes *V* and set of edges *E*. Nodes can be used to represent entities V_{Entity}, Security Conditions $V_{SecurityCondition}$, Exploits $V_{Exploit}$. Therefore we can say, $V = V_{Entity} \cup V_{SecurityCondition} \cup V_{Exploit}$. Edges $E_{Relation}$ represent relations between entities. Later, we may introduce relations between security conditions and relation between exploits. Each type of entity may have any number of properties as key-value pairs which uniquely describe those entities.

6.1.1 Graph Data Model

The graph data corresponding to the simple example network (shown in Figure 6-1). The *sshd* service has a vulnerability CVE-2002-0640 which allows remote attacker to gain user privilege on the host running the service. *sshd*(1) and *sshd*(2) are the service instance fact nodes representing the facts that there are two instances of *sshd* service running at *host*1 and *host*2 respectively. The service access instance fact nodes *sshd*(3, 1), *sshd*(3, 2), *sshd*(1,2) represent facts about different hosts from which those service instances can be accessed. The goal instance node *user*(3) means that the attacker has user privilege at *host*3 and this is the only privilege the attacker initially has.

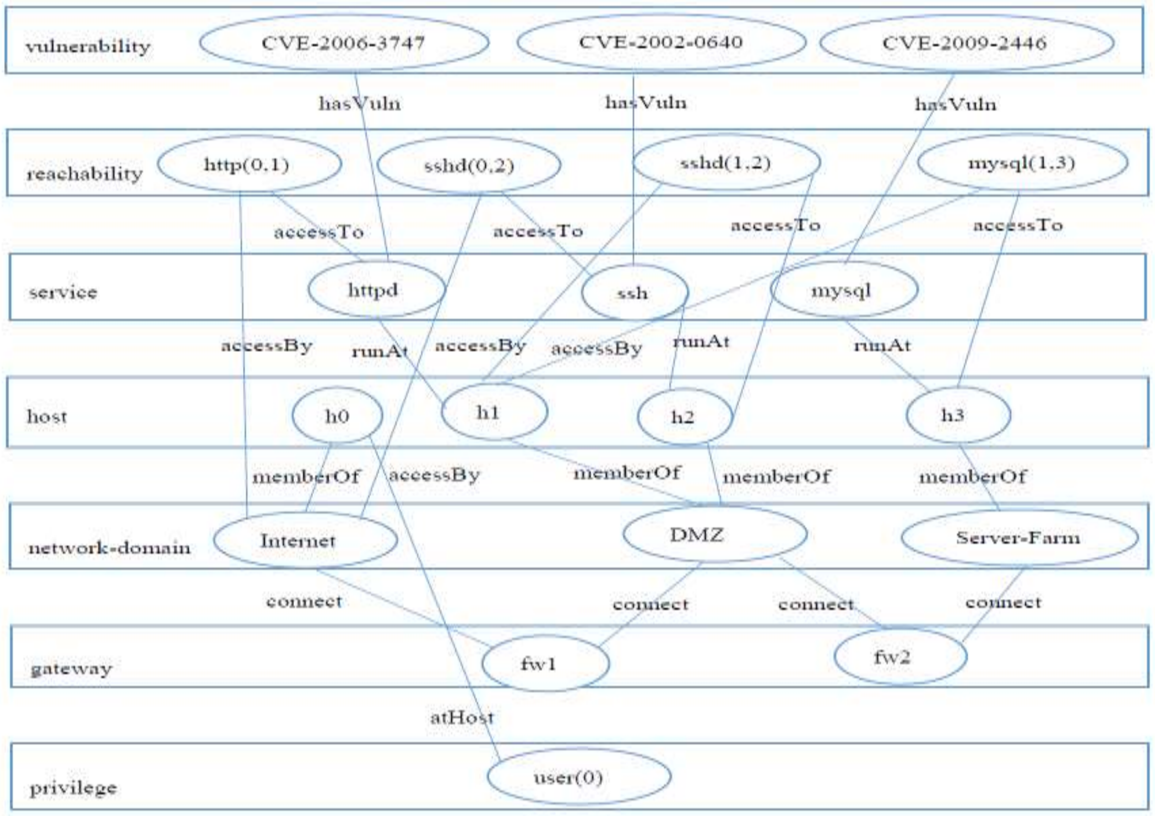


Figure 6-1 Data representing Simple Network Configuration

The generated attack graph is stored in the same graph database, with direct links to input information for each. Our approach is advantageous compared to approaches based on relational databases where large table joins are necessary for determining network conditions that enables an attacker to execute exploits. Another key benefit is that Cypher graph query language has rich support for path queries. Most of the attack graph based analysis tasks can be easily implemented using Cypher path queries.

Chapter 7

Methodology

We have implemented our interpreter using java compiler compiler (javacc). This chapter portrays the entire implementations with the algorithms running in the back end.

7.1 Flowchart

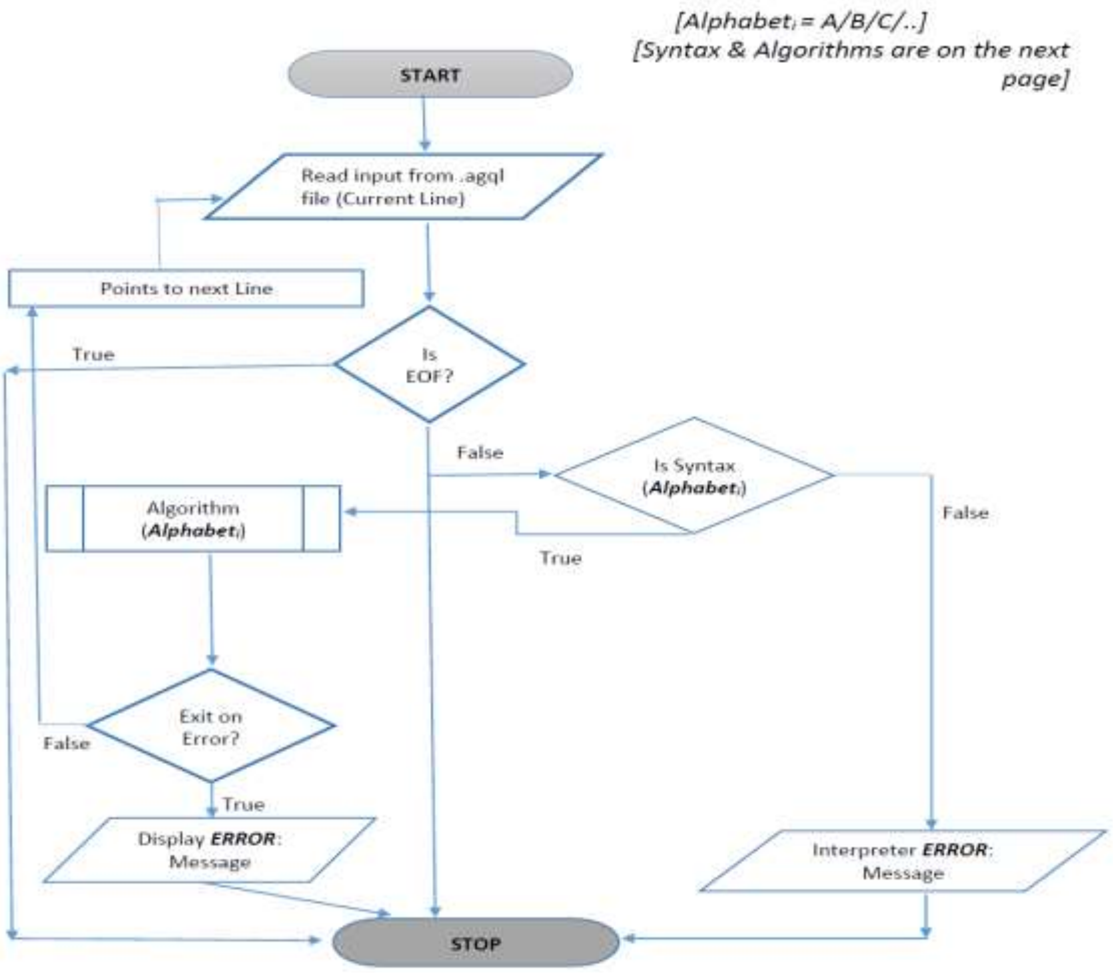


Figure 7-1 Flowchart of the Interpreter

7.2 E-R Diagram

An Entity-Relationship diagram describes interrelated things of interest in a specific domain of knowledge. A basic ER model is composed of entity types (which classify the things of interest) and specifies relationships that can exists between entities (instances of those entity types). Below is the ER diagram of our model. It is generated with the help of popular website [15], after we added our data's present in the database as input.

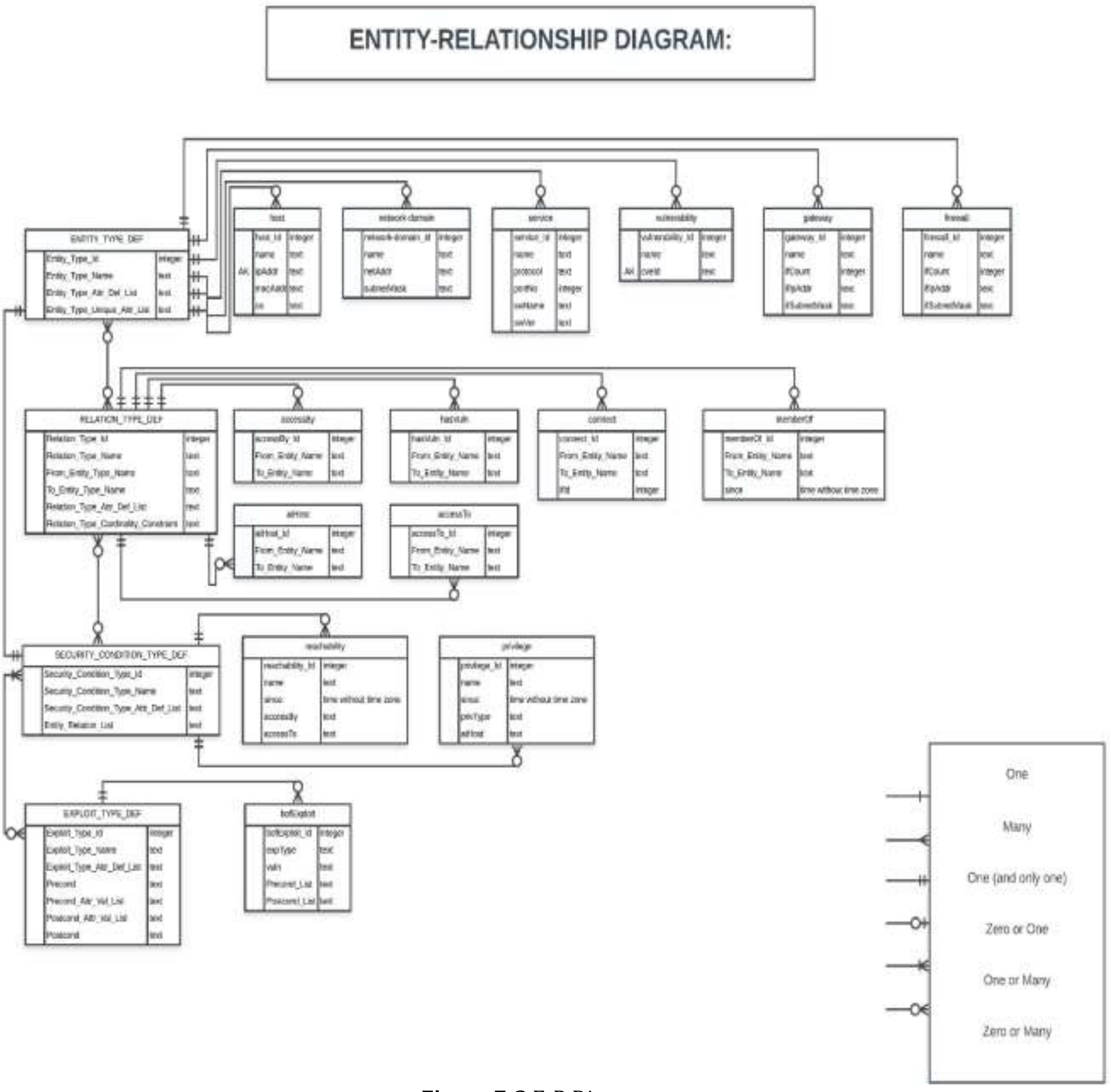


Figure 7-2 E-R Diagram

7.3 Class Diagram

A class diagram is a type of static structure diagram that describes the structure of a system's classes, their attributes, their operations (or methods), and the relationships among objects. The class diagram given below (figure 7-3) of our backend is generated from tools of eclipse, which are available in [16].

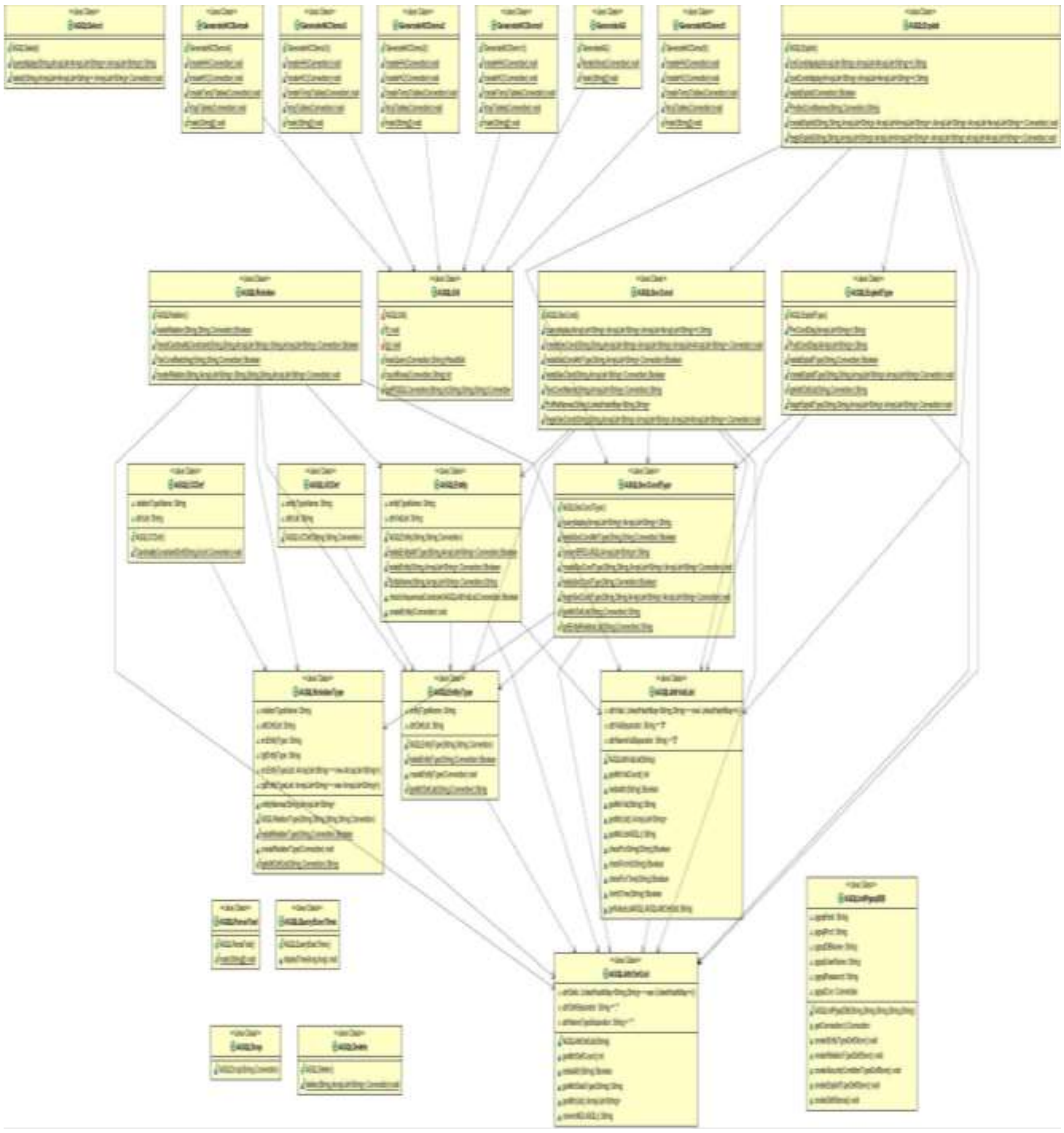


Figure 7-3 Class Diagram

7.4 Algorithms

Algorithm is an unambiguous specification of how to solve a class of problems. Therefore the backend algorithms running behind for each successful query is listed below one by one. Each example of query is taken and given as input in any certain time, then the output by the interpreter is given within box.

7.4.1 Entity Type Definition

Input: AGQL Entity Type Definition Query
Output: An Entity Type is defined in relational database.

```
1  Check whether an entry with value in <Entity_Type_Name> column in ENTITY_TYPE_DEF
   matches with <Entity_Type_Name>?
2  If
     Exists:  Then display: "ERROR: <Entity Type> is already defined". Stop.
3  Else:
              Insert an entry into the ENTITY_TYPE_DEF with Column value for
              Entity_Type_Unique_Attr_List to be empty.
              Create an Entity Type naming as <Entity_Type_Name> where number of columns are
              equal to the number of attributes in <Attr_Def_List> and a <Entity_Type_Name>_id
              column is appended.
```

Example:
If query is ***define entity-type host (name:string, ipAddr:string, macAddr:string, os:string)***

```
Encountered: define entity-type host (name:string, ipAddr:string, macAddr:string, os:string)
1 tuple added, Values: [(host) , (name:string, ipAddr:string, macAddr:string, os:string) , ()]
inserted in 'ENTITY_TYPE_DEF'
Entity Type 'host' created successfully
Query Processing Time: 72.64 ms.
```

7.4.2 Relation Type Definition

Input: AGQL Relation Type Definition Query
Output: A Relation Type is defined in relational database.

```
1  Check whether an entry with value in <Relation_Type_Name> column matches with
   <Relation_Type_Name> in RELATION_TYPE_DEF?
2  If
     Exists:   Then display: "ERROR: <Relation Type> is already defined". Stop.
3  Else:
               Insert an entry into the RELATION_TYPE_DEF with value in
               Relation_Type_Cardinality_Constraint column as empty and two more columns namely
               <From_Entity_Type_Name> and <To_Entity_Type_Name> are appended to it.

               Create a Relation Type with name as that of <Relation_Type_Name> with number of
               columns equal to the number of attributes in <Attr_Def_List> and a column with name
               <Relation_Type_Name>_id and two more columns namely <From_Entity_Type_Name>
               and <To_Entity_Type_Name> are appended to it.
```

Example:
If the query is ***define relation-type memberOf (since:time) between {host,firewall}, {network-domain}***

```
Encountered: define relation-type memberOf (since:time) between {host,firewall}, {network-domain}
1 tuple added, Values: [(memberOf) , (host,firewall) , (network-domain), (since:time), ()] inserted in 'ENTITY_TYPE_DEF'
Relation Type 'memberOf' created successfully
Query Processing Time: 73.91 ms.
```

7.4.3 Security Condition Type Definition

Input: AGQL Security Condition Type Definition Query
Output: A Security Condition Type is defined in relational database.

```
1   Check whether an entry with value in Security_Condition_Type_Name column in
    SECURITY_CONDITION_TYPE_DEF matches with <Security_Condition_Type_Name>?
2   If
    Exists:  Then Display "ERROR: <Security Condition Type> already defined". Stop.
3   Else:
4            While (i<n)         [i=0 (initialization); n is total number of
                                 Relation_Type_Names present in definition]
             Check whether Relation Type <Relation_Type_Name (i)> exits?
5            If Not
             Exists:  Then Display "ERROR: <Relation_Type_Name (i)> does not exists". Stop.
6            Else:
7                     While (j<m)      [j=0 (initialization); m is total number of Entity Names
                                       present in definition with respect to i]
                      Check whether Entity Type <Entity_Type_Name (i)_(j)> exits?
8                     If Not
                      Exists:  Then Display "ERROR: <Entity_Type_Name_(i)_(j)> does not exists".
                               Stop.
9                     Else:
                               Increment j.  Goto Line 7.
                      End of while.
             Increment i. Goto Line 4.

             End of while.

10           Insert an entry into the SECURITY_CONDITION_TYPE_DEF with two more columns
             namely <From_Entity_Type_Name> and <To_Entity_Type_Name> are appended to it.

11           Create a security-condition type with name as that of <Security_Condition_Type> with
             number of columns equal to the number of attributes in <Attr_Def_List> and a column
             with name <Security_Condition_Type>_id column is appended to it.
```

Example:

If the query is ***define security-condition-type reachability (name:string, since:time, rchType:string) accessBy {network-domain,host} accessTo {service}***

```
Encountered: define security-condition-type reachability (name:string, since:time,
rchType:string) accessBy {network-domain,host} accessTo {service}
1 Entity_Type dropped: [reachability]
1 tuple added Values: [(reachability) , (name:string, since:time, rchType:string)] inserted in
'SECURITY_CONDITION_TYPE_DEF'
Security Condition Type 'reachability' created successfully
Query Processing Time: 110.88 ms.
```

7.4.4 Exploit Type Definition

Input: AGQL Exploit Type Definition Query.
Output: An Exploit Type is defined in relational database.

```
1   Check whether an entry with value in Exploit_Type_Name column in EXPLOIT_TYPE_DEF matches
    with <Exploit_Type_Name>?
2   If
    Exists: Then Display "ERROR: <Exploit Type> already defined". Stop.
3   Else:
4       While (i<n)   [i=0 (initialization); n is total number of
                      Security_Condition_Type_Name present in pre-condition]
            Check whether Security Condition Type <Security_Condition_Type_Name (j)> exits?
5           If Not
            Exists:  Display "ERROR: <Security_Condition_Type_Name (j)> does not exists". Stop
6           Else:
                Check whether <Attr_Val_List(i)> after <Security_Condition_Type_Name(i)>
                is present in precondition & whether attr(s) in <Attr_Val_List (i)> is present in
                <Security_Condition_Type_Attr_Def_List> of
                SECURITY_CONDITION_TYPE_DEF?
7               If Not
                Exists:  Display the attr(s) in <Attr_Val_List (i)> that are not present during the
                         security condition definition. Stop.
8               Else:
                         Increment i.  Goto Line 4.
            End of while.

9       While (j<m)   [j=0 (initialization); m is total number of
                      Security_Condition_Type_Name present in post-condition]
            Check whether Security Condition Type <Security_Condition_Type_Name (j)> exits?
10          If Not
            Exists:  Display "ERROR: <Security_Condition_Type_Name (j)> does not exists". Stop.
11          Else:
                Check whether <Attr_Val_List(j)> after <Security_Condition_Type_Name(j)>
                is present in precondition & whether attr(s) in <Attr_Val_List (j)> is present in
                <Security_Condition_Type_Attr_Def_List> of
                SECURITY_CONDITION_TYPE_DEF?
12              If Not
                Exists:  Display the attr(s) in <Attr_Val_List (j)> that are not present during the
                         security condition definition. Stop.
13              Else:
                         Increment j.  Goto Line 9.
            End of while.

14      Insert a tuple of values extracted from the query for <Exploit_Type_Id,
        Exploit_Type_Name, CveId(s), Exploit_Type_Attr_Def_List, Precond (names),
        Precond_Attr_Val_List, Postcond (names), Postcond_Attr_Val_List> in
        EXPLOIT_TYPE_DEF.

15      Create a exploit type with name as <Exploit_Type> with number of columns equal to the
        number of attributes in <Attr_Def_List> and a <Exploit_Type>_id column, Precond_List
        column, Postcond_List column are appended to it.
```

Example:

If the query is ***define exploit-type bofExploit (name:string, category:string) CVE ("CVE-2008-0106") precond {privilege(privType:"user"), reachability(rchType:"xyz")} postcond {privilege(privType:"user")}***

```
Encountered: define exploit-type bofExploit (name:string, category:string) CVE ("CVE-2008-0106") precond {privilege($privType$:#"user"#), reachability($rchType$:#"xyz"#)} postcond {privilege($privType$:#"user"#)}
1 tuple added Values: [(bofExploit) , ("CVE-2008-0106") , (name:string, category:string)] inserted in 'EXPLOIT_TYPE_DEF'
'bofExploit' created successfully
Query Processing Time: 77.55 ms.
```

7.4.5 Unique Constraint Definition

Input: AGQL Uniqueness Constraint Definition Query
Output: Uniqueness constraint is defined in relational database.

```
1  Check whether an Entity <Entity_Type_Name> exists in ENTITY_TYPE_DEF?
2  If
     Exists:  Check whether attr(s) in <Attr_List> is present in <Entity_Type_Attr_Def_List> in
              ENTITY_TYPE_DEF?
              If
                Present:  Update entry (if it's empty) for <Entity_Type_Name> in ENTITY_TYPE_DEF with
                          value in <Entity_Type_Unique_Attr_List> column as attr(s) that are present in
                          <Attr_List>.
              Else:
                          Display the attr(s) in <Attr_List> that are not present. Stop.
3  Else:
              Display "ERROR: <Entity Type> doesn't exists". Stop.
4  Now, According to the Values present in <Entity_Type_Unique_Attr_List> of the
   <Entity_Type_Name> if not empty.
5  Set those column(s) of Entity <Entity_Type_Name> as UNIQUE attributes.
```

Example:

If the query is ***define unique vulnerability (cveId)***

```
Encountered: define unique vulnerability (cveId)
'ENTITY_TYPE_DEF' updated successfully.
Uniqueness Constraint is assigned to Entity Type :: vulnerability
Query Processing Time: 77.56 ms.
```

7.4.6 Cardinality Constraint Definition

Input: AGQL Cardinality Constraint Definition Query
Output: Cardinality constraint is defined in relational database.

```
1  Check whether Relation <Relation_Type_Name> exists in RELATION_TYPE_DEF?
2  If Not
   Exists:  Then display: "ERROR: <Relation Type> is not defined". Stop.
3  Else:
            Update that entry (only if it's empty) for <Relation_Type_Name> in
            RELATION_TYPE_DEF with value in <Relation_Type_Cardinality_Constraint>
            column as <Cardinality_Number1> : <Cardinality_Number2>.
```

Example:
If the query is ***define cardinality memberOf n:1***

```
Encountered: define cardinality memberOf n:1
'RELATION_TYPE_DEF' updated successfully.
Cardinality Constraint is assigned to Relation Type :: memberOf
Query Processing Time: 11.13 ms.
```

7.4.7 Drop

Input: AGQL Drop Query
Output: Target definition is dropped from relational database.

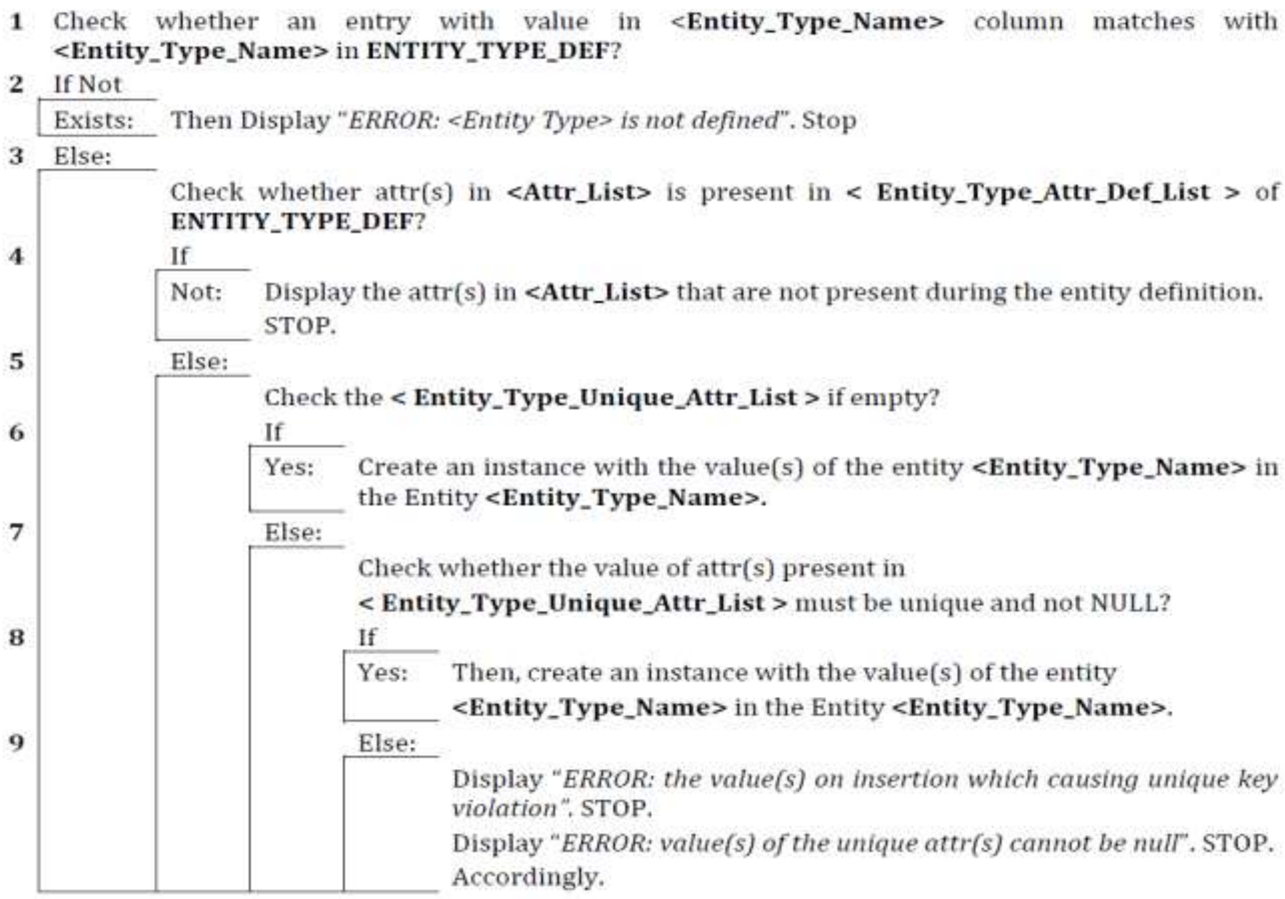
```
1  Check whether <Type> exists?
2  If
   Exists:  Extract <Type_Name (i)> [i = 1 to n]... one by one as <Type_Name> and Check whether
            <Type_Name> exists in <Type>?
3           If
            Exists:  If <Type> belongs to <ENTITY_TYPE_DEF>:
                         Check <Type_Name (i)> also belongs to
                         <SECURITY_CONDITION_TYPE_DEF>?
                         If
                         Exists:  Delete that <Type_Name (i)> instance from <Type>.
                     Else:
                                  Delete that <Type_Name (i)> instance from <Type>.
                                  Delete the <Type> named as <Type_Name (i)> that
                                  exists separately.
4           Else:
                     Display "ERROR: <Type_Name> doesn't exists". Stop.
5  Else:
            Display "ERROR: <Type> doesn't exists". Stop.
```

Example:
If the query is ***drop entity-type (host, network-domain)***

```
Encountered: drop entity-type (host, network-domain)
1 tuple(s) deleted Values: ["Entity_Type_Name" = 'host']
1 deleted: [host]
WARNING.AGQLDrop: "Entity_Type_Name" = 'network-domain' Does not exist.
Query Processing Time: 6.54 ms.
```

7.4.8 Entity Creation

Input: AGQL Entity Creation Query
Output: An Instance of Entity Type is created in relational database.



```
1 Check whether an entry with value in <Entity_Type_Name> column matches with
  <Entity_Type_Name> in ENTITY_TYPE_DEF?
2 If Not
  Exists: Then Display "ERROR: <Entity Type> is not defined". Stop
3 Else:
          Check whether attr(s) in <Attr_List> is present in < Entity_Type_Attr_Def_List > of
          ENTITY_TYPE_DEF?
4         If
          Not:  Display the attr(s) in <Attr_List> that are not present during the entity definition.
                STOP.
5         Else:
                Check the < Entity_Type_Unique_Attr_List > if empty?
6               If
                Yes:  Create an instance with the value(s) of the entity <Entity_Type_Name> in
                      the Entity <Entity_Type_Name>.
7               Else:
                      Check whether the value of attr(s) present in
                      < Entity_Type_Unique_Attr_List > must be unique and not NULL?
8                     If
                      Yes:  Then, create an instance with the value(s) of the entity
                            <Entity_Type_Name> in the Entity <Entity_Type_Name>.
9                     Else:
                            Display "ERROR: the value(s) on insertion which causing unique key
                            violation". STOP.
                            Display "ERROR: value(s) of the unique attr(s) cannot be null". STOP.
                            Accordingly.
```

Example:
If the query is ***create entity host (name:"h1", ipAddr:"192.168.148.1", macAddr: "xx:xx:xx:xx:xx:xx", os:"Ubuntu")***

```
Encountered: create entity host ($name$:#"h1"#, $ipAddr$:#"192.168.148.1"#,
$macAddr$:#"xx:xx:xx:xx:xx:xx"#, $os$:#"Ubuntu"#)
1 tuple added Values: [("name","ipAddr","macAddr","os") , ('h1', '192.168.148.1',
'xx:xx:xx:xx:xx:xx', 'Ubuntu')] inserted in 'host'
Query Processing Time: 10.88 ms.
```

7.4.9 Relation Creation

Input: AGQL Relation Creation Query
Output: An instance of Relation Type is created in relational database.

```
1   Check whether an entry with value in Relation_Type_Name column matches with
    <Relation_Type_Name> in RELATION_TYPE_DEF?
2   If Not
    Exists: Then Display "ERROR: <Relation Type> is not defined". Stop.
3   Else:
            Check whether attr(s) in <Attr_List> is present in < Relation_Type_Attr_Def_List > of
            RELATION_TYPE_DEF?
4           If
            Not: Display the attr(s) in <Attr_List> that are not present during the relation
                 definition. STOP.
5           Else:
                 Check the < Relation_Type_Cardinality_Constraint > if not empty?
6                If
                 Yes: Create an instance with the value(s) of the entity <Relation_Type_Name>
                      in the Relation <Relation_Type_Name>.
7                Else:
                      Check whether the variables <Variable_1_A> matches with that of
                      <Variable_1_B> and <Variable_2_A> matches with that of
                      <Variable_2_B>?
8                     If
                      No: Then Display "ERROR: Variables doesn't match". Stop.

                      Check whether the source entity of entity type <Entity_Type_Name_1>
                      satisfying the condition <Entity_Condition_1> exists?
9                     If
                      No: Then Display "ERROR: there is no such entity". Stop.

                      Check whether the target entity of entity type <Entity_Type_Name_2>
                      satisfying the condition <Entity_Condition_2> exists?
10                    If
                      No: Then Display "ERROR: there is no such entity". Stop

                      Check the source entity type <Entity_Type_Name_1> and the target entity
                      type <Entity_Type_Name_2> exceeds the cardinality constraint attribute
                      mentioned under < Relation_Type_Cardinality_Constraint >?
11                    If
                      Yes:  Display "ERROR: Violating cardinality constraint". Stop.
12                    Else: Create an instance with the value(s) of the relation
                            <Relation_Type_Name> in the Relation <Relation_Type_Name>.
```

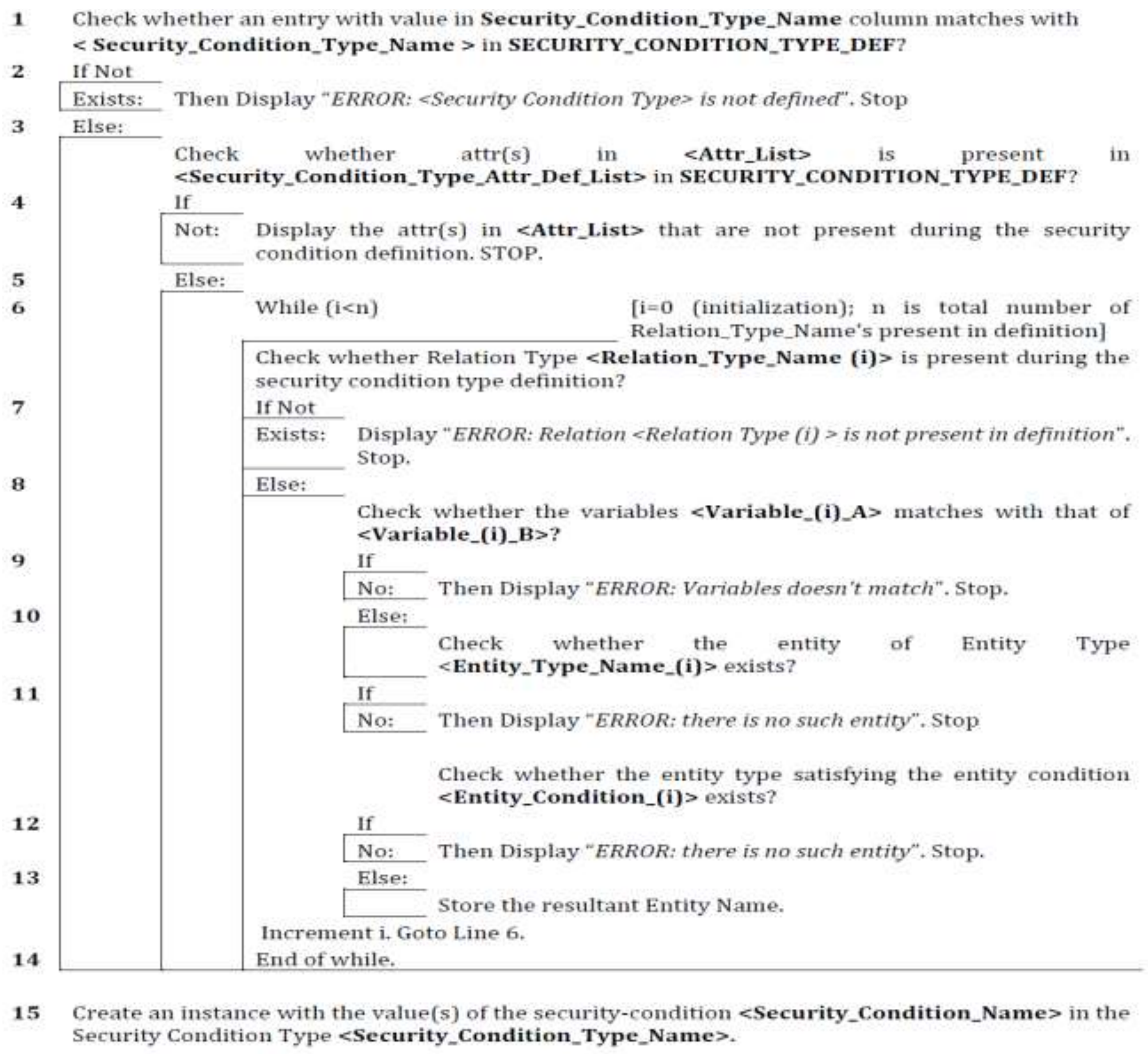
Example:

If the query is ***create relation x:host (x.ipAddr=="192.168.148.2") memberOf (since:"10:11:37") y:network-domain (y.netAddr=="192.168.148.0")***

```
Encountered: create relation x:host (x.ipAddr=="192.168.148.2") memberOf
($since$:#"10:11:37"#) y:network-domain (y.netAddr=="192.168.148.0")
1 tuple(s) added Values: [("since") , ('10:11:37')] inserted in 'memberOf'
Query Processing Time: 21.99 ms.
```

7.4.10 Security Condition Creation

Input: AGQL Security Condition Creation Query
Output: A Security Condition is created in relational database.



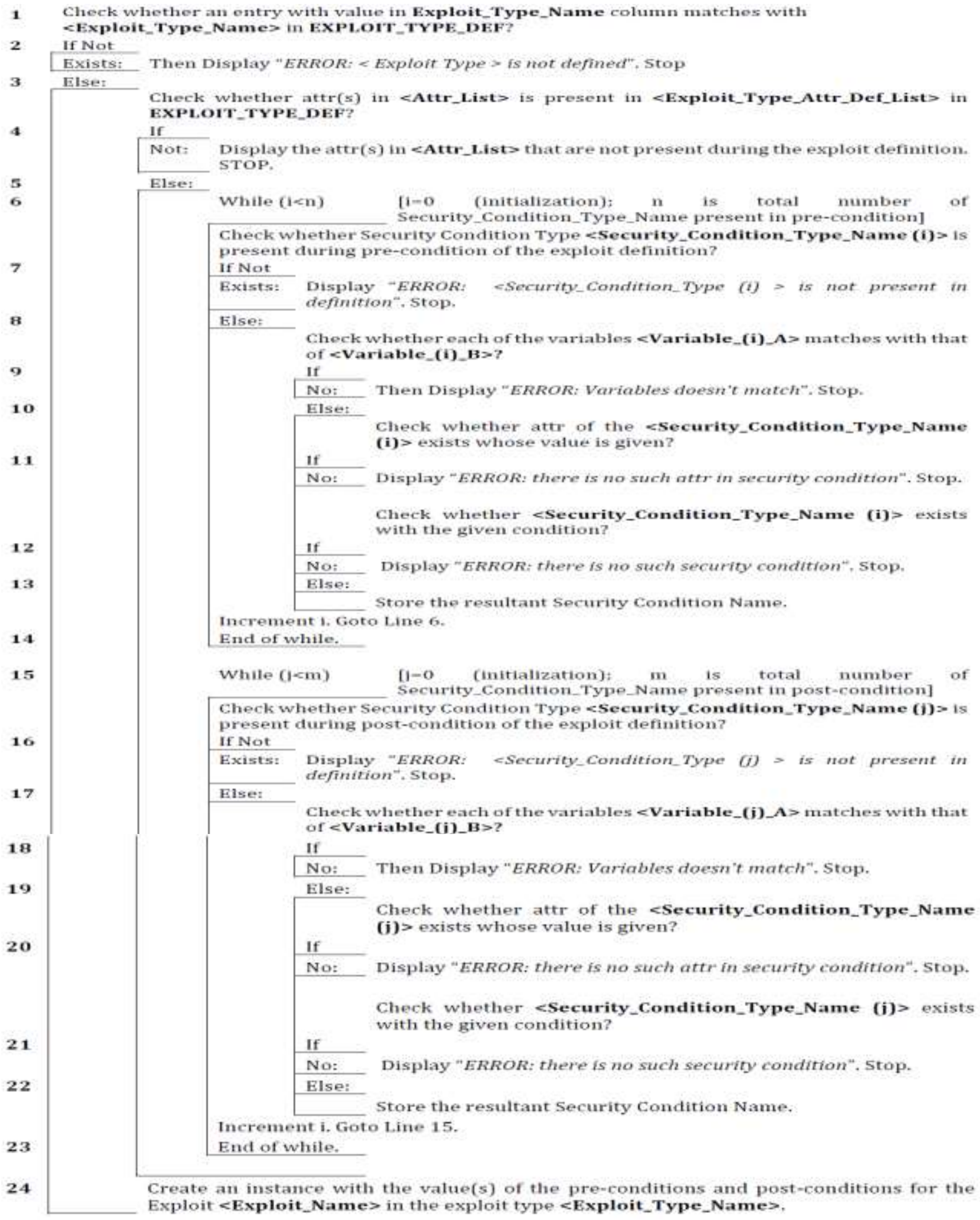
```
1   Check whether an entry with value in Security_Condition_Type_Name column matches with
    < Security_Condition_Type_Name > in SECURITY_CONDITION_TYPE_DEF?
2   If Not
    Exists:  Then Display "ERROR: <Security Condition Type> is not defined". Stop
3   Else:
             Check whether attr(s) in <Attr_List> is present in
             <Security_Condition_Type_Attr_Def_List> in SECURITY_CONDITION_TYPE_DEF?
4            If
             Not:  Display the attr(s) in <Attr_List> that are not present during the security
                   condition definition. STOP.
5            Else:
6                  While (i<n)                 [i=0 (initialization); n is total number of
                                               Relation_Type_Name's present in definition]
                   Check whether Relation Type <Relation_Type_Name (i)> is present during the
                   security condition type definition?
7                  If Not
                   Exists:  Display "ERROR: Relation <Relation Type (i) > is not present in definition".
                            Stop.
8                  Else:
                            Check whether the variables <Variable_(i)_A> matches with that of
                            <Variable_(i)_B>?
9                           If
                            No:    Then Display "ERROR: Variables doesn't match". Stop.
10                          Else:
                                   Check whether the entity of Entity Type
                                   <Entity_Type_Name_(i)> exists?
11                          If
                            No:    Then Display "ERROR: there is no such entity". Stop

                                   Check whether the entity type satisfying the entity condition
                                   <Entity_Condition_(i)> exists?
12                          If
                            No:    Then Display "ERROR: there is no such entity". Stop.
13                          Else:
                                   Store the resultant Entity Name.
                    Increment i. Goto Line 6.
14                 End of while.

15  Create an instance with the value(s) of the security-condition <Security_Condition_Name> in the
    Security Condition Type <Security_Condition_Type_Name>.
```

Example:
If the query is ***create security-condition reachability (name:"httpd12", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.1")} accessTo {y:service (y.name == "httpd2")}***

```
Encountered: create security-condition reachability ($name$:#"httpd12"#, $rchType$:
#"xyz"#) accessBy {x:host (x.ipAddr == "192.168.148.1")} accessTo {y:service (y.name ==
"httpd2")}
WARNING: No. of Attributes in Creation = [ 2 ] & Defintion = [ 3 ] are unequal
1 tuple added Values: [("name","rchType") , ('httpd12','xyz')] inserted in 'reachability'
Query Processing Time: 44.68 ms.
```

7.4.11 Exploit Creation

Input: AGQL Exploit Creation Query.
Output: An Exploit is created in relational database.



```
1   Check whether an entry with value in Exploit_Type_Name column matches with
    <Exploit_Type_Name> in EXPLOIT_TYPE_DEF?
2   If Not
    Exists:  Then Display "ERROR: < Exploit Type > is not defined". Stop
3   Else:
        Check whether attr(s) in <Attr_List> is present in <Exploit_Type_Attr_Def_List> in
        EXPLOIT_TYPE_DEF?
4       If
        Not:  Display the attr(s) in <Attr_List> that are not present during the exploit definition.
              STOP.
5       Else:
6           While (i<n)     [i=0 (initialization); n is total number of
                            Security_Condition_Type_Name present in pre-condition]
            Check whether Security Condition Type <Security_Condition_Type_Name (i)> is
            present during pre-condition of the exploit definition?
7           If Not
            Exists:  Display "ERROR: <Security_Condition_Type (i) > is not present in
                     definition". Stop.
8           Else:
                Check whether each of the variables <Variable_(i)_A> matches with that
                of <Variable_(i)_B>?
9               If
                No:  Then Display "ERROR: Variables doesn't match". Stop.
10              Else:
                    Check whether attr of the <Security_Condition_Type_Name
                    (i)> exists whose value is given?
11              If
                No:  Display "ERROR: there is no such attr in security condition". Stop.
                    Check whether <Security_Condition_Type_Name (i)> exists
                    with the given condition?
12              If
                No:  Display "ERROR: there is no such security condition". Stop.
13              Else:
                    Store the resultant Security Condition Name.
            Increment i. Goto Line 6.
14          End of while.

15          While (j<m)     [j=0 (initialization); m is total number of
                            Security_Condition_Type_Name present in post-condition]
            Check whether Security Condition Type <Security_Condition_Type_Name (j)> is
            present during post-condition of the exploit definition?
16          If Not
            Exists:  Display "ERROR: <Security_Condition_Type (j) > is not present in
                     definition". Stop.
17          Else:
                Check whether each of the variables <Variable_(j)_A> matches with that
                of <Variable_(j)_B>?
18              If
                No:  Then Display "ERROR: Variables doesn't match". Stop.
19              Else:
                    Check whether attr of the <Security_Condition_Type_Name
                    (j)> exists whose value is given?
20              If
                No:  Display "ERROR: there is no such attr in security condition". Stop.
                    Check whether <Security_Condition_Type_Name (j)> exists
                    with the given condition?
21              If
                No:  Display "ERROR: there is no such security condition". Stop.
22              Else:
                    Store the resultant Security Condition Name.
            Increment i. Goto Line 15.
23          End of while.

24      Create an instance with the value(s) of the pre-conditions and post-conditions for the
        Exploit <Exploit_Name> in the exploit type <Exploit_Type_Name>.
```

Example:

If the query is

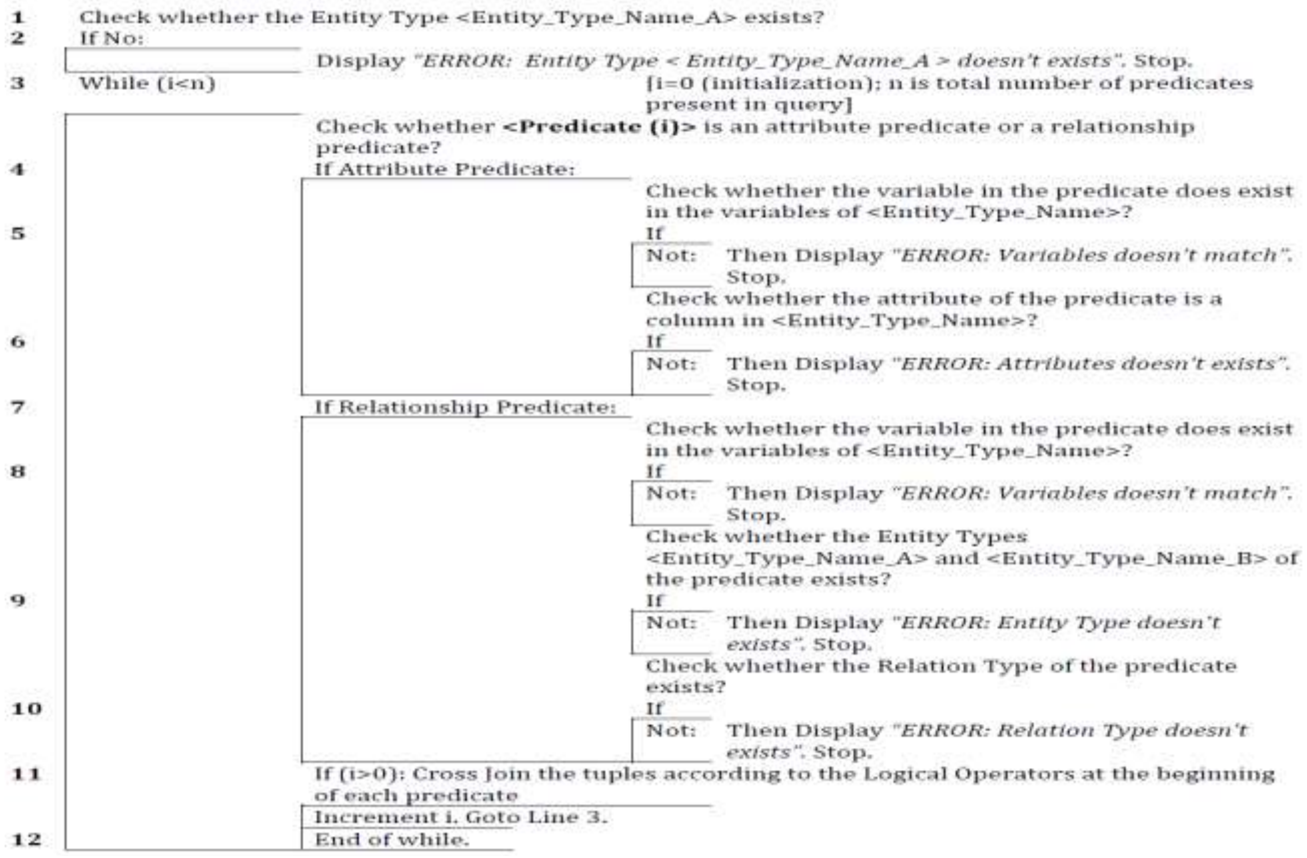
create security-condition privilege (name:"user1", privType:"user") atHost {x:host (x.ipAddr == "192.168.148.1")}

create exploit bofExploit (name:"bofExp31", category:"remote") precond {x:privilege (x.name == "user3"), y:reachability (y.name == "httpd31")} postcond {x:privilege (x.name == "user1")}

```
Encountered: create security-condition privilege ($name$:#"user1"#, $privType$:#"user"#) atHost {x:host (x.ipAddr == "192.168.148.1")}
WARNING: No. of Attributes in Creation = [ 2 ] & Defintion = [ 3 ] are unequal
1 row added Values: [("name","privType") , ('user1','user')] inserted in Table 'privilege'
Query Processing Time: 10.99 ms.
Encountered: create exploit bofExploit ($name$:#"bofExp31"#, $category$:#"remote"#) precond {x:privilege (x.name == "user3"), y:reachability (y.name == "httpd31")} postcond {x:privilege (x.name == "user1")}
1 row added Values: [("name","category") , ('bofExp31','remote') , (user3,httpd31) , (user1)] inserted in Table 'bofExploit'
Query Processing Time: 54.17 ms.
```

7.4.12 Select

Input: AGQL Select Query
Output: Items are selected from relational database and displayed in the screen.



```
1   Check whether the Entity Type <Entity_Type_Name_A> exists?
2   If No:
        Display "ERROR: Entity Type < Entity_Type_Name_A > doesn't exists". Stop.
3   While (i<n)                  [i=0 (initialization); n is total number of predicates present in query]
        Check whether <Predicate (i)> is an attribute predicate or a relationship predicate?
4       If Attribute Predicate:
            Check whether the variable in the predicate does exist in the variables of <Entity_Type_Name>?
5           If
            Not:  Then Display "ERROR: Variables doesn't match". Stop.
            Check whether the attribute of the predicate is a column in <Entity_Type_Name>?
6           If
            Not:  Then Display "ERROR: Attributes doesn't exists". Stop.
7       If Relationship Predicate:
            Check whether the variable in the predicate does exist in the variables of <Entity_Type_Name>?
8           If
            Not:  Then Display "ERROR: Variables doesn't match". Stop.
            Check whether the Entity Types <Entity_Type_Name_A> and <Entity_Type_Name_B> of the predicate exists?
9           If
            Not:  Then Display "ERROR: Entity Type doesn't exists". Stop.
            Check whether the Relation Type of the predicate exists?
10          If
            Not:  Then Display "ERROR: Relation Type doesn't exists". Stop.
11      If (i>0): Cross Join the tuples according to the Logical Operators at the beginning of each predicate
        Increment i. Goto Line 3.
12      End of while.
```

Example:

If the query is ***select x:host where (x.ipAddr == "192.168.148.3") or (x.name == "h1")***

```
Encountered: select x:host where (x.ipAddr == "192.168.148.3") or (x.name == "h1")

host_Id    name    ipAddr           macAddr              os
1          h3      192.168.148.3    xx:yy:zz:xx:xx:xx    Ubuntu
2          h1      192.168.148.1    xx:xx:xx:xx:xx:xx    Ubuntu
[2] Records from have been found
Query Processing Time: 60.09 ms.
```

7.4.13 Delete

Input: AGQL Delete Query
Output: Target is deleted from relational database.

```
1  Check whether <Type_Name> exists?
2  If
   Exists:  Check whether <Type_Condition> exists in <Type_Name>?
3           If
            Exists:  Check whether <Variable_A> matches with <Variable_B>?
4                    If
                     No:    Then Display "ERROR: Variables doesn't match". Stop.
5                    Else:
                            Delete that particular instance from <Type_Name>.
6           Else:
                     Display "ERROR: There is no such instance of the type <Type_Name>". Stop.
7  Else:
            Display "ERROR: <Type_Name> doesn't exists". Stop.
```

Example:

If the query is ***delete x:host where (x.name=="h1")***

```
Encountered: delete x:host where (x.name=="h1")
1 No. of host objects deleted
Query Processing Time: 47.77 ms.
```

7.5 Assumptions

These assumptions are taken with respect to the current status of the system and few might be changed in the later process just after assuring that the system is fundamentally correct and is ready for detailing in advance level. As per as the current implementation stands, below are the following assumptions that are considered:

- <attr = "name"> must be present for all and must be unique, later "id" will be made unique instead of "name".
- Unique constraint and cardinality constraint must be declared before creation of any instances those shall be particularly restricted with constraints.
- Entities involved as post conditions for exploit must be created before creating exploit.
- CVE ("") in exploit definition must have one parameter in string format with valid cve-id(s), whose validating are not added as of now.
- Newline is considered as terminator for any statement unless the statement stands incomplete.
- Due to internal logic associated in backend algorithms, the users are strictly prohibited to use characters like '$', '#' in the Attribute Value List while using queries for creation.

7.6 Features

7.6.1 Interactivity

The first version of AGQL is implemented as an interpreted language and ready to use as an alpha version module. The user friendly approach provides the user with immediate feedback on individual instructions, thus aiding in the debugging and learning process. Besides checking whether the syntax of the query is perfect, it also is capable of producing error messages on regard to its post-actions of the query. Error messages are kept descriptive with proper reasons and also contains the exact messaging source that is responsible for creating such errors. For example: In EntityType.java contains exact messaging source as AGQLEntityType which might be useful for debugging the source code while in testing.
Error Statement1 –"ERROR.AGQLEntityType: Entity <" + entityTypeName + "> already exists".
Error Statement2 –"ERROR.AGQLEntityType: Entity type " + entityTypeName + " is not defined".

7.6.2 Modularity and Extensibility

Extensibility and Modularity is a necessity of the designing, developing and upgrading systems that needs continuous maintenance for removal of bugs and adding extensive properties for its large scale use of the language. We have used the concept of modular programming for its benefit in small unit testing. It also helps us to focus in debugging and working independently in a particular module for any extension or modification.

7.6.3 Flexibility

Flexibility provides the design to expand and also support changes to requirements during the development phase. In our case we can call this kind of flexibility as "flexibility by

construction" because our system can automatically determine the presence of attributes in creation from its definition irrespective of position.
J. Schoenmakers mentioned in his Paper titled 'TOM: On the Flexibility of Programming Languages' that "Backward Flexibility is the flexibility to change your mind; to change, as the developer of past code, aspects of your implementation without affecting future code. Code is not affected if it does not require recompilation." In our language the Attribute list's while creation doesn't require to be a Complete Set with respect to the number of attributes defined. Here we change our mind while creation and it even doesn't require re-compilation. So, we can claim our language allows Backward Flexibility.

7.6.4 Strict Data Type Matching

Attributes are strictly checked while creation with that of its data types mentioned in its definition. At the moment the supported three data types namely; string, int and time does strict data type checking. Only valid attribute values will be allowed to store in Database.

7.6.5 Scalability

Administrative Scalability for every networks has grown huge and thereby demanding an additional feature to include in our system. Our system promises to have explored the scalable features. It can handle a growing amount of data that is obvious in todays' world by allowing to store and make use of unlimited storage in the database.

7.6.6 Computational Time Estimation per Query

The System calculates the computational time for each query. Thereby the user can enjoy this feature without any additional syntax. The time it displays it is in milliseconds. The time starts just when the query is just checked and the post-action for the syntax begins. And the time ends when the process for that query gets completed. The difference is displayed to the user. Such computations are meant to display for each and every query. It helps in research work to estimate complexity and computing time of the task associated with the query.

7.7 Exception Handling

Exception handling is the process of responding to the occurrence, during computation, of exceptions – anomalous or exceptional conditions requiring special processing often disrupting the normal flow of program execution. The syntax for the backend relational database is very crucial as it must be syntactically correct and specific. Hence, to avoid any run time exception while running the query, the codes are kept inside try and catch blocks. Common exceptions are well handled by the try and catch blocks. If any other type of error is encountered, then the interpreter halts.

7.8 Data Structures & Error Messages

Data Structures plays an important role in the interpreter. The complex data structures used in the interpreter mostly facilitates the quickness of the interpreter by reducing the complexity of the program as much as possible. In general, we have taken linked hash map for the attribute definition list and attribute value list for direct access. We have used array list in most of the cases to work in low memory available zone and also the elements can be directly accessed by its index value.

Error messages are very essential for interactive interpreter. Our interpreter shows the appropriate error messages that will benefit the client. Since, it is in the developing phase, it is kept as such that if any error is occurred, then absolute position from which block is returned to the user as output with the appropriate error messages with suggestive logics if any. In some cases there can be different semantics but due to some additional feature of the interpreter, the code gets overlooked and no longer is treated as an error. In those cases a warning message is displayed and the backend tasks gets performed successfully.

Chapter 8

Attack Graph Generation

8.1 Experimental Analysis

Instead of modeling an attack graph, we have modeled necessary inputs required for generating the attack graph. The attack graph then becomes the result of relational queries over these inputs. Such a result may be simply kept as the definition of relational views. This chapter focus on the attack graph generation using TVA Approach.

8.1.1 Attack Graph Generation

For generating attack graph two types of information i.e. network configuration and domain knowledge are necessary. The network configuration information includes network topology information, firewall (perimeter and/or host based) rule set and per host vulnerability information. The domain knowledge refers to the interdependency between different types of vulnerabilities and network conditions.

Example:
To generate the attack graph, we need the network configuration and domain knowledge shown in Figure 8-1. The left-hand side shows the connectivity between three hosts. Initially, hosts 1 and 2 satisfy the condition *x* and host 3 satisfies *y*. The right-hand side says that an attacker can exploit the vulnerability *A* on the destination (denoted by the symbol *D*) host, if it satisfies *x* and the source host satisfies *y* at the same time. This exploitation will then satisfy *y* on the destination host.

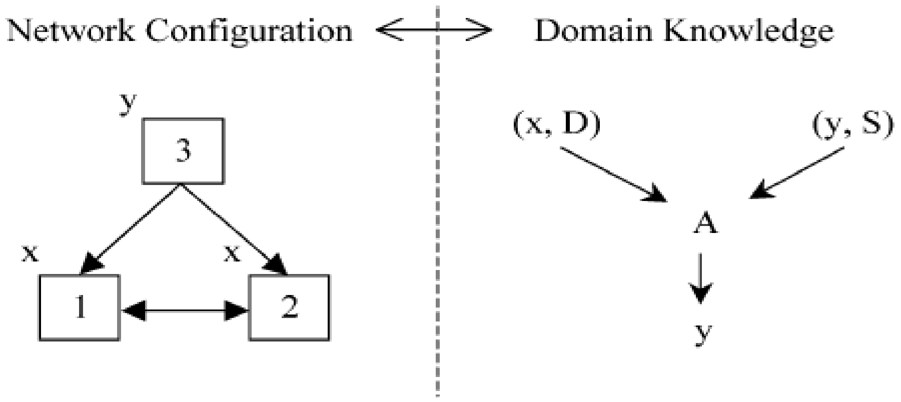


Figure 8-1 Example of network configuration and domain knowledge.

We assume the domain knowledge required for generating attack graphs is available from tools like the Topological Vulnerability Analysis (TVA) system, which covers more than 37,000 vulnerabilities taken from 24 information sources including X-Force, Bugtraq, CVE, CERT, Nessus and Snort. On the other hand, we assume the configuration information including vulnerabilities and connectivity can be obtained using available tools, such as the Nessus scanner.
The *connectivity relation* represents the connectivity from the source host H_s to the destination host H_d. The *condition relation* indicates that a host H has an initial condition C. The condition vulnerability dependency relation indicates that a condition C is required for exploiting a vulnerability V on the destination host. The attribute F indicates whether the condition C belongs to the source (S) or the destination (D) host. The vulnerability–condition dependency relation indicates a condition C is satisfied by exploiting a vulnerability V.
The next three relations and the condition relation together represent the complete attack graph. Those relations may or may not need to be materialized. The vertices are conditions (that is, the relation HC) and exploits (that is, the relation EX), and the edges interconnecting them are represented by relations CE and EC. Each relation has a composite key composed of all attributes in that relation.

The following relational schemata is given below:

- *Connectivity HH* = (H_s, H_d),
- *Condition HC* = (H,C),
- *Condition–vulnerability dependency CV* = (C, F, V),
- *Vulnerability–condition dependency V C* = (V ,C),
- *Exploit EX* = (H_s, H_d, V),
- *Condition–exploit CE* = (H, C, H_s, H_d, V),
- *Exploit–condition EC* = (H_s, H_d, V, H, C).

8.1.2 Algorithm for TVA

Input: AGQL Relational Database after running AGQL queries.
Output: TVA relations are captured.

To Populate 'HH':

```
For each tuple ($R_i$) in 'REACHABILITY'
        Check whether the Host ($H_{to}$) at which the service $R_i$.ACCESS_TO is running (by
        referring 'RUNAT')?
        If $R_i$.ACCESS_BY == HOST ($H_{from}$) then
            The tuple < getid($H_{to}$), getid($H_{from}$)> is inserted into 'HH'
        else if $R_i$.ACCESS_BY == NETWORK_DOMAIN (ND) then
            For all Host ($H_{from}$) in ND (get it from MEMBEROF table)
                The tuple <getid(h1), getid(h2)> is inserted into 'HH'
```

To Populate 'HC':

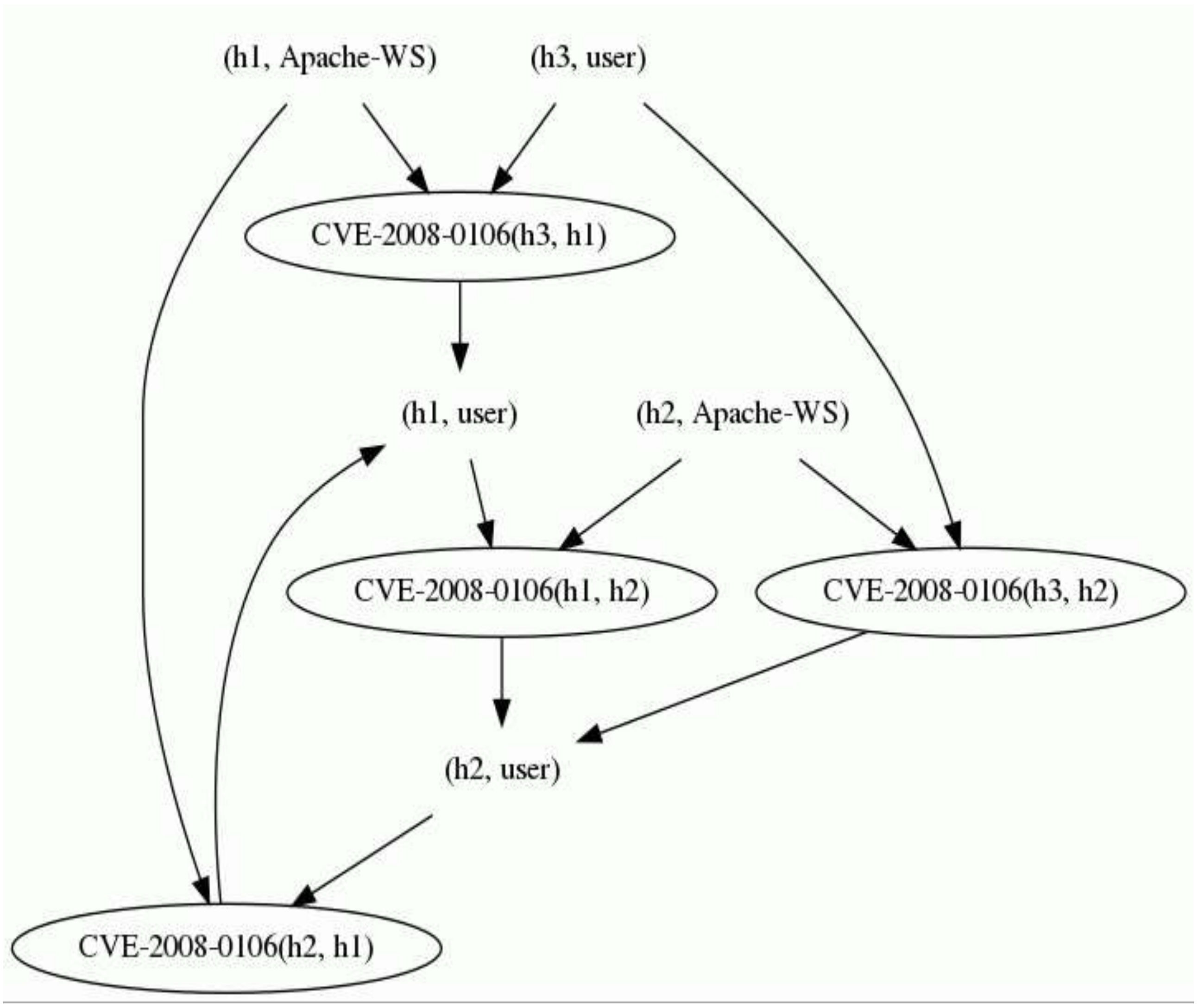
```
For all tuple ($R_i$) in 'PRIVILEGE'
        H1 = ($R_i$).ATHOST is obtained.
        SC1 = ($R_i$).privtype is obtained.
        The tuple <getid(H1), SC1> is inserted into 'HC'
For each tuple ($R_i$) in 'REACHABILITY'
        The sw_name (S1) corresponding to the service name $R_i$.ACCESS_TO (by referring
        'SERVICE') is obtained.
        The host (h2) at which the service $R_i$.ACCESS_TO is running (by referring 'RUNAT')
        is obtained.
        The tuple <getid(H1), S1> is inserted into 'HC'
```

To Populate 'CV', 'VC':

```
        cve_id = <The CVE ID from exploit query is taken>
        The service name (serv_name) corresponding to this cve_id is obtained
        The sw_name (SW1) corresponding to the service name serv_name (by referring
        'SERVICE') is obtained
        Insert tuple <SW1, 'D', cve_id> into 'CV'
        Check whether in precondition there is security condition of type privilege exists?
        If
        Yes:  Then the tuple <privtype, 'S', cve_id> inserted into 'CV'
        Check whether in postcondition there is security condition of type privilege exists?
        If
        Yes:  Then the tuple <privtype, cve_id> inserted into 'VC'
```

8.1.3 Sample Output

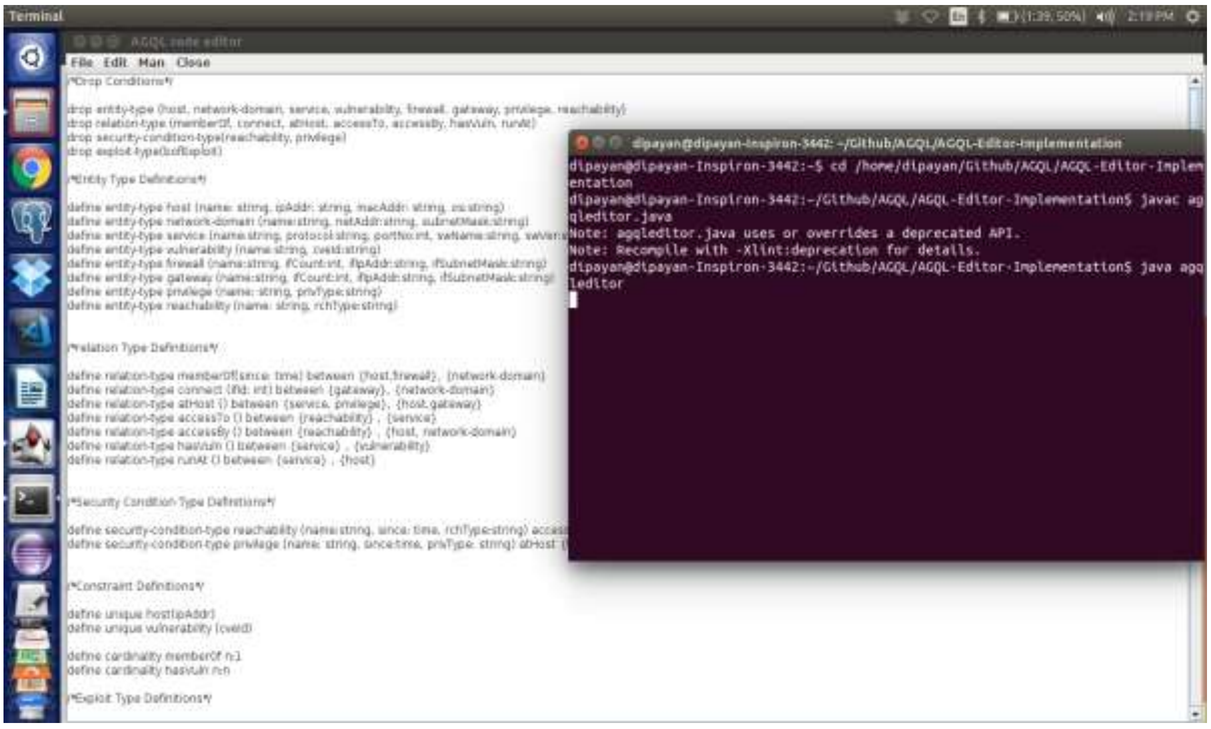
The sample code that is taken as input is given in Appendix B. This attack graph (given below) is generated by the interpreter when the sample code is run.



Chapter 9

AGQL Editor

AGQL Editor is under more improvisation. It is created for sole purpose of writing .agql programs. The front look of the editor can be seen in the below given screenshot (Figure 9-1). This Editor can only open, save files only if they are of the .agql extension. It is very clear that the Editor has four menus in the top bar namely ***File, Edit, Man and Close***. File menu has options to create a NEW File, OPEN File, SAVE File, PRINT File. Edit menu has options to Cut, Copy, Paste. Till now no key shortcut is added in Cut, Copy and Paste. Man menu represents "***Manual***" meaning the description of any command with respect to Linux Terminal. This menu option when pressed writes the syntax format of the query on the current position as a hint. Close menu closes and exits from the editor. The Editor is made simple as of now by using JFrame.



```
/*Drop Conditions*/

drop entity-type (host, network-domain, service, vulnerability, firewall, gateway, privilege, reachability)
drop relation-type (memberOf, connect, atHost, accessTo, accessBy, hasVuln, runAt)
drop security-condition-type(reachability, privilege)
drop exploit-type(bofExploit)

/*Entity Type Definitions*/

define entity-type host (name: string, ipAddr: string, macAddr: string, os:string)
define entity-type network-domain (name:string, netAddr:string, subnetMask:string)
define entity-type service (name:string, protocol:string, portNo:int, swName:string, swVers
define entity-type vulnerability (name:string, cveId:string)
define entity-type firewall (name:string, ifCount:int, ifIpAddr:string, ifSubnetMask:string)
define entity-type gateway (name:string, ifCount:int, ifIpAddr:string, ifSubnetMask:string)
define entity-type privilege (name: string, privType:string)
define entity-type reachability (name: string, rchType:string)


/*relation Type Definitions*/

define relation-type memberOf(since: time) between {host,firewall} , {network-domain}
define relation-type connect (ifId: int) between {gateway} , {network-domain}
define relation-type atHost () between {service, privilege} , {host,gateway}
define relation-type accessTo () between {reachability} , {service}
define relation-type accessBy () between {reachability} , {host, network-domain}
define relation-type hasVuln () between {service} , {vulnerability}
define relation-type runAt () between {service} , {host}


/*Security Condition Type Definitions*/

define security-condition-type reachability (name:string, since: time, rchType:string) acces
define security-condition-type privilege (name: string, since:time, privType: string) atHost (


/*Constraint Definitions*/

define unique host(ipAddr)
define unique vulnerability (cveId)

define cardinality memberOf n:1
define cardinality hasVuln n:n

/*Exploit Type Definitions*/
```

dipayan@dipayan-Inspiron-3442: ~/Github/AGQL/AGQL-Editor-Implementation

```
dipayan@dipayan-Inspiron-3442:~$ cd /home/dipayan/Github/AGQL/AGQL-Editor-Implem
entation
dipayan@dipayan-Inspiron-3442:~/Github/AGQL/AGQL-Editor-Implementation$ javac ag
qleditor.java
Note: agqleditor.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
dipayan@dipayan-Inspiron-3442:~/Github/AGQL/AGQL-Editor-Implementation$ java agq
leditor
```

Figure 9-1 Screenshot of AGQL Editor

Chapter 10

Conclusion & Future Work

In our thesis we have presented a brief description of a query language AGQL and its different constructs. The provision to define constraints to enforce different network semantics into the generated data is one key feature of this language. AGQL supersedes other existing query languages by introducing query constructs to generate attack graph and extract information form it. AGQL facilitates many data definition queries for definitions of different network entities as well as many data manipulation queries for creation of data corresponding to different entity instances. Apart from this two more query categories is kept as a future work i.e. transaction control query and data control query.

With the growth of language we can construct more formal syntax to build more automated queries that will help effectively and contribute a lot more to the System / Network Administrators. Henceforth, up-gradation and maintenance of the interpreter shall lead to its large scale popularity. One benefit of having a semantic definition is that we can use it as input to an interpreter back-end generator. In our case it would be from AGQL to SQL or AGQL to Cypher (or many more as per as requirement). Current interpreter stands as an attractive solution by providing a natural query interface to the network attack data. This might relieve the network security analysis of learning any domain specific language to query the network attack data. This might allow the researchers to model the domain of network security using formalisms, other than graphs.

In future, it will be great to see the entire interpreter supporting thousands of syntax and their uses. The interpreter is just capable of acquiring connections with the Neo4j, the graph database system. And this backend is the other half that we are trying to implement which is kept as a future work.

References

[1] J. Han, Y. Fu, W. Wang, K. Koperski, O. Zaiane, and others, "DMQL: A data mining query language for relational databases," *Proc. 1996 SiGMOD*, vol. 96, pp. 27–34, 1996.

[2] C. Wang, N. Du, and H. Yang, "Generation and analysis of attack graphs," *Procedia Eng.*, vol. 29, pp. 4053–4057, 2012.

[3] S. Jajodia, S. Noel, and B. O'Berry, "Topological Analysis of Network Attack Vulnerability," *Manag. Cyber Threat.*, pp. 247–266, 2005.

[4] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," p. 217, 2004.

[5] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia, "An attack graph-based probabilistic security metric," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 5094 LNCS, pp. 283–296, 2008.

[6] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006.

[7] M. Barrere and E. C. Lupu, "Naggen: A network attack graph generation tool-IEEE CNS 17 poster," *2017 IEEE Conf. Commun. Netw. Secur. CNS 2017*, vol. 2017-January, no. January 2018, pp. 378–379, 2017.

[8] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A Structured English Query Language," *ACM SIGFIDET (now SIGMOD) Work. Data Descr. Access Control*, pp. 249–264, 1974.

[9] L. P. Swiler, C. Phillips, D. Ellis, and S. Chakerian, "Computer-attack graph generation tool," *Proc. - DARPA Inf. Surviv. Conf. Expo. II, DISCEX 2001*, vol. 2, pp. 307–321, 2001.

[10] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "XML-QL: A Query Language for XML," *W3C, \url{http//www.w3.org/TR/1998/NOTE-xml-ql-19980819/}*, vol. 31, pp. 1155–1169, 1998.

[11] K. Gro\ssjohann and N. Fuhr, "XIRQL: A query language for information retrieval in XML documents," *Proc. 24th Annu. Int. ACM SIGIR Conf. Res. Dev. Inf. Retr.*, pp. 172–180, 2001.

[12] S. Noel1, E. Harley, K.H. Tam, M. Limiero and M. Share, "CyGraph: Graph-Based Analytics and Visualization fornCybersecurity", Chapter · January 2016 DOI: 0.1016/bs.host.2016.07.001.

[13] Xinming Ou, Sudhakar Govindavajhala, Andrew W. Appel, "MulVAL: A Logic-based Network Security Analyzer", 14th USENIX Security Symposium, USENIX Association.

[14] Accessed https://www.neo4j.org/

[15] Accessed https://www.lucidchart.com/

[16] Accessed https://www.objectaid.com/

[17] Barik, Mridul Sankar and Chandan Mazumdar, "*A novel approach to collaborative security using attack graph*". In 2011 IEEE 5th International Conference on INTERNET Multimedia Systems Architecture and Application (pp. 1-6). IEEE.

[18] Barik, Mridul Sankar, and Chandan Mazumdar, "*A graph data model for attack graph generation and analysis.*" In International Conference on Security in Computer Networks and Distributed Systems, pp. 239-250. Springer, Berlin, Heidelberg, 2014.

Appendix A

Concrete Syntax

<Query$_{EntDef}$>	::=	define entity-type <EntityType> <Attr$_{DefList}$>
<Query$_{RelDef}$>	::=	define relation-type <RelationType> <Attr$_{DefList}$> between {<EntityType> [,<EntityType>]*} , { <EntityType> [,<EntityType>]*}
<Query$_{SecCondDef}$>	::=	define security-condition-type <SecCondType>(<Attr$_{DefList}$>) <RelationTypeName> {<EntityType> [,<EntityType>]*} [<RelationTypeName> {<EntityType> [,<EntityType>]*}]*
<Query$_{ExploitDef}$>	::=	define exploit-type <ExploitType> (<Attr$_{DefList}$>) CVE (<StringValue>) precond {[<SecCondType> [(<Attr$_{DefList}$>)]?]+} postcond {[<SecCondType> [(<Attr$_{DefList}$>)]?]+}
<Query$_{EntCrt}$>	::=	create entity <EntityType> (<Attr$_{ValList}$>)
<Query$_{RelCrt}$>	::=	create relation <EntityVar> (<AttrPredicate>) <RelationType> (<Attr$_{ValList}$>) <EntityVar> (<AttrPredicate>)
<Query$_{SecCondCrt}$>	::=	create security-condition <SecCondType> (<Attr$_{ValList}$>) <RelationType> <EntityVar> (<AttrPredicate>) (<RelationType> <EntityVar> (<AttrPredicate>))*
<Query$_{ExploitCrt}$>	::=	create exploit <ExploitVar> (<Attr$_{ValList}$>) precond {[<SecCondVar>(AttrPredicate)]+} postcond {[<SecCondVar>(AttrPredicate)]+}
<Query$_{Select}$>	::=	select <EntityVar> where <Expr>
<Query$_{Drop}$>	::=	drop <Type> (<TypeName>[, <TypeName>]*)
<Query$_{Del}$>	::=	delete <EntityVar|RelationVar|SecCondVar|ExploitVar> where <Expr>
<Expr>	::=	<Predicate> | (<Predicate> and <Expr>) | (<Predicate> or <Expr>)
<Predicate>	::=	<RelPredicate> | <AttrPredicate>
<RelPredicate>	::=	(<EntityVar> <RelationVar> <EntityVar>)
<AttrPredicate>	::=	(<Variable>.<Attr> <RelOp> <Value>)
<EntityVar>	::=	<Variable>: <EntityType>
<RelationVar>	::=	<Variable>: <RelationType>
<SecCondVar>	::=	<Variable>: <SecCondType>
<ExploitVar>	::=	<Variable>: <ExploitType>
<Attr$_{DefList}$>	::=	∈|(<Attr$_{Def}$> [,<Attr$_{Def}$>]*)
<Attr$_{ValList}$>	::=	∈|(<Attr$_{Val}$> [,<Attr$_{Val}$>]*)
<Attr$_{List}$>	::=	∈|(<Attr> [,<Attr>]*)
<Attr$_{Def}$>	::=	<Attr> : <ValueDomain>
<Attr$_{Val}$>	::=	<Attr> : <Value>
<Attr>	::=	<Identifier>
<EntityType>	::=	<Identifier>
<RelationType>	::=	<Identifier>
<SecCondType>	::=	<Identifier>
<ExploitType>	::=	<Identifier>
<Variable>	::=	<Identifier>
<TypeName>	::=	<Identifier>
<Type>	::=	entity-type | relation-type | security-condition-type | exploit-type
<RelOp>	::=	==|>|<|<=|>=
<Value>	::=	<StringValue> | <IntegerValue> |<TimeValue>
<ValueDomain>	::=	string | int | time
<StringValue>	::=	" (~["])* "
<IntegerValue>	::=	[0-9]+
<TimeValue>	::=	[0[0-9]|1[0-9]|2[0-3]]:[0-5][0-9]:[0-5][0-9]
<Identifier>	::=	([A-Z, a-z])([A-Z, a-z, 0-9, -, _])*

Appendix B

Sample code (for output of the code, refer section 8.1.3)

```
/*Drop Conditions*/
drop entity-type (host, network-domain, service, vulnerability, firewall, gateway, privilege, reachability)
drop relation-type (memberOf, connect, atHost, accessTo, accessBy, hasVuln, runAt)
drop security-condition-type(reachability, privilege)
drop exploit-type(bofExploit)

/*Entity Type Definitions*/
define entity-type host (name: string, ipAddr: string, macAddr: string, os:string)
define entity-type network-domain (name:string, netAddr:string, subnetMask:string)
define entity-type service (name:string, protocol:string, portNo:int, swName:string, swVer:string)
define entity-type vulnerability (name:string, cveId:string)
define entity-type firewall (name:string, ifCount:int, ifIpAddr:string, ifSubnetMask:string)
define entity-type gateway (name:string, ifCount:int, ifIpAddr:string, ifSubnetMask:string)
define entity-type privilege (name: string, privType:string)
define entity-type reachability (name: string, rchType:string)

/*relation Type Definitions*/
define relation-type memberOf(since: time) between {host,firewall}, {network-domain}
define relation-type connect (ifId: int) between {gateway}, {network-domain}
define relation-type atHost () between {service, privilege}, {host,gateway}
define relation-type accessTo () between {reachability} , {service}
define relation-type accessBy () between {reachability} , {host, network-domain}
define relation-type hasVuln () between {service} , {vulnerability}
define relation-type runAt () between {service} , {host}

/*Security Condition Type Definitions*/
define security-condition-type reachability (name:string, since: time, rchType:string) accessBy {network-domain, host} accessTo {service}
define security-condition-type privilege (name: string, since:time, privType: string) atHost {host,gateway}

/*Constraint Definitions*/
define unique host(ipAddr)
define unique vulnerability (cveId)
define cardinality memberOf n:1
define cardinality hasVuln n:n

/*Exploit Type Definitions*/
define exploit-type bofExploit (name:string, category:string) CVE("CVE-2008-0106") precond {privilege (privType:"user"), reachability (rchType:"xyz")}
   postcond {privilege (privType:"user")}

/*Entities*/
create entity host (name:"h3", ipAddr:"192.168.148.3", macAddr:"xx:yy:zz:xx:xx:xx", os:"Ubuntu")
create entity host (name:"h1", ipAddr:"192.168.148.1", macAddr:"xx:xx:xx:xx:xx:xx", os:"Ubuntu")
create entity host (name:"h2", ipAddr:"192.168.148.2", macAddr:"xx:yy:xx:xx:xx:xx", os:"Ubuntu")
create entity network-domain (name:"nd1", netAddr:"192.168.148.0",subnetMask:"255.255.255.0")
create entity service (name:"httpd1", protocol:"tcp", portNo:80, swName:"Apache-WS", swVer:"1.1.2")
create entity service (name:"httpd2", protocol:"tcp", portNo:80, swName:"Apache-WS", swVer:"1.1.2")
create entity vulnerability (name:"Vuln1", cveId:"CVE-2008-0106")

/*Relations*/
create relation x:host (x.ipAddr == "192.168.148.1") memberOf (since:"10:11:34") y:network-domain (y.netAddr == "192.168.148.0")
create relation x:host (x.ipAddr == "192.168.148.2") memberOf (since:"10:11:37") y:network-domain (y.netAddr == "192.168.148.0")
create relation x:host (x.ipAddr == "192.168.148.3") memberOf (since:"10:11:39") y:network-domain (y.netAddr == "192.168.148.0")
create relation x:service (x.name == "httpd1") runAt() y:host (y.ipAddr == "192.168.148.1")
create relation x:service (x.name == "httpd2") runAt() y:host (y.ipAddr == "192.168.148.2")
create relation x:service (x.name == "httpd1") hasVuln() y:vulnerability (y.cveId == "CVE-2008-0106")
create relation x:service (x.name == "httpd2") hasVuln() y:vulnerability (y.cveId == "CVE-2008-0106")

/*Security Conditions*/
create security-condition reachability (name:"httpd31", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.3")} accessTo {y:service (y.name == "httpd1")}
create security-condition reachability (name:"httpd32", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.3")} accessTo {y:service (y.name == "httpd2")}
create security-condition reachability (name:"httpd12", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.1")} accessTo {y:service (y.name == "httpd2")}
create security-condition reachability (name:"httpd21", rchType:"xyz") accessBy {x:host (x.ipAddr == "192.168.148.2")} accessTo {y:service (y.name == "httpd1")}
create security-condition privilege (name:"user3", privType: "user") atHost {x:host (x.ipAddr == "192.168.148.3")}


/*Exploits*/
```