MASTER

A Domain-Specific Query Language to Investigate Industrial Network Security Data

Pejathaya Murali, S.

Award date:
2020

Department of Mathematics and Computer Science
Security and Privacy Research Group

A Domain-Specific Query Language to Investigate Industrial Network Security Data

Master Thesis

Sashaank Pejathaya Murali

Supervisors:
Dr. Luca Allodi
Dr. Mario Dagrada

Eindhoven, August 2020

Abstract

Over the years, cyber attacks targeting OT-environments such as industrial control systems (ICS) have been surging in number. Intrusion detection systems (IDS) monitor network activities, and report events such as alerts and network logs. An important role of security analysts is to analyze events raised by an IDS to gauge the presence of adversarial activity. Unfortunately, an IDS can generate numerous events per day including false alarms, making the life of security analysts cumbersome if they have to address every one of them. In the process of prioritizing events, some critical events might be overlooked. Furthermore, the data collected by an IDS is raw and unstructured from different data sources and therefore complicated with complex relationships. Therefore, there is a need for defining a simple, yet expressive way to query this complex data, in order to ease the task of security analysts. This thesis aims at assisting security analysts with their task of identifying and investigating OT security incidents by providing an abstraction to the underlying data, with a usable and expressive query language.

Acknowledgements

I would like to thank my supervisor, Prof. Luca Allodi, for being my constant source of inspiration throughout my Masters program, right from his courses, to the way he guided me through my summer internship and my Masters thesis.

I would like to truly thank my graduation tutor, Mario Dagrada for his continuous help, patience and guidance during this project. This project would not have been successful without him.

I would also like to thank all my colleagues at Forescout Technologies B.V., for their constructive reviews and feedback, in helping me achieve better results. Especially, Elisa Costante for giving me this opportunity and trusting me with this project.

My biggest thanks goes to my family who have continuously supported me throughout all these years, even if we were miles apart. I would not have been able to write these lines without them.

Last but not the least, a special thanks to my best friend, well-wisher and pillar of strength, Sangavi, for her unlimited support and encouragement throughout my Masters program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of cyber-attacks targeting industrial networks and, in general, Operation Technology (OT) systems, is on the rise. The most renowned attack on an industrial network is the Stuxnet[1] attack. Stuxnet was a malware that targeted an Iranian nuclear enrichment facility, that aimed to infiltrate and control the facility. It self-replicated by using a zero-day exploit in the Windows Active Directory in the lookout for the Siemens STEP 7 software, which was used to manipulate the Programmable Logic Controllers (PLCs) that managed the centrifuges of the uranium cooling system. The compromised PLCs were then manipulated by the malware to change the rotational speed of the centrifuges, eventually leading to breakage. The network was infected with the malware by a USB drive, which managed to stay unnoticed and the changes of the centrifuges were also not monitored.

The security of an industrial networks is paramount for the following reasons:

- Industrial networks use legacy systems with outdated software that maybe vulnerable.
- Proprietary protocols that are used by the internal systems are insecure, i.e, there is no authentication or encryption of any sort.

The main objective[2] of adversaries who target OT networks is to disrupt the internal working of the industrial systems, ultimately causing damage. OT networks are mostly targeted by attackers that have a military or a political cause. These threat actors maybe either state-sponsored, hacktivists or just competitors. These attacks are often carefully planned and highly elaborate, in order to gain access into the industrial network.

Although it is difficult to defend against these cyber-attacks, it is however greatly possible to detect them by means of intrusion detection systems (IDS). The role of an IDS is to monitor the network and raise alarms if it detects any kind of malicious or suspicious activity. OT-specific Network IDS (OT-NIDS) aim at reducing the threat exposure of OT networks by monitoring network activities, fingerprinting network devices and alerting in case of suspicious activities. Unfortunately, NIDSs generate thousands (if not millions) of event logs per day including false positives, making it hard for the security analyst to single out relevant information in case of an incident. A complementary issue in the standard security workflow involving OT networks is that security analysts usually lack OT domain-specific knowledge and thus struggle with understanding the meaning and implications of alerts coming from the OT network. Oftentimes,

[1]https://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet

[2]https://www.trendmicro.com/vinfo/pl/security/news/cyber-attacks/why-do-attackers-target-industrial-control-systems

a security analyst might also overlook some critical events that get drowned in the ocean of logs generated. The data collected by an IDS is raw and unstructured, and may also come from different data sources and in different data formats. These data sources are often isolated from each other, having complicated relationships among one another. Thus, there is a need for a high-level way to be able to query this complex data that provides abstraction to the underlying complexity and also to be able to be used on multiple data sources directly, thereby providing abstraction also to the native language of the backend database. Furthermore, there is also a need for situational awareness or context around the data, for the security analyst to be able to obtain an overview of the assets and the events generated around them. Therefore, defining a query language that can query the OT security data for intuitive testing and analysis of the security of OT environments is indispensable.

1.1 Problem Statement

Directly exposing a complex query language such as SPARQL [41] or Cypher [32], to a security analyst would not be ideal because of the complexity and the steep learning curve for each language.

Moreover, there are some other problems associated with querying data collected by an IDS to investigate security incidents. The following sections describe some of the commonly encountered problems during incident investigation.

Abstraction

Some events generated by an IDS might be too detailed and elaborate, for a security analyst to be able to detect threats. This would require deep technical knowledge of the network and industrial processes. Possession of this deep domain knowledge cannot be expected out of all analysts. Furthermore, the abundant OT security information is often too detailed and difficult to be interpreted by the analyst, in a short time. Sometimes, the amount of events generated by the IDS is so huge that the analyst might get drowned and exhausted in the process of investigation. Thus, there is a need for security domain-specific knowledge to not be fully exposed to the analyst, in order for him to able to quickly query for what he wants without being lost in the sea of information.

Context

The concepts or entities (such as assets, alerts, network logs, vulnerabilities and so on) from the data collected by an IDS are often isolated, because they are aggregated from different data sources and are available in different data formats. Additionally, the relationships among these entities are difficult to access as the data sources are separated from each other. For instance, let us take the example of a single alert from an IDS. There will be a queue of events and a security analyst cannot afford to waste too much time in deeming whether this alert is a threat to the organization or not. Therefore, a thorough analysis of the context around this alert is important for him to draw conclusions. From the context around the alert, he would be able to find out details such as:

- Who the attacker is?
- Who the target is?
- What type of attack it is?

- How long this event has persisted?
- What the motive of the attacker was?
- If the attack was successful or not?
- What vulnerability was exploited?

Thus, there is a need to provide situational awareness or context to the analyst, so that he is fully aware of what exactly is happening in the network.

Exploration

Because of the huge amount of unstructured data, that is also isolated, it is hard to navigate across the underlying data model fully, even more so, in the presence of time constraints. Furthermore, it is also quite difficult to validate the investigation hypothesis of the analyst, because of the inability to explore across the relationships from one entity to another and the occurence of alternative scenarios of a possible cyber-attack. Therefore, there is a need to not only provide context around the data, but also it is equally important to provide ways to explore between the entities and relationships in the underlying data.

1.1.1 Research Question

Thus, the above problem statement can be formulated into a two-fold research question as follows:

1. Is there a simple, yet expressive, way to query the data collected by an IDS for security incident investigation?
2. Does this solution allow the security analyst to investigate the context around the data, while offering abstraction of the underlying data and its complexity?

1.1.2 Proposed Solution

We want to enable security analysts (end users) to search within the entirety of data collected by an Intrusion Detection System (IDS) from different data sources, with queries ranging from simple free-text to advanced search queries. The outcome of this project would be that it returns and contextualizes the most relevant results, greatly improving a security analyst's investigation workflow within a Security Operations Center (SOC). Besides, we also wish to incorporate built-in functions, keywords or filters that are domain-specific.

Therefore, the main aim of this project is to define how to query the data collected by an IDS using a syntax that should be simple (for usability), yet flexible and expressive enough to support all the required use cases (that shall be mentioned in the Chapter 3). In essence, we wish to define a domain-specific query language called the ”Incident Investigation Query Language (IIQL)” that is simple and easy to use (compared to all other existing query languages) that can be later translated to a more complex query language such as Cypher [32] depending on the OT security data model and backend. Thus, with this project we aim at providing security analysts with a solution that can support their identification and investigation of OT security incidents.

1.2 Thesis Organization

This thesis is structured as follows:

Chapter 2 introduces technology and terminology that will be used repeatedly throughout this thesis, to provide necessary background to the reader.

Chapter 3 elaborates on the literature survey performed for this research, detailing on the existing query languages.

Chapter 4 discusses the proposed solution for solving the research question by leveraging an existing query language without reinventing the wheel.

Chapter 5 presents the case studies performed to evaluate the developed prototype with a real-world scenario.

Chapter 6 discusses the results from the validation of the developed prototype with internal domain experts.

Chapter 7 concludes the research conducted with scope for possible future work.

Chapter 2

Background

This chapter is meant to introduce some background terminology in order to understand this research work such as industrial control systems (ICS) and Operational Technology (OT) incidents. It provides detailed information on the technologies used in this thesis and the research conducted.

2.1 Operational Technology (OT)

Operation Technology[1] refers to the hardware and software that manages the control of physical devices in an industrial enterprise. Typically, these enterprises maybe oil, gas or nuclear plants. Initially, OT security was not important, as OT systems were not connected to the Internet, and thus the attack surface of OT was minimal and unexposed to external threats. However, the increasing need for connectivity has converged traditional IT (Information Technology) and OT systems together, thereby increasing the attack surface of OT networks.

2.1.1 Industrial Control Systems (ICS)

Industrial Control Systems (ICS)[2] collectively refer to different types of control systems that include hardware that operate or automate industrial processes. Some types of ICS systems are:

- Supervisory Control and Data Acquisition (SCADA) systems are devices that provide control on a supervisory level. They are used to monitor devices on field sites.
- Distributed Control Systems (DCS) are devices that provide control on field or local level, unlike SCADA.
- Programmable Logic Controllers (PLC) are devices that automate physical processes by receiving signals from sensors and actuators at the local management level.
- Human Machine Interfaces (HMI) are devices that offer a user interface for human users to operate controllers and for adjusting parameters in the industrial environment.
- Data Historian is a centralized database server for logging all information within an industrial setting.

[1] https://www.i-scoop.eu/industry-4-0/operational-technology-ot/
[2] https://www.trendmicro.com/vinfo/us/security/definition/industrial-control-system

Fig 2.1 below shows an example ICS network architecture of an enterprise, organized by the Purdue Reference Model [3]. In the Purdue Architecture, levels 0 to 3 comprise OT and higher

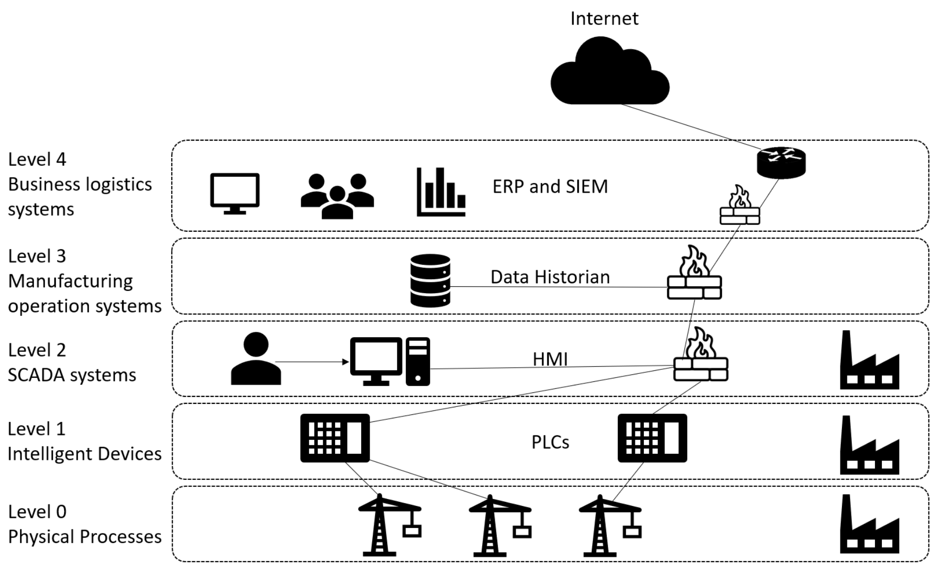


Figure 2.1: ICS Network Architecture of an enterprise

levels comprise IT. As can be seen in the figure:

- Level 0 comprises the devices that perform the actual physical processes
- Level 1 consists of the PLCs that control the devices in Level 0.
- Level 2 comprises the SCADA systems that supervise the devices in Level 1. They consist of DCSs and HMIs.
- Level 3 comprises the manufacturing operations systems that manage the workflow of production. This level contains the Data Historian to log the operational data.
- Level 4 consists of the corporate network that handles business logistics of the enterprise, such as Enterprise Resource Planning (ERP) and Security Incident and Event Management (SIEM). This level contains devices that are connected to the Internet, performing reporting and management.

2.2 Intrusion Detection Systems

An Intrusion Detection System (IDS) is a piece of hardware or software that monitors systems to detect suspicious or malicious activity. It can be Host-based or Network-based. Host-based IDSs (HIDS) monitor individual devices, such as, an antivirus software and Network-based IDSs (NIDS) monitor network traffic. The role of a security analyst is to analyse the events/logs generated by an IDS and make a decision or perform further investigation accordingly.

[3] https://en.wikipedia.org/wiki/Purdue_Enterprise_Reference_Architecture

2.2.1 Intrusion Data Sources

An IDS may collect unstructured data from diferent data sources [24]. A HIDS may inspect data from sources such as audit logs, database logs or firewall logs. An NIDS can monitor data from packet capture logs or other network based data sources. Sometimes, these different data sources maybe isolated or scattered from each other and maybe available in different databases (like SQL or NoSQL) and different data formats (like JSON, XML). Some of the different databases[4] that maybe involved are:

Relational Databases

Relational Databases are databases in which data is fit under pre-defined categories, having a set of tables, consisting of rows (data instances) and columns (categories). Some examples of relational databases are Microsoft SQL Server, MySQL, Oracle Database and so on.

NoSQL Databases

NoSQL (Not-Only SQL) databases are databases that store large amounts of data in different formats such as JSON and not just in the form of tables. Some examples of NoSQL databases are MongoDB, HBase and Couchbase.

Graph Databases

Graph databases are also a kind of NoSQL databases, in the sense that data is not stored as tables, but in the form of nodes (entities) and edges (relationships) as a graph. Some examples of graph databases are Neo4j, AllegroGraph, Virtuosa, OrientDM and so on.

2.3 Summary

As we can see, this chapter introduces the OT environment, ICS network architecture, the concept of IDS and how they collect data from different data sources, with introduction to different types of databases. The amalgamation of IT with OT has exposed ICS networks to a larger attack surface. Thus, it is vital to have an IDS (or more) deployed in an ICS network to monitor network traffic and to be able to quickly respond to incidents.

[4] `https://www.tutorialspoint.com/Types-of-databases`

Chapter 3

State of the Art

There are different query languages for querying different kinds of databases, suited to different needs and users. This chapter mentions some of the existing query languages with a focus on the technologies used by security analysts to query security data models and gives a comparative overview of the prevalent query languages based on certain parameters explained below. The spectrum of query languages that are mentioned in this section are classified into four major categories such as query languages for relational databases, NoSQL databases, search engines and company specific query languages. These categories were formulated based on previous work [13].

3.1 Related Work

A good amount of literature was studied for surveying existing query languages. A detailed overview of this exhaustive list of technologies is presented in Appendix A. This section provides a summarized overview of the existing technologies for querying databases in literature, along with their features and limitations with respect to the problem that is being addressed in this research. This information is tabulated in Table 3.1 below.

Table 3.1: Overview of existing query languages in literature

Categories of Query Languages	**Query Language (s)**	**Advantages**	**Limitations (with respect to the problem to be addressed)**
Query Language for Relational Database	SQL	Basic operations within relations (basic filtering) and basic joins are possible	Expensive joins across tables to obtain context information
Query Languages for Graph Databases	Neo4j's Cypher, Oracle's PGQL, MITRE's CyQL, Diffbot QL, Facebook's GraphQL, Apache's Gremlin	Context information can be obtained (data organized as graphs with nodes and relationships)	Steep learning curve, complex language and syntax

Query Languages for Semantic Web	SPARQL, RQL	Context information can be obtained (data organized as entity-relationship ontologies)	Steep learning curve, complex language and syntax
Query Languages for Search Engines	Splunk's SPL, ElasticSearch's Kibana and DSL	Easy to use syntax with basic filtering, keyword searching and piping	Context information obtained is limited because there is no possibility of exploration between entities, Steep learning curve
Company Specific Query Languages	Nozomi's N2QL	Supports basic filtering, advanced searching and context information can be obtained	This language is proprietary and closed-source
	Microsoft's AHQL	Supports basic filtering, advanced searching and context information can be obtained	This language is proprietary and closed-source
	Endgame's EQL	Easy to use syntax and supports basic filtering, advanced searching, multiple database backends and it is not tightly coupled to the underlying schema	No provision to obtain context information and exploration between entities

3.2 Initial Requirements for the Prototype

The main goal of this research is to define a query language that solves the problem of context, abstraction and exploration to aid a security analyst to investigate incidents more efficiently. Forescout Technologies[1] came up with an initial set of requirements to be achieved with the prototype. The following Table 3.2 tabularizes the basic requirements of the query language that is to be defined, along with some example use-cases. Additionally, these requirements are later used to compare the existing query languages in literature, against each other.

[1]https://www.forescout.com/

Table 3.2: Requirements for Prototype

Requirement	**Example**
Standard full-text search (Searching for a specific keyword)	Search for the keyword “Axis” in the data to find all Axis cameras
Basic filtering (==, !=, AND, OR) within a single concept (e.g. device or alert)	Find a PLC with IP address 192.0.1.2 Find all devices running Windows Find all cameras with vendor ’XXX’
Advanced search among different concepts (e.g. devices, alerts, vulnerabilities, network logs) (with context)	Find EWS initiating port scans Find malfunctioning devices Find all devices with ’CVE-XXX’
Querying abstracted OT operations (with just using a keyword, without knowing the underlying event types)	Find all write operations done by PLCs
Looking for user/hostname activity (for entire context around data for forensic investigation)	Look for all concepts related to user “admin”
Looking for exploitable paths between two assets or from one asset to other assets in the network	Detect exploitable attack path from a given PLC

3.3 Overview of Existing Query Languages

The existing query languages in literature discussed in Section 3.1 above are evaluated against the initial requirements listed in Section 3.2 above. This section gives an overview of the evaluation, weighing their pros and cons.

3.3.1 Ranking Methodology

Since there is a myriad of existing query languages, there is a need to narrow down our research to a few specific ones, that can be used to support our requirements for IIQL. For this reason, we have an initial ranking procedure in which 14 languages discussed above are first ranked according to how many of the requirements they each satisfy, followed by a selection procedure where we zero down on them further to pick one from the final lot to build IIQL from. In table 3.5, they are scored based on a 'YES' (a score of 1) or 'NO' (a score of 0) and their average score is computed to find out how many languages can be considered for the next selection procedure. The average scores for the languages can be seen in 3.3. As can be seen from the table, in order

Table 3.3: Ranking of Query Languages for Selection Procedure

Ranking	Query Languages	Average Score
1	Cypher, CyQL	0.83
2	SQL, ELK, SPL, EQL, N2QL, AHQL	0.3
3	PGQL, Gremlin, SPARQL, RQL, GraphQL, DQL	0.16

to reduce the number of query languages to consider for the next selection procedure, the lowest ranked query languages (ranked 3rd) with a score of 0.16, can be discarded. Thus we are left with only 8 languages to consider for the selection procedure.

3.3.2 Selection Procedure

Table 3.6 tabulates the 8 query languages from the previous ranking step, weighing their pros and cons in the form of certain metrics (with scores allotted to each metric). The authors of [21, 18, 7, 20] use metrics such as Expressiveness, Query Complexity, Learning Curve and so on, to compare query languages with each other. Some of these parameters were refined to compare the 8 query languages in this step, as explained in detail below:

1. **Query Complexity:** This parameter refers to how long or how difficult the query language syntax is to understand. This metric is measured by how hard it is to craft queries efficiently. It is denoted as 'LOW', 'MEDIUM' or 'HIGH' with scores of 3, 2 and 1 respectively (because the less complex the language is, the easier it is to grasp) .

2. **Documentation Support:** This parameter refers to how much documentation support is available online to learn the language and use it. It is represented as 'LOW', 'MEDIUM' or 'HIGH' with scores of 1, 2 and 3 respectively.

3. **Industry Use:** This parameter refers to the degree of industry usage of the particular query language. It is denoted as 'LOW', 'MEDIUM' or 'HIGH' with scores of 1, 2 and 3 respectively.

4. **Open Source:** This parameter refers to whether the query language implementations (for lexer, parser and so on) are open source (the code is freely available) or not. It is denoted as 'YES' or 'NO' with scores of 1 and 0 respectively.

5. **Expressiveness/Selectivity:** This metric refers to how powerful and precise the queries can be formulated. This is an intuitive measurement, that is closely related to the functionality, depending on the variety of ideas that can be represented and communicated in that language. It is denoted as 'LOW', 'MEDIUM' or 'HIGH' with scores of 1, 2 and 3 respectively.

6. **Support for complex operations:** This metric refers to whether the query language provides support for operations such as joins, aggregation, negation, recursion and nesting queries. It is denoted as 'YES' or 'NO' with scores of 1 and 0 respectively.

7. **Support for Database Backends:** This metric refers to whether the query language has support for more than one database backend. This essentially means whether the language offers an abstraction layer that can be used to translate it to another native query language depending on the database backend. It is denoted as 'YES' or 'NO' with scores of 1 and 0 respectively.

8. **User Knowledge:** This metric refers to the amount of user knowledge or training that is required to use the query language. This parameter is closely related to query complexity. Sometimes, a user needs to have domain knowledge as well to query effectively, in case of languages such as EQL, N2QL and AHQL. Hence, this metric is measured by adding both the complexity of the query language and the domain knowledge (if required). It is denoted as 'LOW', 'MEDIUM' or 'HIGH' with scores of 3, 2 and 1 respectively (because the less training, the faster the language can be adopted).

9. **General Purpose/Domain Specific:** This parameter refers to whether the query language is a general purpose one with a score of 1 (if it is applicable to any domain, such as SQL) or if it is specific to a certain domain with a score of 0 (such as CyQL).

10. **Database Dependence:** This metric refers to how much the query language relies on the underlying database. This quantity is measured as follows: 'LOW' implies that the query language can be used with any database and 'HIGH' implies the query language must have a specific database backend to work precisely. It is denoted as 'LOW', 'MEDIUM' or 'HIGH' with scores of 3, 2 and 1 respectively (because the lesser the DB dependence is, the better the language can be used across other DBs).

Thus, in this selection procedure, each of the 8 languages were given a score according to its metric (as explained above) and the average of the total score for 10 metrics for each language was calculated and tabulated in table 3.4. As can be seen from the table, in order to further filter out the query languages to pick for the final lot, the last 3 classes of languages that are ranked 4th, 5th and 6th with scores of 1.4, 1 and 0.9 can be discarded. Thus we are left with only 4 languages to consider for the selection procedure.

Table 3.4: Ranking of Query Languages for Final Lot

Ranking	**Query Languages**	**Average Scores**
1	SQL	2
2	ELK	1.7
3	EQL, SPL	1.6
4	Cypher	1.4
5	CyQL	1
6	N2QL, AHQL	0.9

Table 3.5: Requirements supported by existing query languages

		Query Languages for Graph Databases						**Query Languages for the Semantic Web**		**Query Languages for Search Engines**		**Company Specific Query Languages**		
Requirements	**SQL**	**Cypher**	**PGQL**	**CyQL**	**DQL**	**Graph QL**	**Gremlin**	**SPAR QL**	**RQL**	**SPL**	**ELK (Kibana and DSL)**	**Endgame's EQL**	**Nozomi's N2QL**	**Microsoft's AHQL**
Full Text Search	YES	YES	NO	YES	NO	NO	NO	NO	NO	YES	YES	NO	NO	NO
Advanced Search among concepts (with context)	NO	YES	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
Basic Filtering (AND, OR, ==, !=) for main concepts	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES
Related Concepts Exploration	NO	YES	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
Attack Reachability/ Attack Paths	NO	YES	NO	YES	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
Querying abstracted OT operations	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	YES	YES	YES

Table 3.6: Selection Procedure with Metrics

Metric	**SQL**	**Cypher**	**CyQL**	**SPL**	**ELK**	**EQL**	**N2QL**	**AHQL**
Query complexity	LOW	HIGH	MEDIUM	MEDIUM	MEDIUM	MEDIUM	MEDIUM	MEDIUM
Documentation Support	HIGH	HIGH	LOW	HIGH	HIGH	HIGH	LOW	LOW
Industry Use	HIGH	HIGH	LOW	HIGH	HIGH	LOW	LOW	LOW
Open Source	YES	YES	NO	NO	YES	YES	NO	NO
Expressiveness	HIGH	MEDIUM	MEDIUM	MEDIUM	MEDIUM	HIGH	MEDIUM	MEDIUM
Support for Complex Operations	YES	YES	YES	YES	YES	YES	YES	YES
Support for Database Backends	YES	NO	YES	NO	NO	YES	NO	NO
User Knowledge	LOW	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH
General Purpose/ Domain Specific	General Purpose	General Purpose	Domain Specific	General Purpose	General Purpose	Domain Specific	Domain Specific	Domain Specific
Database Dependence	HIGH	HIGH	HIGH	LOW	LOW	LOW	HIGH	HIGH

3.4 Conclusion

As can be seen from table 3.4, Cypher, ELK's QLs, Splunk's SPL and SQL support all or most of the requirements and are ranked high in the selection procedure as well. However, all of them (except SQL) have a steep learning curve and they all have an extensive code base to build our own domain specific query language from. Additionally, Splunk is proprietary as well, and not open source. SQL on the other hand, can not be used for non-relational databases. Thus, Endgame's EQL (ranked 3rd with a score of 1.6) is the best solution, as it has a relatively smaller code base with simple Python helpers to build our own language from. Furthermore, it is not tightly coupled to a particular kind of database and it can be translated to multiple database backend languages.

Chapter 4

The Proposed Solution

This chapter introduces some detailed use-cases that were aimed to be achieved with the project. Furthermore, it chooses one of the existing solutions discussed in the literature overview, justifies the choice and explains how it will be extended to achieve the required use cases. Finally, it concludes by covering the steps taken to implement IIQL (extending EQL) and the transpilation of IIQL to Cypher (the native database backend query language of Neo4j) and SQL. In order to build our own domain-specific data model driven language, it is natural that we can base our language from an existing solution for reasons to be easily integratable, rather than to build a new language from scratch, thereby reinventing the wheel.

4.1 Use Case Scenarios in the OT/IT environment

This section discusses a common use-case scenario in the daily-life of a security analyst who investigates incidents in any IT/OT network. Moreover, it also introduces some other use cases relevant to incident investigation in IT and OT networks which will serve as a basis for validating the proposed query language prototype in Chapter 6.

4.1.1 Typical Use Case Scenario in an OT/IT environment

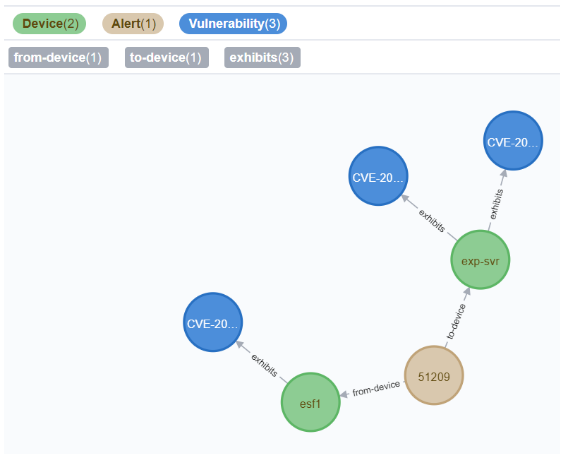
The following subsection mentions an example classic use case scenario in an OT/IT incident investigation environment and how the developed prototype will aim to achieve it.

A typical use-case scenario for a security analyst would be to either start with an asset/device or an alert, in case he wants to know the vulnerable devices in the network, or how an alert is related to another. The functionality of finding attack paths/attack reachability is one of the values that is being added via IIQL. This essentially means finding the sequence of vulnerabilities that an adversary could exploit for lateral movement through the network. For example, a literal translation of a query that can be answered could be "Show me how an attacker can get from this device to this?". This answers the question of how adversaries can potentially leverage multiple vulnerabilities to incrementally penetrate a network.

Example Use Case

An analyst might want to investigate the source of an alert and check if the alert exploits any known vulnerabilities of the target device in the network. The result that would be obtained is shown in Fig. 4.1. Therefore, the analyst can now see an attack path with which an attacker

Figure 4.1: Attack Reachability between two devices



might have penetrated the network. So a prediction that the analyst can make is that the attacker could have exploited the vulnerability in the ‘esf1’ device (which is the source device of the alert) and attacked the ‘exp-svr’ device in the same subnet (which is the destination device of alert) exploiting one of the two vulnerabilities it has. Thus, in this way, attack paths can be mined from the data. An analyst could also obtain entire context information such as the roles of the devices, how alerts are related, and what vulnerabilities are present in the device.

4.1.2 Extended Use-Cases

In addition to the above, the following set of extended use cases were envisioned for the proposed query language prototype.

A. Network Analysis

The primary use case here is to enable proactive investigation by querying all data collected by an IDS. The use cases of this category fall in two sub-categories namely: Security and OT related use cases.

Security-related use cases

The security-related use cases are generic, and can be applicable to both IT and OT scenarios. Some of them are to be able to:

- Search for all devices with a certain CVE ’CVE-XXX’ within a network.
- Search for all the Programmable Logic Controllers (PLCs) that are vulnerable in the network (and also obtain related information such as alerts, change logs and so on).
- Search for all the Engineering Workstations initiating a port scan (and also obtain related information such as their alerts and vulnerabilities).
- Find all outdated Windows (older than Windows 10) devices in a network.

OT-related use cases

The OT-related use cases are applicable specifically to the OT environment. Some of them are to be able to:

- Find devices performing reconfiguration operations (such as firmware download or upload).
- Find devices performing write operations (such as write registry).
- Find malfunctioning devices connected to a particular workstation.

B. Investigation

The primary use case here is to enable reactive investigation activities from pre-identified issues (such as via the correlation algorithms). The use cases of this category fall under two sub-categories namely:

Investigation-related use cases

The investigation-related use cases are generic and applicable in both IT and OT environments. Some of them are to be able to:

- Find exploitable or attack paths starting from a particular device. This functionality can be used to gauge the attack reachability from an asset to another asset in the network.
- Find all information related to a particular device. This essentially is one of the main research goals of this project, showing the entire context around a device.

Forensic-related use cases

Forensic-related use cases are again very generic and broad, and can be applied to both OT and IT security environments. These use-cases are context-related as well. Some of them are to be able to:

- Retrieve activities related to a specific user (such as look for "admin" user and retrieve all related information such as alerts and devices).
- Retrieve all activities related to a host name.

C. Search Solutions

The search solutions that we aim to achieve from this project.

1. Full-text search, such as look for "*device AND CVE-XXX*". Although this is less precise and it is difficult to express complex queries thereby adding performance issues. The primary use case here is to enable free text search in the available data. Some of the possible use cases may be as follows:
    - Search for "download" and find all concepts containing the word "download" with their relationships (devices, related CVEs, aggregated alerts and so on).
    - Search for "maintenance" and find all concepts related to a device under possible maintenance.

- Search for “rpc” to find all concepts related to RPC calls (devices, related CVEs, aggregated alerts and so on).

2. Advanced structured queries, such as ” *device(role=plc, is_vulnerable=True) JOIN device(ip=10.1.2.3)*”. This solution is more precise and it is easier to express more complex queries, but this would require the knowledge of the query language.

4.2 Proposed Existing Solution - A Starting Point

Thus, as can be seen from the literature overview in the previous chapter, EQL by Endgame is the most promising existing solution that we can build upon because:

- The parser and the abstract syntax tree (AST) is open source and a lot of documentation is available.
- In some query languages, a good understanding of the data schema is necessary for formulating precise queries. However, EQL is schema-optional, which means one can use it with whatever schema one has at hand. There is no need to have a predefined schema to use EQL. Also there are helper tools for translating schemas (such as JSON to XAML) as well.
- It supports multiple database backends and conversion between data formats. EQL is used mainly for security events stored in the JSON format, but it can also be used for other data formats (as long as we provide the data as a list of Python dictionaries and have the key-value pairs be compatible with the JSON data format). Additionally, EQL can be used as a higher level language and can be translated to another language.
- It features ‘eqllib’ that is a library of event-based analytics to detect adversary behaviours that is mapped to MITRE’s ATT&CK framework. Also, keyword mappings are possible to any security event dataset.

4.2.1 Limitations of the existing solution

As mentioned in Table 3.5, the existing solution (EQL) supports all of the use cases except:

1. Standard full-text search
2. Advanced Search with context
3. Related concepts exploration
4. Attack paths discovery

Thus, in order to achieve the use cases we need, we will have to extend the current EQL implementation to:

1. Translate it into the native DB language.
2. Modify it to introduce a full text search module.
3. Modify it to facilitate related concepts exploration and finding attack reachability between assets.

4.2.2 Bridging the Gap by IIQL

Thus, as can be seen from the above use-cases (in Section 4.1, in an IT/OT security environment, it is paramount to have context information around a concept, such as a device or an alert. This is not possible in any of the query languages covered in the literature overview, except Cypher for Neo4j. IIQL would essentially do a translation between a high level easy syntax to the native backend Cypher. The main purpose of IIQL is that the security analyst is not exposed to the syntax of the native backend query language, i.e, Cypher. It encapsulates the knowledge of the different relationships present in the underlying data model. IIQL is domain specific and allows flexible ad-hoc queries to query the unified data model that correlates alerts to devices and vulnerabilites. It poses as a middle-tier component that translates EQL (high level language) queries to the lower level Cypher.

Solving the problem with IIQL

1. IIQL will be a declarative/ non- procedural query language, where one specifies what needs to be done (i.e. a simple one line query) rather than exactly how to do it (a more elaborate Cypher query)
2. IIQL will increase the clarity of analytic queries against our OT security data model, especially as the model becomes more complex. It does this by encoding cyber semantics into the query language itself, encapsulating and hiding many of the constraints that must be expressed in the native graph database queries. This greatly helps reduce the learning curve and increase the productivity of security analysts.
3. IIQL will provide full scope of adversary activities (Exploitation Paths)
4. IIQL will provide relevance to known vulnerability paths (Attack Reachability)
5. IIQL will provide situational awareness (Subnet information and what devices has what vulnerabilities)

4.3 Proposed Extension to EQL

This section proposes and describes the new operators that were added to EQL, in order to achieve the use cases in an OT security environment and provide ease-of-use to a security analyst.

4.3.1 Plan to Extend EQL to IIQL

IIQL (extended EQL) can be used in two modes, namely, the Basic Mode (for basic operations as mentioned in table 4.3) and the Investigation/Advanced Mode (for context-aware capabilities). The table 4.1 enlists the proposed operators along with the use-cases that they aim to achieve and table 4.2 mentions the proposed function, with their respective functionality and syntax.

4.3.2 Solving the problem with the operators proposed

Thus, the operators proposed above collectively solve the main problems such as context and exploration, including the specific use-cases (mentioned in Table 4.1) that we aim to solve. The problems, however, cannot be solved by other languages considered in the literature overview chapter including EQL by itself (due to its limitations mentioned in section 4.2.1), except the query languages for graph databases. Nevertheless, these will have a steep learning curve because of their syntax.

Table 4.1: Proposed New Operators to EQL

Operator	**Functionality**	**Syntax**	**Use-case (s) Achieved**
investigate	Can be used for context information involving specific concepts	| **investigate** concept_names separated by commas	Advanced search among different concepts and entire context around data
find/explore	Can be used to perform a free text search and fetching all context around it	**find** <concept_name>"text" **explore** "text"	Standard full-text search and entire context around data
path	Can be used to get the underlying subgraph connecting one concept to another concept	**path** [start_concept where expr] [end_concept where expr]	Exploitable paths between assets and alert chains between assets
action	Can be used to find out if a device is performing a specific action specified in the keyword	| **action** <keyword>	Querying for abstracted operations
filter	Can be used to filter out results based on concept name and respective attribute	| **filter** <concept_name>. <attribute><comparison_operator> <value>	Advanced search /filtering between concepts
influencer	Can be used to find out the most influential device in the network (which is the one with the most number of relationships or events)	| **influencer** | head n	Abstraction of entire context around assets

Table 4.2: Proposed New Function to EQL

Function	**Purpose**	**Syntax**
duration()	Can be used to find out the links or devices that are available or persistent for the longest time in the network (after computing the difference between the first seen and the last seen timestamps of the link or device).	**duration**(last_seen, first_seen)

4.3.3 Example Queries with Use Cases and Added Value

This subsection explains some of the use cases that can be covered with the newly proposed operators with example queries that can be formulated along with the added value that each operator would bring to an OT security analyst.

1. **Operator:** investigate

Use case #1: Find for all devices with a certain CVE
Example Query:

```
vulnerability where name contains "CVE-2014" | investigate device
```

Use case #2: Find devices performing write operations (e.g., write registry)
Example Query:

```
operation where name contains "write" and file_path contains "\\MACHINE\\" |
    investigate device
```

Expected Usage: This operator is useful to an analyst if he wants to narrow down his search for context information for one or more concepts only (for e.g. if he wants to just look for context with alerts and vulnerabilities or only alerts)

2. **Operator:** find/explore
Use Case #1: Find all information related to a particular device (not necessarily only links), i.e. show the context around a device with respect to a single concept i.e. 'device'
Example Query:

```
find device "exp-svr"
```

Use Case #2: Retrieve activities related to a specific user (e.g., look for "admin" user and retrieve all related information such as alert and devices)
Example Query:

```
explore "admin user"
```

Expected Usage: This is useful to an analyst if he wants to perform a full-text search with entire context around nodes having the text (for e.g. if he wants to search for all details about a particular device i.e. its vulnerablilities, alerts and roles)

3. **Operator:** path
Use Case: Find subgraph from one device to another
Example Query:

```
path by link [device where name = "exp-svr"] [device where name = "esf1"]
```

Expected Usage: This operator is useful to an analyst if he wants to assess the reachability from one device to another specific device or to all devices in the network, for e.g. from one vulnerable device to another vulnerable or normal device and vice-versa.

4. **Operator:** influencer
Use Case: Find the top 5 devices in the network that have the most number of relationships/events
Example Query:

```
device where true | influencer | head 5
```

Expected Usage: This operator is useful to an analyst if he wants to narrow down his investigation of devices, which are not very useful and focus on the ones that dominate the network.

5. **Operator:** action
   Use Case: Find if a device is performing a reconfiguration operation
   Example Query:

   ```
   device where true | action reconf
   ```

   Expected Usage: This operator is OT-specific, it is useful to an analyst if he wants to search for devices that perform specific actions, such as dangerous operations, thereby also providing abstraction to the underlying event types in the IDS.

6. **Operator:** filter
   Use Case: Filter alerts by high-severity
   Example Query:

   ```
   device where role == 'plc' | action dangerous | filter alert.severity > 2
   ```

   Expected Usage: This operator is useful to an analyst if he wants to filter out the large number of results obtained from the previous operator chained to it, by a specific concept and attribute.

7. **Function:** duration
   Use Case #1: Find the top 5 persistent links in the network
   Example Query:

   ```
   link where duration(last_seen, first_seen) | head 5
   ```

   Use Case #2: Find the least 5 active devices in the network
   Example Query:

   ```
   device where duration(last_seen, first_seen) | tail 5
   ```

   Expected Usage: This function is useful for an analyst if he wants to know the longest active links or devices in the network. This is useful in case he wants to know the devices that were involved in the most persistent links to look into the bytes exchanged, for how long the devices were connected, what roles they were playing and their network logs or vulnerabilities. With this function, he can also find out the most inactive devices on the network.

4.4 Basic Mode of IIQL

The following table 4.3 lists the basic operators that are already present in EQL for us to incorporate in IIQL. The operators that can be used in this Basic Mode can also be used to query the PostgreSQL database with the OT security data, in addition to querying the Neo4j database.
From table 4.1, the 'find' operator (that is newly proposed) can be added and used in this basic mode for free-text search of context information within a single concept as opposed to the 'explore' operator that can be used in the investigation mode (for entire context information across concepts)

Pipes

EQL queries can include pipes (|) for aggregations, statistics and filtering of results. Table 4.4 below lists the pipe commands already present in EQL.

Table 4.3: Basic Operators present in EQL

Operators	**Example Queries**
Boolean operators (and, or, not)	Host where Name contains 'svr' and/or criticality >2 Role where not sdID == 'plc'
Value comparisons (<,<=, ==, != , >=, >)	Host where criticality >= 2 Role where sdID = 'plc'
Wildcard Matching	IPAddress where sdID contains '10.1.'
Joins	join by Vulnerability [Host where Name == 'exp-svr'] [Role where sdID == 'windows_ws']

Table 4.4: Pipe Commands present in EQL

Pipe Operators	**Purpose**	**Example Query**
count	Returns the number of occurences	Host where Name contains 'svr' | **count**
sort	Sorts results according to the criteria	Link where true | **sort** tx_bytes
head	Outputs the first N results	Host where criticality >0 | **head** 5
tail	Outputs the last N results	Host where criticality >0 | **tail** 5

4.5 Investigation Mode of IIQL

All other proposed operators in Table 4.1 except 'find', such as 'investigate', 'explore', 'path' , 'action' and 'influencer' can be used in the investigation mode of IIQL, because these operators give more insight to the security analyst by providing more context-oriented information. The transpilation of IIQL operators to the respective native languages of the backend databases, in the two available modes is shown in Table 4.5. As we can see from the table, some of the operators in the Investigation Mode cannot be transpiled to SQL, as they are context-oriented operators, and take into account the nodes and relationships of the underlying data.

Table 4.5: Transpilation of IIQL operators

IIQL Modes	Operators	Neo4j	SQL
Basic Mode	- Sort - Head - Tail - Logical AND, OR, NOT	YES	YES
Advanced Mode	- Explore - Path - Action - Investigate - Influencer	YES	PARTIALLY (only Investigate and Action)

4.6 General Architecture

The general architecture proposed for the project is shown in Fig. 4.2 below. (Note: The grey color-coded parts is out of scope of this research/work).

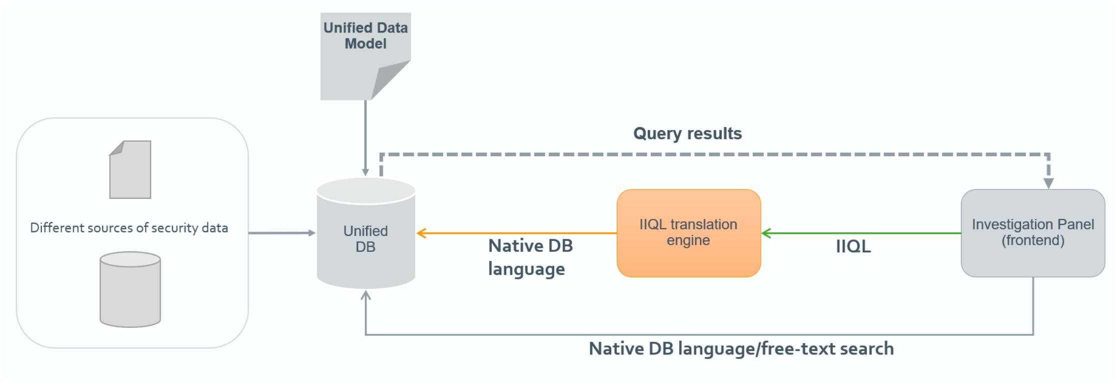

Unified Data Model
Different sources of security data
Unified DB
Query results
Native DB language
IIQL translation engine
IIQL
Investigation Panel (frontend)
Native DB language/free-text search

Figure 4.2: Proposed General Architecture

The different components of the architecture are described below:

1. **Unified Data Model:** To ingest and unify all the OT network security data collected from different sources, in a single model.
2. **Unified Database:** The database backend that IIQL will query. It consists of data fitted against the unified data model.
3. **IIQL Translation engine:** To translate the domain-specific IIQL into the native database

language such as Neo4j's Cypher query language [31] as there is an existing Neo4j database from which we started work on.

4. **Frontend:** To visualize the results of the queries as graphs or tables.

4.6.1 Knowledge Graph Model for IIQL

IIQL can be used on any data model or database; however Fig. 4.3 depicts the underlying data model that IIQL was used on, expressed as a graph of nodes (entities) and edges (relationships). Table 4.6 below shows the list of concepts and attributes that are available for the user to query for.

Figure 4.3: Knowledge Graph Model for IIQL

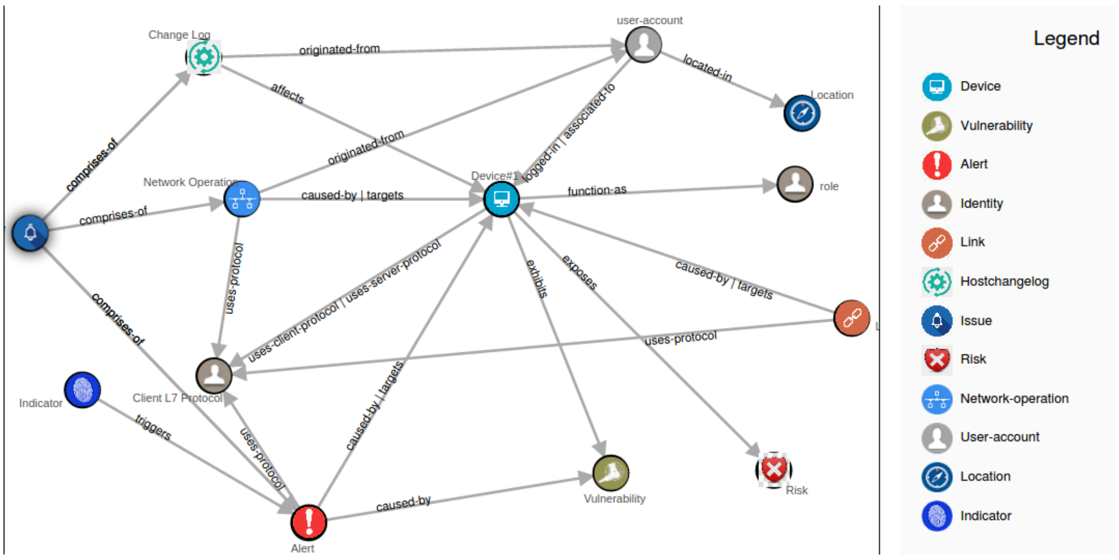


Table 4.6: List of concepts and attributes of data model

Name	Description	Attributes to query
device	Network hosts	name main_role os_version ip purdue_level vendor
alert	Alerts	event_type_id event_name src_port dst_port severity timestamp
hostchangelog	Logs indicating some changes in a host	event_name event_type_id new_value old_value timestamp
link	Network links	first_seen last_seen rx_bytes tx_bytes ports
vulnerability	Vulnerabilities associated to devices	name cvss matching_confidence vendor
l7protocol	Application protocols	name
user	Users	user_id
indicator	Indicators of compromise	name indicator_type
network_operation	Network operations (DNS resolution, authentication,...)	event_type_id severity timestamp

4.7 Grammar and Parse Tree of IIQL

An external dependency for IIQL (extended EQL) is the Python library Lark. Lark generates a parser generator for the below grammar, which IIQL uses to parse queries. The grammar and the parser are by EQL, which IIQL uses, to check the syntax of the input query. If there is a syntactic error, it is thrown to the user.

4.7.1 Grammar

EQL's already existing grammar was modified with the new operators that were proposed to be added to it. A snippet of the EQL grammar modified is shown below.

```
definitions: definition*
?definition: macro | constant

macro:    "macro" name "(" [name ("," name)*] ")" expr
constant: "const" name EQUALS literal | "const" name CONTAINS literal

query_with_definitions: definitions piped_query
piped_query: base_query [pipes]
           | pipes
base_query: sequence
          | join
          | find
          | explore
          | path
          | event_query
event_query: [name "where"] expr
find: "find" name literal
explore: "explore" literal
path: "path" join_values? subquery_by subquery_by+
pipes: pipe+
pipe: "|" name [single_atom single_atom+ | expressions]

join_values.2: "by" expressions
subquery_by: subquery named_params? join_values?
subquery: "[" event_query "]"

// Expressions
expressions: expr ("," expr)* [","]

// Need to recover these tokens
EQUALS: "==" | "="
CONTAINS: "contains"
COMP_OP: "<=" | "<" | "!=" | ">=" | ">"
?comp_op: EQUALS | COMP_OP | CONTAINS
NOT_OP:     "not"
?single_atom: literal
            | field
            | base_field
base_field: name
field: FIELD
literal: number
       | string
```

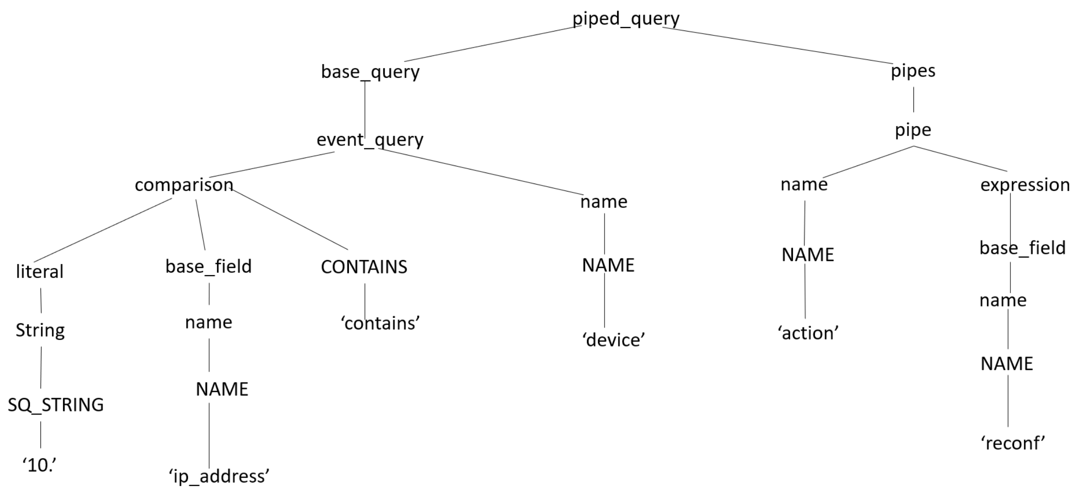
4.7.2 Abstract Syntax Tree - Example

EQL's parser recognizes the tokens of the input IIQL query in terms of EQL's grammar structure and maps them to an abstract syntax tree. The abstract syntax tree generated for the example IIQL query:

```
device where ip_address contains '10.' | action reconfig
```

is shown in Fig. 4.4 As can be seen from the figure, every IIQL query is a *piped_query* which may contain a *base_query* and *pipes*. The *base_query* contains an *event_query* that consists of the main concept name and attribute, along with the condition. The *event_query* is the main aspect of this language, because every IIQL query must contain an *event_query* at the basic level (for primary filtering). The pipes contain the particular pipe that consists of a name (of the pipe) and expression (value passed to the pipe).

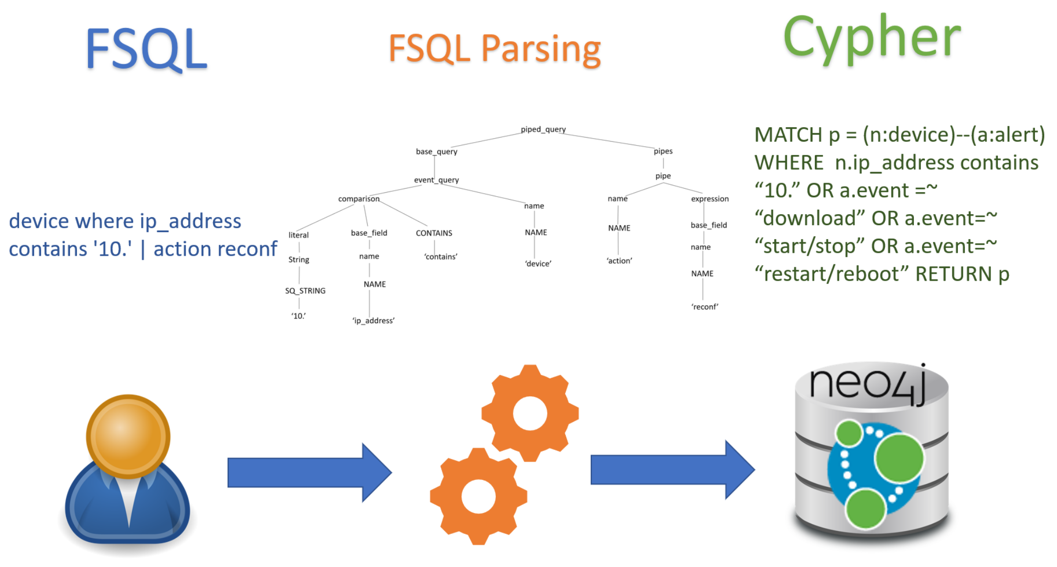
Figure 4.4: Example Grammar Parse Tree



4.8 Translation of IIQL to Cypher

Fig. 4.5 shows the process of how the IIQL query formulation, translation, and execution happens. The analyst formulates a query in IIQL, which is first parsed by EQL's parser according to the IIQL grammar, after which an Abstract Syntax Tree is generated. This is essentially the syntactic check of the input IIQL syntax. Once the syntax is checked, the query is then translated by the translation engine to the native database query language of Neo4j (Cypher) by modifying and embedding the tokens of the input IIQL query into the respective Cypher query. This allows for formulating flexible and expressive queries while still maintaining a simple and readable syntax.

Figure 4.5: IIQL Query Processing



```
device where ip_address
contains '10.' | action reconf
```

```
MATCH p = (n:device)--(a:alert)
WHERE  n.ip_address contains
“10.” OR a.event =~
“download” OR a.event=~
“start/stop” OR a.event=~
“restart/reboot” RETURN p
```

4.9 The IIQL prototype

This section elaborates on the developed prototype with the different operators supported with screenshots of the results. The IIQL engine can display results in three formats namely,

1. JSON list
2. JSON table
3. JSON graph format (with nodes and relationships)

4.9.1 An example IIQL query in Basic Mode

Use Case

Get devices starting with OS ’Windows’ sorted by Purdue Level and return the top 4 results.

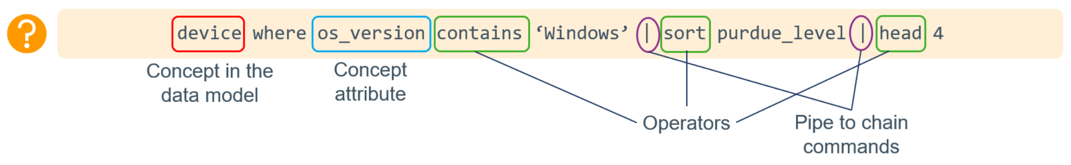
IIQL Query

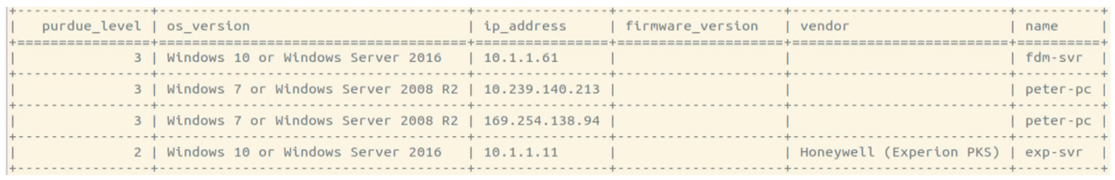
```
device where os_version contains 'Windows' | sort purdue_level | head 4
```

Results

The screenshot in Fig. 4.6 is from the Command Line Interface (CLI) of the IIQL engine.

Figure 4.6: An example IIQL query in Basic Mode


device where os_version contains 'Windows' | sort purdue_level | head 4
Concept in the data model
Concept attribute
Operators
Pipe to chain commands



```
+----------------+-------------------------------------+-----------------+--------------------+---------------------------+-----------+
|   purdue_level | os_version                          | ip_address      | firmware_version   | vendor                    | name      |
+================+=====================================+=================+====================+===========================+===========+
|              3 | Windows 10 or Windows Server 2016   | 10.1.1.61       |                    |                           | fdm-svr   |
+----------------+-------------------------------------+-----------------+--------------------+---------------------------+-----------+
|              3 | Windows 7 or Windows Server 2008 R2 | 10.239.140.213  |                    |                           | peter-pc  |
+----------------+-------------------------------------+-----------------+--------------------+---------------------------+-----------+
|              3 | Windows 7 or Windows Server 2008 R2 | 169.254.138.94  |                    |                           | peter-pc  |
+----------------+-------------------------------------+-----------------+--------------------+---------------------------+-----------+
|              2 | Windows 10 or Windows Server 2016   | 10.1.1.11       |                    | Honeywell (Experion PKS)  | exp-svr   |
+----------------+-------------------------------------+-----------------+--------------------+---------------------------+-----------+
```

4.9.2 Example IIQL queries in Advanced Mode

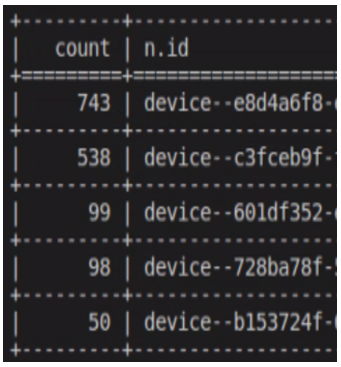
1. Operator: Influencer

Use-Case: Get top 5 influential devices
Query:

```
device where true | influencer | head 5
```

Result: The screenshot in Fig. 4.7 is from the Command Line Interface (CLI) of the IIQL engine. It shows the top 5 devices with the most number of connections in the network. The 'count' header represents the number of relationships of the particular device (whose respective 'id' in the network is displayed as well).

Figure 4.7: An example Influencer query



```
+---------+------------------
|   count | n.id
+=========+==================
|     743 | device--e8d4a6f8-
+---------+------------------
|     538 | device--c3fceb9f-
+---------+------------------
|      99 | device--601df352-
+---------+------------------
|      98 | device--728ba78f-
+---------+------------------
|      50 | device--b153724f-
+---------+------------------
```

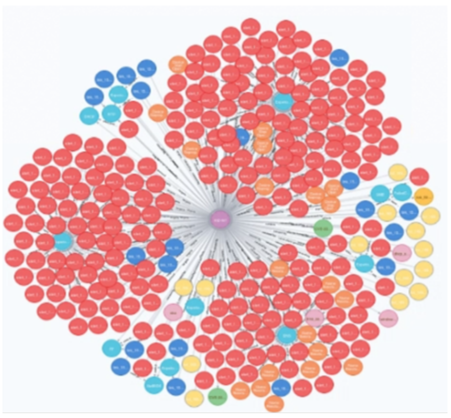
2. Operator: Explore

Use-Case: Free-text search of a device
Query:

```
explore 'exp-svr'
```

Result: The screenshot in Fig. 4.8 is from the Neo4j GUI. The central purple node is the device 'expsvr' and the entire first-order context around is it displayed in the form of alerts(in red), operations(in orange), host-change logs(in yellow), links (in dark blue), roles (in pink) and so on.

Figure 4.8: An example Explore query



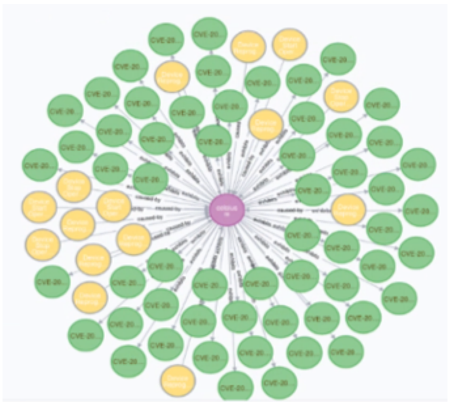
3. Operator: Investigate

Use-Case #1: Investigate a given device for vulnerabilities and network logs in the network
Query:

```
device where name == 'celsius m' | investigate operation, vulnerability
```

Result: The screenshot in Fig. 4.9 is from the Neo4j GUI. The yellow nodes represent the operations/network logs and the green nodes represent the vulnerabilities around the device 'celsius m' (in purple).

Figure 4.9: An example Investigate query

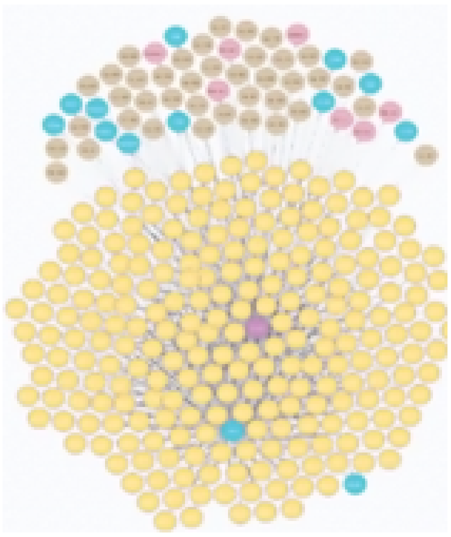


Use-Case #2: Investigate a device for all concepts it has a direct relationship with.
Query:

```
device where ip_address == '192.168.25.104' | investigate
```

Result: The query returns the entire context, consisting of all alerts, change logs, risks, roles, etc. that is linked to '192.168.25.104' (central node in purple). The screenshot in Fig. 4.10 is from the Neo4j GUI.

Figure 4.10: Another example Investigate query



4. Operator: Path

Use-Case #1: Find exploitable paths between a specific device and all the other devices in the network.
Query:

```
path by link [device where ip_address == '10.1.0.1'] [device where true] |
    investigate vulnerability
```

Result: The screenshot in Fig. 4.11 is from the Neo4j GUI. We can see the subgraph that shows us two exploitable paths between the device '10.1.0.1' (boxed) and all other devices in the network.
Use-Case #2: Find alerts connecting by source and destination from one device to others in the network.
Query:

```
path by alert [device where role == 'dcs'] [device where true]
```

Result: The screenshot in Fig. 4.12 is from the Neo4j GUI. We can see the subgraph that shows us how the central device is connected to other devices in the network through alerts (in pink).

5. Operator: Action

The 'action' operator makes use of the abstraction of events (event taxonomy) such as alerts, host change logs and network logs in Forescout's IDS (SilentDefense). The operator searches for

Figure 4.11: An example Path query

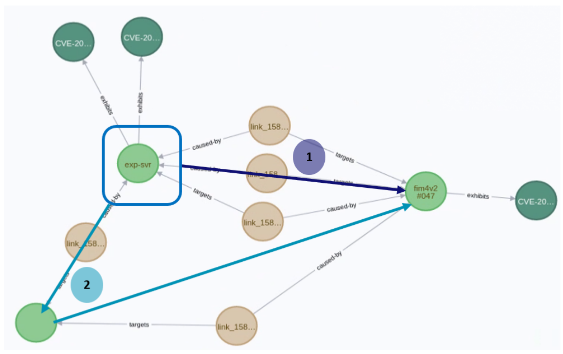
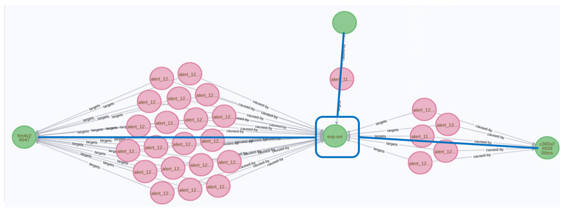


Figure 4.12: Another example Path query

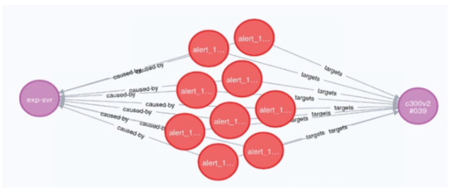


events with the respective regular expression mentioned in the abstraction for each event type.
Use-Case: Get devices starting with IP Address '10.*' performing a reconfiguration operation such as firmware upload, download and so on.
Query:

```
device where ip_address contains '10.' | action dangerous
```

Result: The screenshot in Fig. 4.13 is from the Command Line Interface (CLI) of the IIQL engine. The two devices in the network, which are involved in dangerous operations (alerts in red) are shown.

Figure 4.13: An example Action query

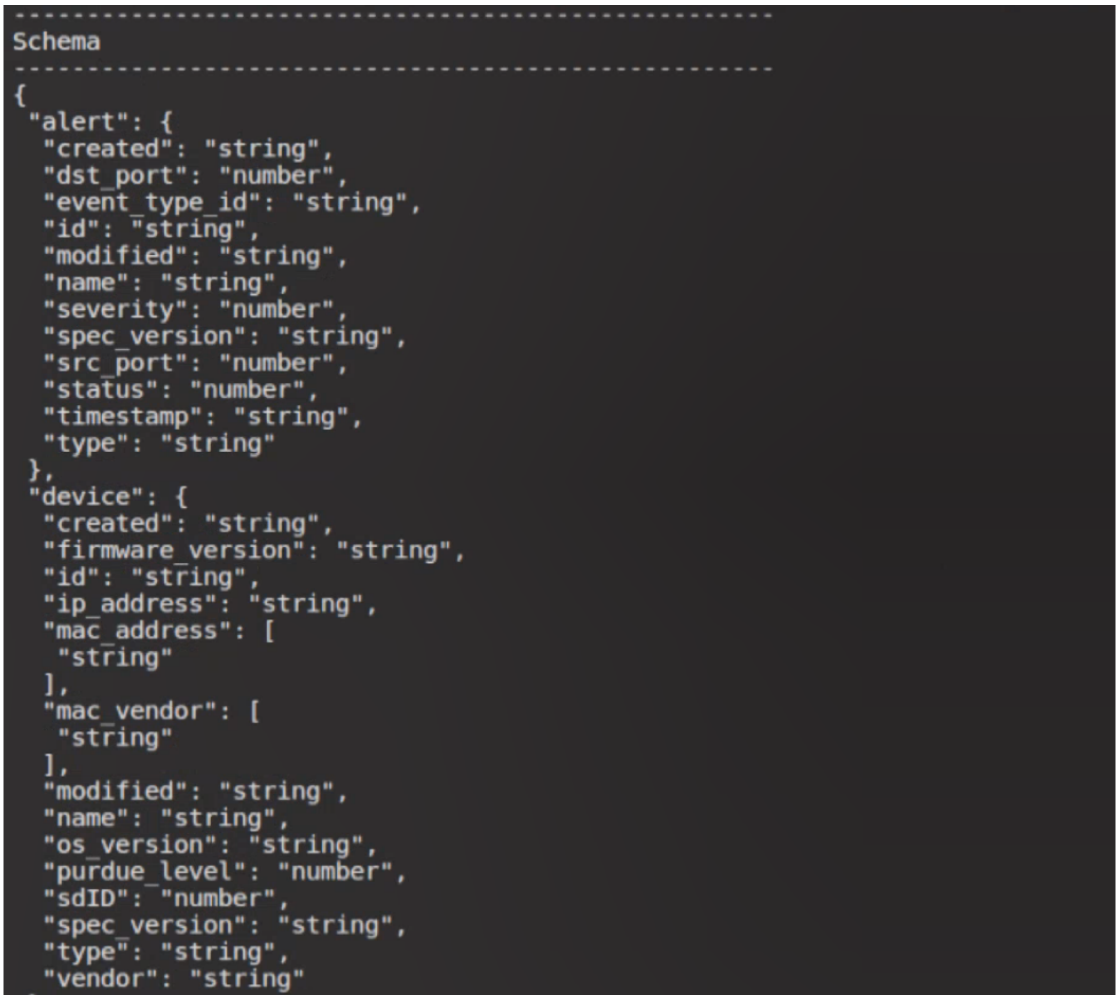


4.9.3 Interpreting the Data Model with IIQL

A noteworthy feature of IIQL is that, it automatically interprets and learns the underlying data model. Therefore, the user can view the list of available concepts and their attributes in JSON formate to query in Neo4j. This feature is leveraged from the base language it was developed from - EQL. A screenshot of the schema displayed in the IIQL Engine CLI is shown in Fig. 4.14.

The concepts and the attributes under each concept, along with their data types, can be seen in the image.

Figure 4.14: Display of underlying data schema



```
-----------------------------------------------------
Schema
-----------------------------------------------------
{
 "alert": {
  "created": "string",
  "dst_port": "number",
  "event_type_id": "string",
  "id": "string",
  "modified": "string",
  "name": "string",
  "severity": "number",
  "spec_version": "string",
  "src_port": "number",
  "status": "number",
  "timestamp": "string",
  "type": "string"
 },
 "device": {
  "created": "string",
  "firmware_version": "string",
  "id": "string",
  "ip_address": "string",
  "mac_address": [
   "string"
  ],
  "mac_vendor": [
   "string"
  ],
  "modified": "string",
  "name": "string",
  "os_version": "string",
  "purdue_level": "number",
  "sdID": "number",
  "spec_version": "string",
  "type": "string",
  "vendor": "string"
```

Chapter 5

Case Studies: IIQL in Practice

This chapter describes in detail how the developed query language prototype was applied to investigate a real-time OT incident, namely the Stuxnet malware incident. The chapter also discusses how this prototype could be used to develop playbooks to identify the presence of MITRE ICS ATT&CK Tactics and Techniques in an organization's network.

5.1 IIQL - Case Studies

This section describes how the IIQL prototype was used to investigate the OT incident - the Stuxnet Malware Attack and investigate the presence of a MITRE ATT&CK TTP in an organization's network.

Technologies used

- Command Line Interface of the developed prototype - the IIQL transpilation engine built completely with Python 3.7.
- Neo4j Graph Database populated with the appropriate dataset for visualization.

5.1.1 Investigation of Stuxnet Attack with IIQL

This is a detailed case study where IIQL is used to analyze an attack scenario step-by-step in the industrial domain.

Aim

The goal of the case study is to showcase the power of IIQL to investigate the Stuxnet malware attack.

Dataset Used

The Stuxnet Dataset (containing 57 hosts and 1452 alerts)

Methodology

An analyst investigating this dataset, can use the following sequence of IIQL queries. The equivalent Cypher queries are also mentioned. For context-oriented queries, SQL queries cannot be formulated because there is a need to take into account, nodes and relationships for contextual information, which is possible in graph databases and not possible in relational databases.
There are multiple scenarios to analyze, but the method followed here investigates the spread of the Stuxnet malware from the devices in the higher Purdue Levels, where it originated, to the critical PLCs in the lower levels.

Investigating the critical assets in the network

In order to identify the critical assets, we look for those with a high or medium risk factor with possible known vulnerabilities. The IIQL query used (to query the Neo4j graph database) is:

```
risk where risk_label in ('High', 'Medium') | investigate device
```

The equivalent Cypher query for this IIQL query is:

```
MATCH p = (n:risk)--(:device) WHERE n.risk_label in ['High', 'Medium'] RETURN p
```

This query returns all the PLCs in Purdue Level 1. These can be further investigated for vulnerabilities.
Then, we want to look for vulnerable devices. The IIQL query used for this is:

```
vulnerability where true | investigate device | sort cvss | head 10
```

The equivalent Cypher query for this IIQL query is:

```
MATCH p = (n:vulnerability)--(:device) RETURN p ORDER BY n.cvss DESC LIMIT 10
```

This query returns 6 vulnerable devices in the 192.168.5.1/24 subnet. Device with IP "192.168.5.62" has several vulnerabilities with high CVSS scores.

Investigating upto the point of malware infection

Now we want to investigate if this device with IP "192.168.5.62" is linked to any devices performing dangerous operations. The IIQL query for this is:

```
path by link [device where ip_address == '192.168.5.62'][device where true] |
    action dangerous
```

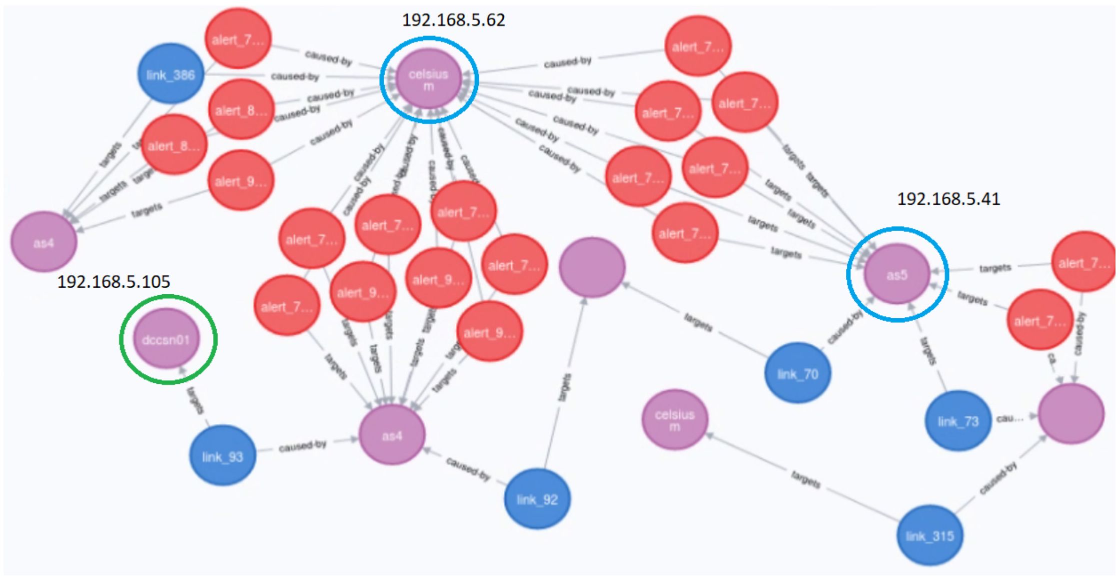
The equivalent Cypher query for this IIQL query is:

```
MATCH p = (b:device)--(:link)--(c:device)--(a:alert) WHERE b.ip_address = "
    192.168.5.62" AND a.event_type_id =~ "itl_ops_pdop(.*)" RETURN p
```

The device was indeed sending device reprogram commands to the PLCs. One of these PLCs ("192.168.5.41") were found to be linked to a non-vulnerable device "192.168.5.105". This subgraph obtained from the Neo4j GUI is shown in Fig. 5.1.
Now, we want to explore (free-text query) this non-vulnerable device "192.168.5.105" to obtain information about its role, activities, etc. The IIQL query for this is:

```
explore '192.168.5.105'
```

Figure 5.1: Subgraph showing the path between the three devices



The equivalent Cypher query for this IIQL query is:

```
call db.index.fulltext.queryNodes("nodes","192.168.5.105") yield node match p = (
    node)--() return p
```

An alert was found for this device which indicated that it was trying to access a blacklisted domain, outside the organization's network. Next, we want to find the source of this blacklisted domain access alert. The IIQL query for this is:

```
device where true | action blacklist
```

The equivalent Cypher query for this IIQL query is:

```
MATCH p = (n:device)--(a:alert) WHERE a.event_type_id =~ "itl_sec_udb.*" RETURN p
```

The source "192.168.5.162" and two other devices with similar alerts in Purdue Level 3 was found. In total 4 devices and 2 alerts were found using this query.
Now we want to check if these 4 devices are related in any way. The IIQL query for this is:

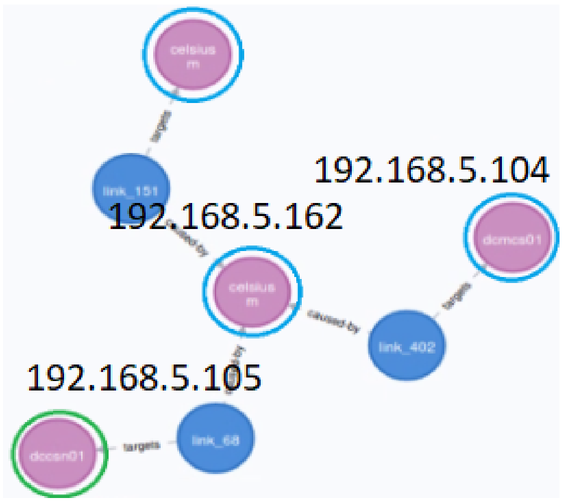
```
path by link
[device where ip_address == '192.168.5.162'][device where true]
```

The equivalent Cypher query for this IIQL query is:

```
MATCH p = (b:device)--(:link)--(c:device) WHERE b.ip_address = "192.168.5.162"
    RETURN p
```

Indeed "192.168.5.162" is related to the three devices via links. This subgraph obtained from the Neo4j GUI is shown in Fig. 5.2.
Next, we want to check the number of relationships (influentiality) of each of the 4 devices found in the previous step. The IIQL query for this is:

Figure 5.2: Subgraph showing the path between the four devices



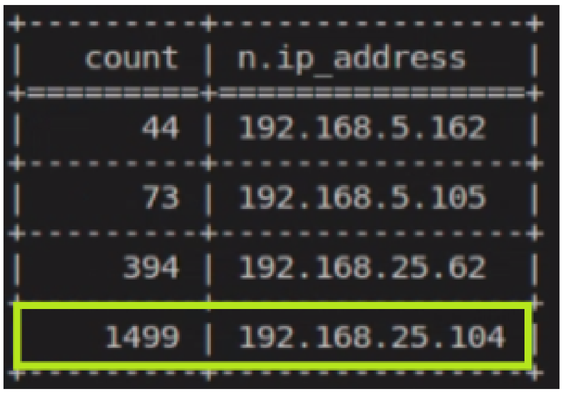
```
device where ip_address in
('192.168.25.104','192.168.25.62','192.168.5.105',
'192.168.5.162')| influencer
```

The equivalent Cypher query for this IIQL query is:

```
MATCH p = (n:device)--() WHERE n.ip_address in ['192.168.25.104', '192.168.25.62',
     '192.168.5.105', '192.168.5.162'] RETURN count(p) as count, n.ip_address
    ORDER BY count
```

Thus, “192.168.25.104” is the most influential device having 1499 relationships. The results from the CLI of IIQL engine is shown in Fig. 5.3.

Figure 5.3: Influentiality of the four devices



```
+---------+-----------------+
|  count  | n.ip_address    |
+=========+=================+
|      44 | 192.168.5.162   |
+---------+-----------------+
|      73 | 192.168.5.105   |
+---------+-----------------+
|     394 | 192.168.25.62   |
+---------+-----------------+
|    1499 | 192.168.25.104  |
+---------+-----------------+
```

Finally, we further investigate this device. The IIQL query for this is:

```
device where ip_address == '192.168.25.104' |investigate alert
```

The equivalent Cypher query for this IIQL query is:

```
MATCH p = (n:device)--(:alert) WHERE n.ip_address = "192.168.25.104" RETURN p
```

We see that “192.168.25.104” has been involved in a lot of events, tried to breach firewall rules by attempting to contact internet hosts and is part of a huge malware peer-to-peer network of 8 hosts in 192.168.25.1/24 subnet.

Observations

1. It was observed that the devices with a vulnerability were performing device configuration operations. Most of these operations were commanded by the master “192.168.5.62”. These were critical assets present in the lower Purdue Levels (1 and 2).
2. The “192.168.5.105” device was a Level 3 device connected to the “plc” slave in Level 1.
3. Some cross-network flows were observed between this “192.168.5.105” and the malware-infected device “192.168.25.104”.
4. This suggests that the infected “192.168.25.104” might have sent device reprogram commands to disrupt the activities of the lowest level “plc” via “192.168.5.105” and “192.168.5.162”.

Possible Conclusion

Thus, this device with IP “192.168.25.104” could have been possible the point of infection of the Stuxnet malware.

5.1.2 MITRE ATT&CK Case Studies

The ICS ATT&CK Matrix

The MITRE ICS ATT&CK framework [29] is used to model an adversary’s Tactics, Techniques and Procedures (TTP) that the adversary uses in order to perform a cyber attack on an Industrial Control Systems (ICS) Network. It is different from the Enterprise ATT&CK matrix in the sense that it focuses on assets in the lower Purdue Levels such as PLCs, RTUs, SCADA masters and other physical and logical control system devices. The ICS ATT&CK matrix is a subset of the Enterprise ATT&CK matrix and is focused on Cyber Threat Intelligence (CTI) in ICS Networks. The use-cases behind the motivation of this ATT&CK matrix is to model adversary behaviour in an ICS network, effective Security Operation Center (SOC) investigation, Defense Gap Enrichment as they call it, to name a few. This knowledge base can be used by us to investigate activities pertaining to specific phases of the Cyber Kill Chain like Discovery, Execution, Lateral Movement to name a few. Each of these Tactics (Kill Chain Phases) has several Techniques like Network Sniffing, Network Service Scanning (for Discovery), and so on. Each Technique is an Attack Pattern in the data model. In this case study, we will use this Attack Patterns along with other general queries to investigate some of these Tactics and confirm its presence in the organization’s network.

Goal

The goal of this case study is create a playbook of some sort, that contains a list of generic queries in IIQL to investigate the ICS ATT&CK TTPs, that can then be used across multiple organizations and specialized by the security analyst, if needed. This playbook is not a complete guide to investigate, analyze and defend against these Tactics, but rather a small and efficient starting point to confirm the presence of these Tactics in the network. It will identify devices that might initiate these Tactics in the network. These devices will then have to be investigated in detail by the security analyst to take meaningful course of action to defend against the attacks. The three Tactics that were chosen to be evaluated using IIQL are Discovery, Initial Access and Inhibit Response Function. The motivation behind this choice was that there was sufficient data from the IDS pertaining to these tactics.

Tactic #1: Discovery

The 'Discovery' tactic consists of techniques that attackers use to survey a network and select their targets for further exploitation. From reading the techniques under this tactic, and their description from the MITRE ATT&CK [29], the following keywords can be looked for: port scans, ICMP ping scans and Domain Controller devices. The query sequence that can be followed to find the presence of this tactic in the network can be:

1. Look for devices perform port scans using 'action' operator.
2. Look for Domain Controllers (PDCs).
3. Look for devices involved in reconnaissance alerts using 'explore' operator.
   If the devices from steps 1,2 and 3 are same or related, then we have our targets for further investigation.

Executing the IIQL playbook in the Stuxnet dataset

1. Technique identified: Network Service Scanning
IIQL query:

```
device where true | action port_scan
```

Result: Device (dcmcs01) with IP '192.168.25.104' has caused two alerts by port scanning (reconnaissance). The screenshot from the Neo4j GUI is shown in Fig. 5.4.

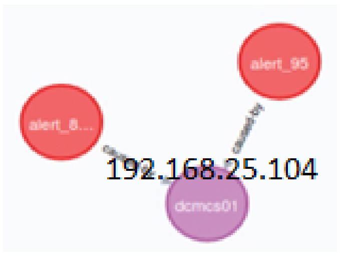


Figure 5.4: Device causing two reconnaisance alerts

2. Technique identified: Control Device Identification
IIQL query:

```
device where role == 'windows_pdc' | investigate
```

Result: On further investigating the 'windows_pdc' ('dcmcs01') device, it was found that this device was part of many alerts that were raised when it tried to connect to the internet-connected devices and has been a part of a malware p2p network. The screenshot from the Neo4j GUI is

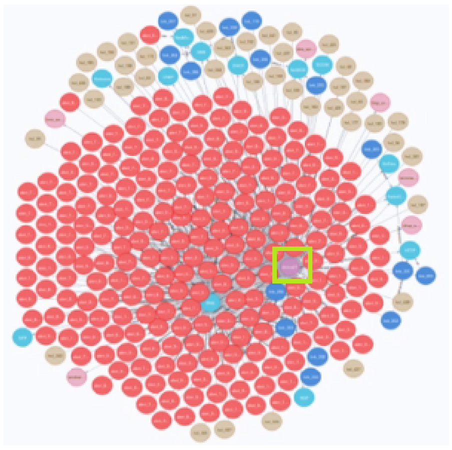


Figure 5.5: Context around the device

shown in Fig. 5.5.
IIQL query:

```
path by alert[device where role == 'windows_pdc'][device where true]
```

Result: On seeing what devices are connected to this device, we can see two internet-connected devices(with role "root-dns server") with IP addresses "193.0.14.129" and "198.41.0.4" and a device with unknown role (fishy) with IP address "198.32.64.12" through public communication alerts and to some devices, in the "192.168.25.1/24" subnet, in Purdue Level 2 through malware alerts (boxed in red in the image). The screenshot from the Neo4j GUI is shown in Fig. 5.6.

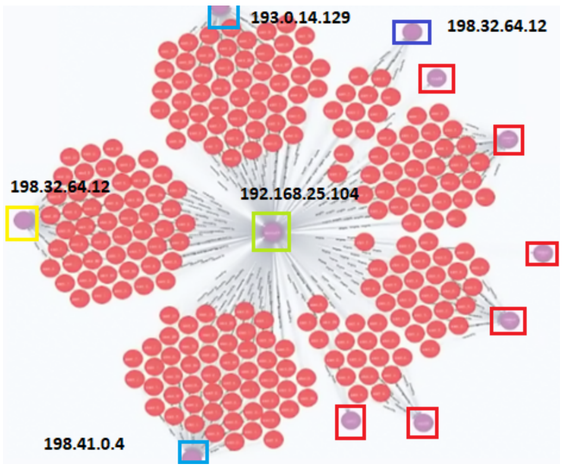


Figure 5.6: Path between the PDC and other devices

Conclusion: Therefore, we see that this particular device (dcmcs01) "192.168.25.104" has been occurring in all the previous steps, being a Domain Controller as well as performing port scans and also being involved in malicious alerts, thereby confirming the presence of this tactic in the network.

Tactic #2: Initial Access

The 'Initial Access' tactic consists of techniques that an adversary may use as entry points to obtain an initial foothold in the network. From reading the techniques under this tactic, and their description from the MITRE ATT&CK [29], the following keywords can be looked for: Engineering Workstations, Internet IPs, Public communication alerts, SMB protocol communication, database servers and RDP/DCOM communication. The query sequence that can be followed to find the presence of this tactic in the network can be:

1. Look for devices with links communicating via the SMB protocol.
2. Look for devices involved in public communication alerts using 'action' operator.
3. Look for devices with RDP/DCOM protocol communication.
   If any of the devices in steps 1,2 and 3 are either Engineering Workstations or Data Historians in the network, they are our targets for investigation.

Executing the IIQL playbook in the Stuxnet dataset

1. Technique identified: External Remote Services
IIQL query:

```
l7protocol where name in ('SMB','RDP','DCOM')| investigate device
```

Result: On querying for devices that use remote discovery protocols such as SMB,RDP and DCOM, a number of devices were displayed. The screenshot from the Neo4j GUI is shown in Fig. 5.7.

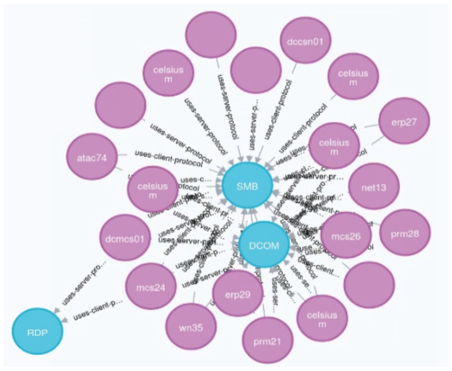


Figure 5.7: Devices using remote discovery protocols

2. Technique identified: Internet Accessible Devices
IIQL query:

```
device where true | action public
```

Result: Device (dcmcs01) with IP "192.168.25.104" has been causing these alerts trying to connect to internet connected devices. The screenshot from the Neo4j GUI is shown in Fig. 5.8.

3. Technique identified: Engineering Workstation Compromise
IIQL query:

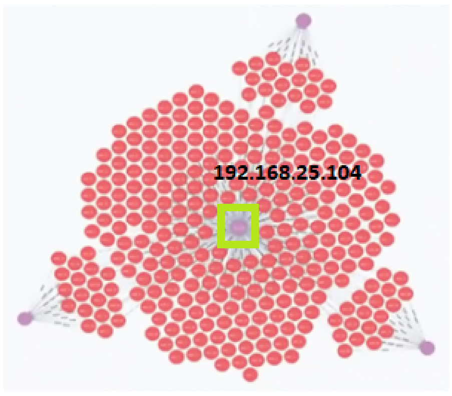


Figure 5.8: Device trying to connect to internet connected devices

```
device where role == 'ews' | investigate alert , operation
```

Result: Device (mcs24) with IP "192.168.25.24" has the role of EWS (engineering workstation) with OS 'Windows XP'. It has 7 network operations and 52 high-severity alerts. The screenshot from the Neo4j GUI is shown in Fig. 5.9.

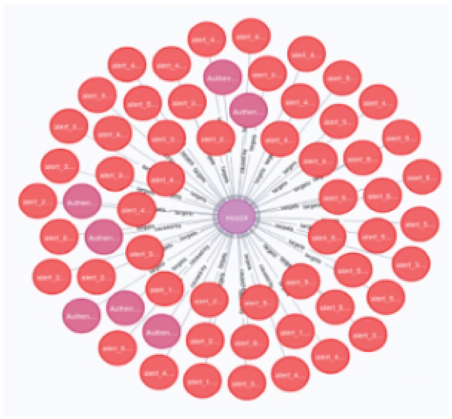


Figure 5.9: Engineering Workstation

4. Technique identified: Data Historian Compromise
IIQL query:

```
device where role == 'database_server' | investigate alert
```

Result: Device (wn35) with IP "192.168.25.35" has the role of Database Server (data historian) with OS 'Windows XP'. It has no network operations and 79 high-severity alerts. The screenshot from the Neo4j GUI is shown in Fig. 5.10.

Conclusion: Both these devices (EWS and Data Historian) have repeatedly occurred in the previous steps, suggesting that they were initial points of access, thereby confirming the presence of this tactic in the network.

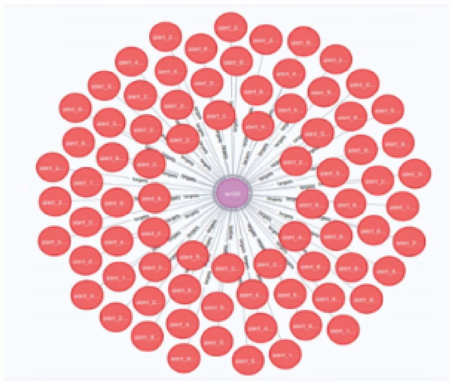


Figure 5.10: Data Historian

Tactic #3: Inhibit Response Function

The 'Inhibit Response Function' tactic consists of techniques that an attacker may use to hinder the protection mechanisms from responding to failures. From reading the techniques under this tactic, and their description from the MITRE ATT&CK [29], the following keywords can be looked for: DOS alerts, dangerous operations, Device Reprogram alerts, Firmware Updates and proprietary protocol communication (such as STEP7 messages in Stuxnet attack). The query sequence that can be followed to find the presence of this tactic in the network can be:

1. Look for devices performing dangerous OT operations using 'action' operator.
2. Look for devices involved in denial of service operations using 'action' operator.
3. Look for devices performing firmware updates, downloads using 'action' operator.
   If any of the devices in steps 1,2 and 3 are same or related, they are our targets for investigation.

Executing the IIQL playbook in the Stuxnet dataset

1. Techniques identified: Program Download and Device Restart/Shutdown
IIQL queries:

```
device where true | action reconfiguration
```

Result: These 5 devices have been involved in program upload and download operations. The device 'celsius m' with IP Address "192.168.5.62" causes program upload alerts targeting PLCs such as 'as4' and 'as5'. This device uploads a program to the PLCs. The screenshot from the Neo4j GUI is shown in Fig. 5.11.

2. Technique identified: Denial of Service
IIQL queries:

```
device where true | action denial
```

Result: Device (Celsius m) with IP "192.168.25.22" has caused DOS alerts targeting device (prm28) with IP "192.168.67.28" which is a Windows workstation. This device uploads a program from the PLCs. The screenshot from the Neo4j GUI is shown in Fig. 5.12.

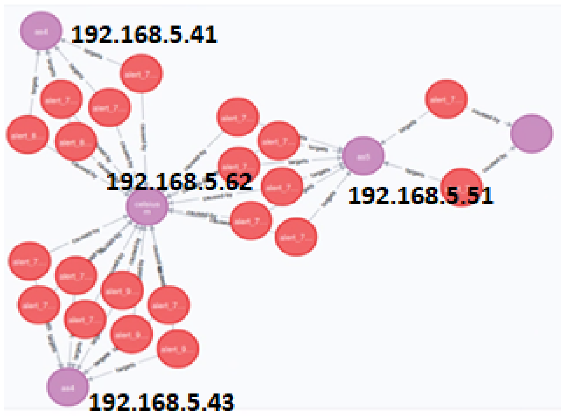


Figure 5.11: Devices involved in reconfiguration alerts

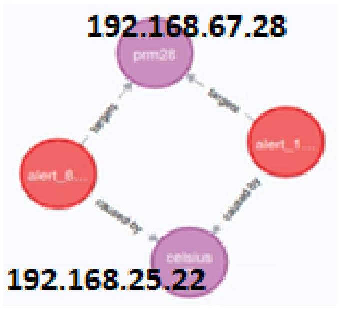


Figure 5.12: Devices involved in DOS alerts

3. Technique identified: Activate Firmware Update Mode
IIQL queries:

```
hostchangelog where event_type_id contains 'new_fw' | investigate device
```

Result: Querying the network for assets that changed firmware. The screenshot from the Neo4j GUI is shown in Fig. 5.13.

Conclusion: We see that the devices that have changed firmware (as4, as5) have also been involved in program download alerts, thereby confirming the presence of this tactic in the network.

Insights from the Analysis

Apart from Stuxnet, this playbook was tried on different real-world datasets. The common insights from the analysis are as follows:

- Usually, it is the same set of devices involved in all the techniques for a given tactic. Most of these devices were in the lower Purdue Levels as DCS masters, or EWS, or a PLC/RTU sometimes.

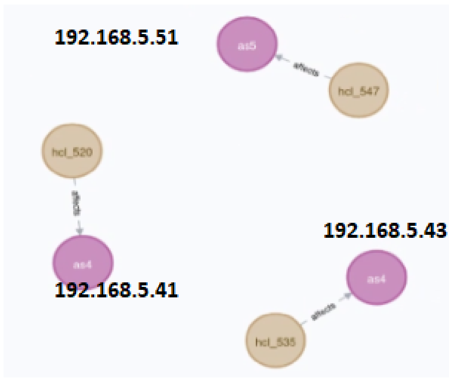


Figure 5.13: Devices involved in firmware updates

- These suspicious devices have a lot of host change logs for different client and server ports, i.e, they have varied communication patterns.
- For the “Inhibit Response Function” tactic, the devices with host change logs pertaining to ‘new_fw’ were performing dangerous OT operations.
- Devices that use the “Discovery” tactic usually tend to cause a denial-of-service on the network, thereby compromising the availability of other assets.
- For the “Initial Access” tactic, the suspicious devices always seem to have SMB/DCOM as their client/server protocols.

Conclusion

The investigation of MITRE ATT&CK TTPs with IIQL provides cyber threat intelligence capabilities, taking into account possible attack scenarios, and gives an idea of how prepared an organization is to detect and respond to intrusions. In this way, an organization knows if it is able to meet its cybersecurity needs and be ready if or when malicious actors strike. This provides a roadmap for defenders to apply against their operational controls to weigh their strengths and weaknesses against adversaries. It can be used to assess the gaps in their defense policies and possible attack points in their network. Summarizing the advantages of using IIQL with MITRE ATT&CK:

- Detailed insights into devices using the ATT&CK Tactics and their targets.
- Identify the point of breach in the network and how the attacker moved laterally from one device to another and so on.
- Powerful post-mortem and proactive incident investigation.

Chapter 6

Validation of IIQL

This section describes how the proposed prototype was validated with some of the internal domain experts at Forescout, in the form of an online survey.

6.1 User Studies

In order to validate the query language, a one-to-one interview was held with some experts in Forescout. In this interview, a presentation containing the overview of IIQL, with its operators, syntax and functionality with some sample queries and results, was given, followed by them filling up an online survey. This survey had 17 questions in total, comprising 14 questions that test their IIQL knowledge and 3 questions to gauge their overall user experience with IIQL.

6.1.1 Validation Questionnaire

This subsection enlists the sections of questions that were given to the domain experts in Forescout to answer based on their user experience with IIQL.

1. For the initial set of use-cases, the users were asked to give the IIQL query for achieving them. The equivalent Cypher and SQL query (if applicable for the use-case) were also displayed for comparison. Some of the use-cases that were given to the user were:
   - Find vulnerable PLCs;
   - Find exploitable paths from one device to other devices in the network;
   - Find devices that perform reconfiguration, connected to an engineering workstation;
   - Find vulnerable Windows devices that perform port scans;
2. For certain queries, the users were asked to derive the use-case that they think the queries would achieve, in natural language. Some of the queries that were given to the user were:
   - *device where role == 'plc' | action dangerous | filter alert.severity > 2*
   - *path by link [device where ip == '192.168.5.62'] [device where true] | action dangerous*
   - *vulnerability where true | investigate device | sort cvss | head 5*
3. The rest of the questions were for understanding the users' IIQL overall experience, thereby intuitively measuring metrics such as:

- **Expressiveness:** The powerfulness of query formulation to match one's needs. This metric is basically the measure of the range of functionality that can be expressed. The user can choose a score from a scale of 1 - 5.
- **Usability:** The ease of use of the query language. This metric is to measure the learning curve. The user can choose from 3 options such as Easy, Medium or Hard.
- **Accuracy:** The degree to which the query results conform to the standard expected outcome. This metric is to measure the overall satisfaction of the query results. The user can choose a score from a scale of 1 - 5.

6.2 Results from the Survey

This section summarizes the results from the survey responses and derive final conclusions from the developed prototype.

6.2.1 Overview of User Performance

The table 6.1 below depicts the number of correctly answered questions out of a total of 14 questions in the survey for each participant. As can be seen from the table, 5 out of 6 participants

Table 6.1: Number of correctly answered questions per user

Participant	Correctly Answered Questions
1	14
2	14
3	14
4	14
5	13
6	14

got all 14 questions testing IIQL knowledge correct. The users were able to quickly grasp the language, and its syntax.

6.2.2 Expressiveness

Out of 6 users, 3 gave a score of 5/5 and 3 gave a score of 4/5 for expressiveness of IIQL. Therefore, the overall average score is 4.5/5. One of the users thought that the language could have explored the possibility of filtering events (such as alerts) by timestamps.

6.2.3 Usability

Out of 6 users, 3 thought learning the syntax of IIQL (within an hour) was of 'Medium' difficulty and 3 thought it was of 'Easy' difficulty. The common feedback that was received from the users who chose 'Medium' was that, there could have been a cheatsheet for IIQL with a graph of the underlying data model consisting of a detailed list of concepts and attributes that can be queried for, in order to know how the concepts were connected to each other. An inference of this feedback is that, there has to be a minimal knowledge of the data model to make more specific queries in IIQL.

6.2.4 Accuracy

Out of 6 users, 3 gave a score of 3/5, 2 gave a score of 5/5 and 1 gave a score of 4/5 for accuracy of IIQL. Thus, the overall average score for accuracy is 4/5. Some of the user feedback obtained from this respect is that:

- The functionality of the *investigate*, *filter* and *action* operators can be combined into one, as they might be confusing sometimes.
- There could have been a possibility of using a *filter* operator, before a *path* operator, in order to filter the results and then obtain a sub-graph from the smaller subset of the data. In this way, the queries would perform faster on very large graph databases. However, in the developed prototype, filtering can only be done after the *path* operator, owing to the fact that it is based on the EQL syntax, and it has this limitation.

6.3 Summary of Validation

As mentioned above, a one-to-one interview was held with some experts in Forescout where IIQL was presented to them, followed by them filling up an online survey (testing both their knowledge of IIQL and overall experience with the language). The validation was done with the initial set of sample use-cases; however, the results are not tailored to these set of use-cases alone and can be extended. For instance, the operators such as *influencer* and the *duration()* function were not part of the initial set of use-cases, yet can solve the issues of narrowing down focus for investigation.

6.3.1 Other Remarks from Experts

Additionally, to score for Expressiveness of IIQL, the experts were asked to think of other use-cases that they would like IIQL to achieve. One of them came up with the use-case of being able to provide an aggregated statistics of the number of alerts per asset or group alerts by a user-specified attribute. However, this is unfortunately not yet possible with the prototype and can be seen as a scope for future work on the prototype.

Chapter 7

Conclusions

This chapter discusses the concluding remarks from the evaluation and validation steps of the developed prototype, along with directions or ways to extend it further in future.

7.1 Query Performance

A number of sample IIQL queries were executed on different datasets on both the Neo4j and SQL databases, and the process time for each query was measured. This section summarizes the results obtained in the form of certain statistics.

7.1.1 Performance in the Neo4j graph database

Fig. 7.1 below shows the query process times for each type of IIQL query on Neo4j, on three different datasets namely:

1. Dataset 1 (24 hosts, 1239 alerts)
2. Dataset 2 (57 hosts, 1452 alerts)
3. Dataset 3 (7899 hosts, 140448 alerts)

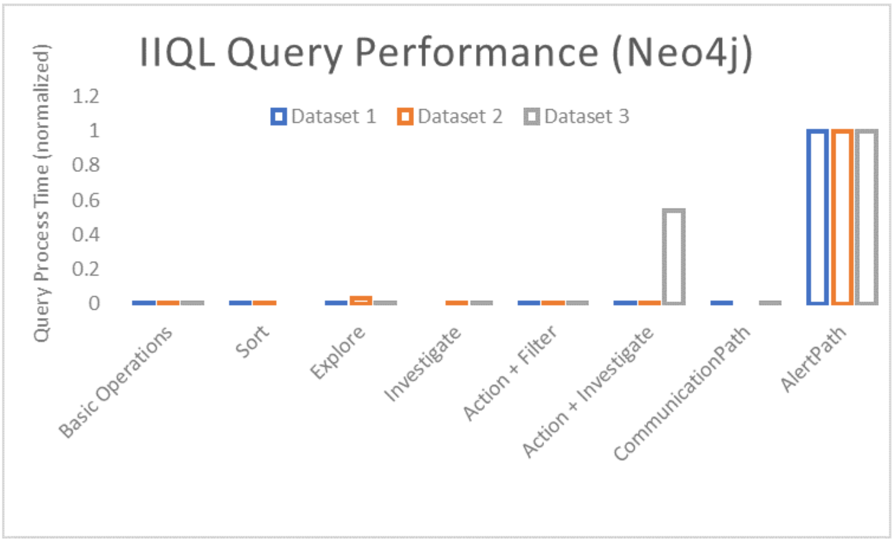
The process times are normalized within a scale of 0 to 1 in the chart, for better visualization, because `path by alert` queries took a very long time to execute in two datasets. As can be seen from the above chart 7.1, the `path by alert` queries take the most time to execute in all 3 datasets (because retrieving chains of alerts across hops takes a lot of time to execute). A more specific query/use-case such as finding alert path between two specific devices like `path by alert [device where name == 'exp-svr'][device where name == 'fdm-svr']` can definitely fetch results faster, instead of fetching all alert paths from a device to all other devices in the network.

7.1.2 Performance in the SQL database

Fig. 7.2 below shows the query process times for each type of IIQL query on SQL, on five different datasets namely:

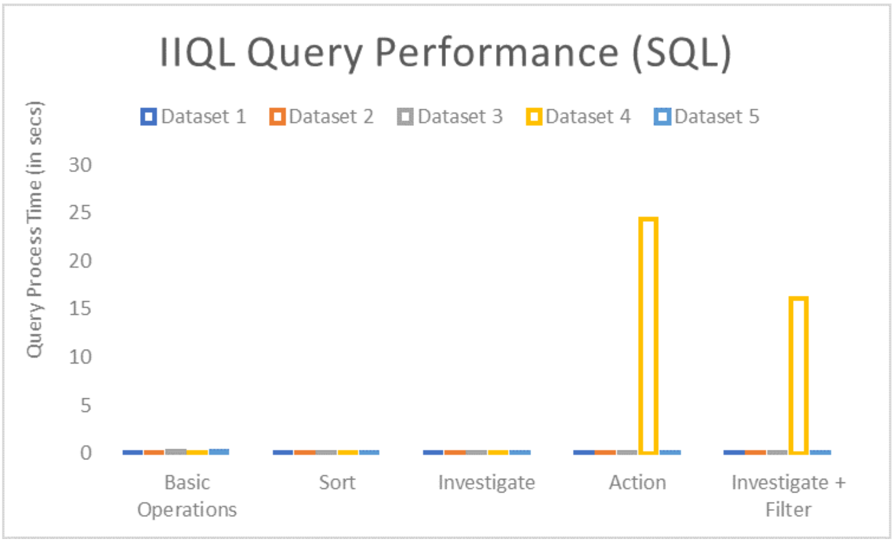
1. Dataset 1 (24 hosts, 1239 alerts)
2. Dataset 2 (57 hosts, 1452 alerts)

Figure 7.1: IIQL Query Performance in Neo4j



3. Dataset 3 (7899 hosts, 140448 alerts)
4. Dataset 4 (4799 hosts, 5918191 alerts)
5. Dataset 5 (330 hosts, 341166 alerts)

Figure 7.2: IIQL Query Performance in SQL



As can be seen from the above chart 7.2, the query performance of most queries are more or less the same, and naturally the performance time of *Investigate + Filter* and *Action* queries in Dataset 4 (depicted in yellow) is the longest, because it is the biggest dataset compared to the other four (with almost 6M alerts). Again, more specific queries would definitely yield results faster than generic queries.

7.2 Query Complexity

This section reiterates how IIQL was less complex and shorter in length than other languages such as SQL and Cypher with some example queries.

7.2.1 Comparison with SQL

This subsection highlights how some IIQL queries are less complex and shorter when compared to SQL queries. Table 7.1 enumerates some sample IIQL queries and their equivalent SQL queries. From the table, one can infer that an 8 line SQL query can be achieved in a 2-3 line IIQL query.

7.2.2 Comparison with Cypher

This subsection elaborates on how some IIQL queries are less complex and shorter when compared to Cypher queries. Table 7.2 enumerates some sample IIQL queries and their equivalent Cypher queries. From the table, one may infer that a user need not completely know the underlying data model to formulate IIQL queries (as opposed to formulating Cypher queries), and a very minimal knowledge is only required.

7.2.3 Conclusion

As can be tangibly seen from both the tables 7.2 and 7.1, IIQL queries are way shorter and less complicated and verbose than the respective Cypher or SQL queries, at the same time, expressive as the Cypher or SQL equivalent as well.

Table 7.1: IIQL queries and equivalent SQL queries

IIQL Query	**Equivalent SQL query**

hosts where ip contains ’192.168’
| investigate links
	SELECT hosts.ip, SUM(links.tx_bytes) as total_tx_bytes,
	ARRAY_AGG (DISTINCT links.proto) AS protocols_used
	FROM hosts
	INNER JOIN links
	ON hosts.id = links.src_host_id
	WHERE text(hosts.ip) LIKE ’%192.168%’
	GROUP BY hosts.ip;

hosts where true | action reconfiguration	SELECT distinct hosts.ip, hosts.purdue_level, hosts.os_version, hosts.criticality FROM hosts INNER JOIN alerts ON (hosts.ip::inet - ’0.0.0.0’::inet) = alerts.src_ip OR (hosts.ip::inet - ’0.0.0.0’::inet) = alerts.dst_ip WHERE alerts.event_type_id ∼* ’...........’ OR alerts.event_type_id ∼* ’..........’ OR alerts.event_type_id ∼* ’..........’;
hosts where criticality >3 and main_name contains ”celsius” | investigate vulnerability | filter vulnerability.cvss_score >7	WITH A AS (SELECT ip, jsonb_array_elements(cve_info) AS cves, criticality, main_name FROM hosts) SELECT ip, ARRAY_AGG(cves) cve_info FROM A WHERE (cves->>’cvss_score’) >text(7) AND criticality >3 AND text(main_name) LIKE ’%celsius%’ GROUP BY ip ;
hosts where main_name contains ’celsius’ | investigate alerts | filter alerts.severity >3	SELECT hosts.ip, ARRAY_AGG (alerts.event_type_id) alert_events FROM hosts INNER JOIN alerts ON (hosts.ip::inet - ’0.0.0.0’::inet) = alerts.src_ip OR (hosts.ip::inet - ’0.0.0.0’::inet) = alerts.dst_ip WHERE text(hosts.main_name) LIKE ’%celsius%’ AND alerts.severity >3 GROUP BY hosts.ip;

Table 7.2: IIQL queries and equivalent Cypher queries

IIQL Query	**Equivalent Cypher query**

path by link
[device where ip == ’192.168.5.62’]
[device where true]
| action dangerous
	MATCH p = (b:device)–(:link)–(c:device)–(d:alert)
	WHERE b.ip = ”192.168.5.62”
	AND
	d.event_type_id =∼”...*”
	RETURN p

path by alert
[device where main_role == ’dcs’]
[device where true]
	MATCH p = (:device)–(a:alert)–(b:device)–
	(l:alert)–(c:device)–(m:alert)–(:device)
	WHERE b.main_role = ”dcs”
	RETURN p

device where true
| action dangerous
| filter vulnerability.cvss == 10
	MATCH p = (a:vulnerability)–(n:device)–(d:alert)
	WHERE d.event_type_id =∼”...*”
	AND
	a.cvss = 10
	RETURN p

explore ’exp-svr’	CALL db.index.fulltext.queryNodes(”nodes”,”exp-svr”) YIELD node MATCH p = (node)–() RETURN p

device where true
| influencer
| head 5
	MATCH p = (n:device)–()
	RETURN count(p) as count, n.ip_address
	ORDER BY count
	DESC LIMIT 5

path by link [device where name == ’exp-svr’] [device where true]	MATCH p = (:vulnerability)–(b:device)– (:link)–(c:device)–(y:vulnerability) WHERE b.name = ”exp-svr” AND y.cvss >7 RETURN p

7.3 Answering the Research Question

This section talks about how the developed prototype answers the research question of the project, along with summarizing the pros and cons of the developed prototype.

7.3.1 Recalling the Research Question

The goal of this research was two-fold:

1. To develop a simple, yet expressive way to query the data collected by an IDS for convenient security incident investigation.
2. In addition to the syntax, it should also allow the security analyst to be able to explore the context around the data, at the same time, offer maximum abstraction to:
    - The underlying data and its complexity.
    - The native database backend language.

7.3.2 The Developed Prototype

The developed prototype of IIQL successfully answers the research goal of the project by providing abstraction, context and exploration of the underlying data, at the same time, having a usable and expressive syntax.

7.3.3 Advantages of IIQL

Thus, the major advantages of IIQL are:

1. It provides abstraction to the underlying data and its complexity.
2. It provides abstraction to native database backend query languages.
3. It comprises of automated underlying data schema learning capabilities, to help the user see what concepts and attributes they can query for.
4. It is simple and usable, yet has an expressive syntax.
5. It provides entire context around data collected from different sources.
6. It is easily extensible, and can be transpiled to any other query language in future.

7.3.4 Limitations of IIQL

Thus, the major limitations of IIQL are:

1. There is a need for a rich data model to successfully achieve exploration and to make the fullest and the best use of IIQL.
2. Possibilities for automated prioritization of attack paths is yet to be explored.
3. Filtering of events by timestamps is yet to be explored.
4. Possibility of providing an aggregated statistics of events per asset, or grouping events by a user-specified attribute is yet to be explored.

5. Since IIQL is based on EQL, the syntax cannot be stretched much and it is confined to a particular grammar, and for this reason, filtering cannot be done before another context-related operator.

6. The translation of IIQL to SQL is still limited, owing to the fact that SQL is a relational database, and not much context can be offered to the user, because that may involve expensive joins across tables.

7.4 Future Work

Though the developed prototype answers the research question successfully, it still has room for further improvement. Some of the possible directions of betterment may come from the following aspects.

7.4.1 Automated Prioritization of Attack Paths

The prototype currently only allows the user to look for exploitable paths from one asset to other assets, or between two specific assets in the network. However, it would be more useful if there was a mechanism to automatically output the list of exploitable paths in the network ordered by criticality of the path. The more critical assets and the more critical the vulnerabilities involved in the path are, the more priority it shall be given. This feature would make a security analyst's job all the more easier, to narrow down and focus on the paths that are the most critical, instead of starting investigation on the less critical paths.

7.4.2 Time Filtering of Events

The prototype currently offers only the possibility of sorting links or assets that are the most active or inactive in the network, by their timestamps but no possibility for filtering events by time, which is a very useful feature. Displaying only alerts within a particular time interval is very powerful as it enables the security analyst to narrow down his focus on the alerts that have occured contiguously than the alerts that have occured much later. Additionally, this will also reduce the number of alerts displayed to the user as well, thereby avoiding clutter and allowing more readability.

7.4.3 Aggregated Statistics of Events per Asset

The prototype currently does not offer the possibility of providing an aggregation of events per asset or grouping events by a user-specified attribute. For instance, to be able to find all devices involved in a dangerous operation grouped together by the type of dangerous operation that they are involved in (such as firmware download). This feature would be useful to an analyst if he wants to view the aggregated statistics of events per asset in the network.

Bibliography

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L Wiener. The lorel query language for semistructured data. *International journal on digital libraries*, 1(1):68–88, 1997. 64

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017. 63

[3] Apache. Apache lucene - query parser syntax. `https://lucene.apache.org/core/2_9_4/queryparsersyntax.html`. [last accessed: 2020-02-07]. 65

[4] James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and semantic web query languages: A survey. In *Reasoning Web*, pages 35–133. Springer, 2005. 63, 64

[5] Chris Bizer. Triql-a query language for named graphs. *http://www. wiwiss. fu-berlin. de/suhl/bizer/TriQL/*, 2004. 64

[6] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. Xquery 1.0: An xml query language. 2002. 64

[7] Angela Bonifati and Stefano Ceri. Comparative analysis of five xml query languages. *ACM Sigmod Record*, 29(1):68–79, 2000. 11

[8] Jeen Broekstra and Arjohn Kampman. Serql: A second generation rdf query language. In *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, pages 13–14, 2003. 64

[9] Stefano Ceri, Sara Comai, Ernesto Damiani, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Xml-gl: a graphical language for querying and restructuring xml documents. *Computer networks*, 31(11-16):1171–1187, 1999. 64

[10] James Clark, Steve DeRose, et al. Xml path language (xpath) version 1.0, 1999. 64

[11] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. Xml-ql: A query language for xml, 1998. 64

[12] DiffBot. Introducing the diffbot knowledge graph. `https://blog.diffbot.com/introducing-the-diffbot-knowledge-graph/`, 2018. [last accessed: 2020-02-07]. 63

[13] Science Direct. Query languages - an overview. `https://www.sciencedirect.com/topics/computer-science/query-languages`, 2016. [last accessed: 2020-02-18]. 8

[14] ElasticSearch. Kibana query language. `https://www.elastic.co/guide/en/kibana/current/kuery-query.html`. [last accessed: 2020-02-07]. 64

[15] ElasticSearch. Query dsl. `https://www.elastic.co/guide/en/elasticsearch/reference/7.5/query-dsl.html`. [last accessed: 2020-02-07]. 64

[16] Jonathan Robie et al. Xql. `https://www.w3.org/TandS/QL/QL98/pp/xql.html`. [last accessed: 2020-02-18]. 64

[17] Facebook. Graphql. `https://graphql.org/`. [last accessed: 2020-02-17]. 63

[18] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A comparison of rdf query languages. In *International Semantic Web Conference*, pages 502–517. Springer, 2004. 11, 64

[19] Florian Holzschuher and René Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204, 2013. 63

[20] Kevin Hutt. A comparison of rdf query languages. In *Proc. of 21th Computer Science Seminar, Hartfort, Connecticut*, pages 1–7, 2005. 11, 64

[21] Matthias Jarke and Yannis Vassiliou. A framework for choosing a database query language. In *Readings in Artificial Intelligence and Databases*, pages 363–375. Elsevier, 1989. 11

[22] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. Rql: a declarative query language for rdf. In *Proceedings of the 11th international conference on World Wide Web*, pages 592–603, 2002. 64

[23] Michael Kay. *XSLT: programmer's reference*. Wrox Press Ltd., 2001. 64

[24] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2(1):20, 2019. 7

[25] Microsoft. Learn the advanced hunting query language. `https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/advanced-hunting-query-language`, 2019. [last accessed: 2020-02-07]. 65

[26] Microsoft. Overview of kusto. `https://docs.microsoft.com/en-us/azure/kusto/query/`, 2019. [last accessed: 2020-02-07]. 65

[27] Libby Miller, Andy Seaborne, and Alberto Reggiori. Three implementations of squishql, a simple rdf query language. In *International Semantic Web Conference*, pages 423–435. Springer, 2002. 64

[28] MITRE. Mitre attack. `https://attack.mitre.org/`. [last accessed: 2020-02-12]. 65

[29] MITRE. Attck for industrial control systems. `https://collaborate.mitre.org/attackics/index.php/Main_Page`, 2020. [last accessed: 2020-06-03]. 41, 42, 44, 46

[30] MySQL. Mysql. `https://www.mysql.com/`. [last accessed: 2020-02-18]. 62

[31] Neo4j. Cypher query language. `https://neo4j.com/developer/cypher-query-language/`. [last accessed: 2020-02-07]. 27, 62

[32] Steven Noel, Eric Harley, Kam Him Tam, Michael Limiero, and Matthew Share. Cygraph: graph-based analytics and visualization for cybersecurity. In *Handbook of Statistics*, volume 35, pages 117–167. Elsevier, 2016. 2, 3, 63

[33] Nozomi. Nozomi networks guardian community edition. `https://community.nozominetworks.com/assets/Guardian-CommunityEdition-QuickStart.pdf`, 2019. [last accessed: 2020-02-07]. 65

[34] Oracle. Pl/sql. `https://www.oracle.com/nl/database/technologies/appdev/plsql.html`. [last accessed: 2020-02-18]. 62

[35] PostgreSQL. Postgresql: The world's most advanced open source relational database. `https://www.postgresql.org/`. [last accessed: 2020-02-18]. 62

[36] Andy Seaborne. Rdql. `https://www.w3.org/Submission/RDQL/`, 2004. [last accessed: 2020-02-18]. 64

[37] Splunk. The power of splunk search. `https://www.splunk.com/en_us/resources/search-processing-language.html`, 2018. [last accessed: 2020-02-07]. 64

[38] Apache Tinkerpop. The gremlin graph traversal machine and language. `http://tinkerpop.apache.org/gremlin.html`. [last accessed: 2020-02-17]. 63

[39] Karsten Tolle and Fabian Wleklinski. Trust and context using the rdf-source related storage system (rdf-s3) and easy rql (erql). In *Berliner XML Tage*, volume 11, page 13, 2004. 64

[40] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016. 63

[41] W3C. Sparql query language for rdf. `https://www.w3.org/TR/rdf-sparql-query/`, 2013. [last accessed: 2020-02-07]. 2, 64

[42] Wikipedia. Sql. `https://en.wikipedia.org/wiki/SQL`. [last accessed: 2020-02-07]. 62

[43] Ross Wolf. Introducing event query language. `https://www.elastic.co/blog/introducing-event-query-language`, 2019. [last accessed: 2020-02-07]. 65

Appendix A

Detailed Literature Overview

This appendix elaborates on the existing query languages in literature, surveyed for this research.

A.1 Query Languages for Relational Databases

This section talks about the most widely used query language for querying relational databases. These databases have data organized in tables, in the form of rows and colummns.

A.1.1 Structured Query Language (SQL)

SQL [42] is a query language used to query relational databases. Many implementations of SQL exist, including MySQL [30], PostgreSQL [35], PL/SQL [34] (by Oracle) and so on. These languages provide language constructs for basic filtering and joins, and some of these variants such as PostgreSQL provides full-text searching capabilities but they all lack constructs for constraint chaining. Furthermore, context information between tables require expensive joins and they are almost impossible to perform in relational databases using SQL.

A.2 Query Languages for NoSQL Databases

This section talks about query languages which are used to query data that is modeled in means other than the tabular relations in relational databases. The different languages listed below are classified into two sub-categories namely query languages for graph databases and the semantic web.

A.2.1 Query Languages for Graph Databases

Query languages for graph databases provide constructs for chaining constraints among nodes but there is a need for the security analyst to learn a new native query language, depending on the graph database. Some of the query languages used to query graph databases are described below.

Neo4j's Cypher Query Language

Neo4j invented Cypher [31], a declarative, SQL-inspired query language that is used to query graph databases. However, evaluating path queries is NP-hard due to the fact that Cypher allows path unwinding [2].

Oracle's PGQL

Oracle developed Property Graph Query Language (PGQL) [40] which is an intuitive SQL-like pattern-matching query language. It has support for regular path queries, path finding and graph construction.

MITRE's CyGraph Query Language (CyQL)

Noel et al. [32] from MITRE Corporation, developed a domain specific query language (CyQL) for their graph database called CyGraph, based on Cypher, relying on matching sub-graph patterns of interest, used for deriving attack paths. Although the underlying CyQL data model is a graph data model comprising entities and relationships, CyQL provides an extra layer of abstraction for CyGraph to support multiple backend data engine implementations, each with their own native query language. Although full-text search is supported, the major drawbacks of CyQL are that it is not open source and it is quite complex (which is precisely what we do not want to expose analysts to and that is what we aim to achieve with IIQL).

DiffBot Query Language (DQL)

Diffbot [12] developed a query language for its knowledge graphs that are created by encompassing the whole web, that is flexible to perform granular searches to find specific information, or to accumulate massive datasets for broader analysis. DQL is mostly used for marketing and sales, recruiting and business intelligence because it is used to gain information about people, companies, products and articles. Thus, it cannot be incorporated for our OT security data model.

Facebook's GraphQL

Facebook created GraphQL [17], a query language for APIs that are not specific to graph databases. Users get to define the structure of the data they want and they get exactly what they asked for. However, the output of a GraphQL query is a set of graphs rather than a result set with properties inside the vertices and edges.

Gremlin

Gremlin [38] is the query language for Apache TinkerPop. It is a graph traversal language that can be used for both OLTP and OLAP databases. It is Groovy-based, but allows developers to write Gremlin queries natively in modern programming languages such as Java, JavaScript and Python. Furthermore, Gremlin performs better than Cypher in the case of Friend of a Friend (FOAF) queries [19] and its expressivity and semantics is equivalent to SPARQL [2].

A.2.2 Query Languages for the Semantic Web

Numerous languages have been developed for data retrieval on the Semantic Web. The famous ones are described below, classified under two sub-categories according to the data formats they retrieve from: RDF and XML [4].

RDF Query Languages

1. **SPARQL:** SPARQL [41] is by far the most widely used query language used to retrieve and manipulate data stored in Resource Description Framework (RDF) format. It is mainly designed to query complex entity-relationship models (ontologies). However, there is no support for path expression and fuzzy (non-boolean) queries.

2. **RQL:** RQL [22] With RQL, access to data and schema can be combined in all manners and it is far more expressive than most other RDF query languages [4, 20].

Apart from the above, there are many other languages to query RDF data, mostly based on the above two languages. SquishQL [27] and RDQL [36] were designed for ease-of-use and limited functionality such as selection and extraction. TriQL [5] extends RDQL by constructs supporting querying of named graphs, allowing for better filtering. SeRQL [8] supports many of the basic RDF-query features such as path expressions, boolean constraints and optional matching [18]. eRQL [39] is a simplification of RQL based on a keyword-based interface supporting mostly one-word and neighbourhood queries, mostly used for search engines.

XML Query Languages

There are numerous languages to query XML data formats but most of them are only proposed theoretically as ideas in academia and not really adopted by the community/industry. The ones that are adopted are namely, XPath [10] (only used for selection and extraction), XSLT [23] (a loosely-typed, scripting language used for transforming/formatting XML data), and XQuery [6] (a strongly-typed query language which provides support for joins and recursions, as opposed to XPath). Some of the other theoretically proposed XML query languages are LOREL [1], XML-QL [11], XML-GL [9] and XQL [16].

A.3 Query Languages for Distributed Search and Analytics Engines

Distributed search and analytics engines like Splunk and ElasticSearch provide searching languages based on keywords and shell-like piping syntax. However, they lack support for exploring related concepts and looking for exploitable paths between assets. Furthermore, there is a need for the security analyst to learn the native query language.

A.3.1 Splunk's Search Processing Language (SPL)

Splunk [37] came up with its own searching language that can be used to analyze and visualize massive amounts of data. SPL supports full-text searching, basic filtering and advanced searching but it is not open source (it is very expensive) and querying over large amount of data might affect speed and performance.

A.3.2 ElasticSearch

ElasticSearch is an open source search engine that consists of primarily two query languages namely the Query Domain Specific Language (DSL) [15] and the Kibana Query Language (KQL) [14]. The former is based on JSON and supports a large number of queries such as compound, match-all, multi-match, full-text, range and geo queries but is less flexible and all data is indexed, which in turn causes an index overhead. The latter is the default query language in ElasticSearch, that is based on Lucene Query Language (LQL) [3]. It features autocomplete and an easy-to-use syntax but cannot search on nested objects.

A.4 Company Specific Query Languages

A.4.1 Microsoft's Advanced Hunting Query Language

Microsoft [25] developed a query language for advanced threat hunting in Microsoft Defender Security Centre that is based on the Kusto query language [26]. It works on their advanced threat hunting schema. This language can be used to retrieve either event information or information about machines and other entities. It supports basic filtering, advanced searching and querying abstracted operations. However, there is no support for full-text searching capabilities and there is a need to understand Microsoft's data schema to create queries spanning multiple tables.

A.4.2 Nozomi's Network Query Language (N2QL)

Nozomi [33] developed their own simple query language (N2QL) that is used to query their IoT security database. This language is inspired by the Linux terminal scripting languages with piping where the output of a command is the input of the next command. This achieves complex data processing with composition of many simple operations. It contains support for basic filtering, advanced searching across concepts and querying abstracted operations but it does not allow for full-text searching. Naturally, there is a need to understand Nozomis's IoT security data model, in order to effectively use queries.

A.4.3 Endgame's Event Query Language (EQL)

Endgame [43] developed an Event Query Language for threat hunting and real-time detection. It is schema-optional (i.e, EQL does not require the data schema to be specified upfront, but it contains features to map data to a schema for effective querying), it supports multiple database backends (i.e, it provides means to translate it to other native query languages, based on the database backend) and also supports conversion between multiple data formats. Additionally, it also features an EQL Analytics library (eqllib) that is a library of event-based analytics to detect adversary behaviors that map to MITRE's ATTACK framework [28]. It is open source and supports field lookups, comparisons, boolean logic, wildcard matching and function calls but it does not support full text keyword searches and finding attack paths/attack reachability, which are two of the use-case requirements of IIQL. (The latter, however, will almost never be supported in anything except CyQL.)