THE UNIVERSITY OF TULSA

THE GRADUATE SCHOOL

HOW TO PREPARE THE PERFECT THESIS OR DISSERTATION DOCUMENT

by
Noah L. Schrick

A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Science
in the Discipline of Computer Science

The Graduate School
The University of Tulsa

2022

THE UNIVERSITY OF TULSA

THE GRADUATE SCHOOL

HOW TO PREPARE THE PERFECT
THESIS OR DISSERTATION DOCUMENT

by
Noah L. Schrick

A THESIS

APPROVED FOR THE DISCIPLINE OF
COMPUTER SCIENCE

By Thesis Committee

I. M. Brilliant, Chair
Second Member
Third Member
Fourth Member
Fifth Member
Sixth Member

COPYRIGHT STATEMENT

ABSTRACT

Noah L. Schrick (Master of Science in Computer Science)

How to Prepare the Perfect Thesis or Dissertation Document

Directed by I. M. Brilliant

92 pp., Chapter 7: Conclusions and Future Works

(29 words)

In order to prepare a perfect thesis or dissertation, we do hereby follow these illustrious instructions to the letter.

ACKNOWLEDGEMENTS

I would like to thank everyone who has made this thesis template possible. Thanks and more thanks. In fact, let me give thanks all over the place.

TABLE OF CONTENTS

COPYRIGHT iii

ABSTRACT iv

ACKNOWLEDGEMENTS v

TABLE OF CONTENTS viii

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER 1: **INTRODUCTION** 1

1.1 **Introduction to Attack Graphs** 1

1.2 **Application to Compliance** 2

1.2.1 *Introduction to Compliance Graphs* 2

1.2.2 *Defining Compliance Graphs* 2

1.2.3 *Difficulties of Compliance Graphs and Introduction to Thesis Work* . 3

1.3 **Objectives and Contributions** 4

CHAPTER 2: **RELATED WORKS** 5

2.1 **Introduction to Graph Generation** 5

2.2 **Improvements to Attack Graph Generation** 5

2.3 **Attack Dependency Graphs** 5

2.4 **Compliance Graphs** 5

CHAPTER 3: **UTILITY EXTENSIONS TO THE RAGE ATTACK GRAPH GENERATOR** 6

3.1 **Path Walking** 6

3.2 **Compound Operators** 6

3.3 **Color Coding** 8

3.4 **Intermediate Database Storage** 9

3.4.1 *Memory Constraint Difficulties* 10

3.4.2 *Maximizing Performance with Intermediate Database Storage* 11

3.4.3 *Portability* 12

3.5 **Relational Operators** 13

CHAPTER 4: **SYNCHRONOUS FIRING** 14
4.1 **Introduction** 14
4.1.1 *Synchronous Firing in Literature* 14
4.2 **Necessary Components** 14
4.3 **Example Networks and Results** 14
4.3.1 *Example Networks* 14
4.3.2 *Results* 14

CHAPTER 5: **UTILIZATION OF MESSAGE PASSING INTERFACE** 15
5.1 **Introduction to MPI Utilization for Attack Graph Generation** 15
5.2 **Necessary Components** 15
5.2.1 *Serialization* 15
5.2.2 *Data Consistency* 16
5.3 **Tasking Approach** 16
5.3.1 *Introduction to the Tasking Approach* 16
5.3.2 *Algorithm Design* 16
Communication Structure 16
Task Zero 16
Task One 16
Task Two 16
Task Three 16
Task Four 19
Task Five 19
5.3.3 *Performance Expectations* 19
5.4 **Subgraphing Approach** 19
5.4.1 *Introduction to the Subgraphing Approach* 19
5.4.2 *Algorithm Design* 19
Communication Structure 19
Worker Nodes 19
Root Node 19
Database Node 19
5.4.3 *Performance Expectations* 19

CHAPTER 6: **PERFORMANCE ANALYSIS** 20
6.1 **Small Networks** 20
6.1.1 *Test Information* 20
6.1.2 *Results* 20
6.1.3 *Analysis* 20
6.2 **Large Networks** 20
6.2.1 *Test Information* 20
6.2.2 *Results* 20
6.2.3 *Analysis* 20
6.3 **Large Exploit Lists** 20
6.3.1 *Test Information* 20
6.3.2 *Results* 20
6.3.3 *Analysis* 20

6.4 **Distributed Hash Tables** 20
6.4.1 *Test Information* 21
6.4.2 *Results* 21
6.4.3 *Analysis* 21

CHAPTER 7: **CONCLUSIONS AND FUTURE WORKS** 22
7.1 **Future Work** 22

NOMENCLATURE 23

BIBLIOGRAPHY 23

APPENDIX A: **THE FIRST APPENDIX** 25
APPENDIX B: **THE SECOND APPENDIX** 26
B.1 **A Heading in an Appendix** 26
B.1.1 *A Subheading in an Appendix* 26
A Sub-subsection in an Appendix 26

LIST OF TABLES

LIST OF FIGURES

3.1 Path Walking to State 14 7

3.2 Color Coding a Small Network Based on Violations 9

5.1 Generation Flowchart of RAGE 17

5.2 Task Overview of the Attack Graph Generation Process 18

CHAPTER 1

INTRODUCTION

1.1 Introduction to Attack Graphs

Cybersecurity has been at the forefront of computing for decades, and vulnerability analysis modeling has been utilized to mitigate threats to aid in this effort. One such modeling approach is to represent a system or a set of systems through graphical means, and encode information into the nodes and edges of the graph. Even as early as the late 1990s, experts have composed various graphical models to map devices and vulnerabilities through attack trees, and this work can be seen through the works published by the authors of [11]. This work, and other attack tree discussions of this time such as that conducted by the author of [12], would later be referred to as early versions of modern-day attack graphs [10]. By utilizing this graphical approach, cybersecurity postures can be measued at a system's current status, as well as hypothesize and examine other postures based on system changes over time.

Attack Graphs are an appealing approach since they are often designed to be exhaustive: all system properties are represented at its intial state, all attack options are fully enumerated, all permutations are examined, and all changes to a system are encoded into their own independent states, where these states are then individually analyzed through the process. The authors of [13] also discuss the advantage of conciseness of attack graphs, where the final graph only incorporates states that an attacker can leverage; no superflous states are generated that can clutter analysis. Despite their advantages, attack graphs do suffer from their exhaustiveness. As the authors of [10] examine, even very small networks with only 10 hosts and 5 vulnerabilites yield graphs with 10 million edges. When scaling attack graphs to analyze the modern, interconnected state of large networks comprising of a multitude of hosts, and utilizing the entries located in the National Vulnerability Database and any custom vulnerability testing, this becomes infeasible. Similar difficulties arise in related fields, where social networks, bio-informatics, and neural network representations also result in graphs with millions of states [14]. Various efforts that will be discussed in Section 2.2 demonstrate methods and techniques that can mitigate these difficulties and improve performance.

1.2 Application to Compliance

1.2.1 Introduction to Compliance Graphs

As an alternative to attack graphs for examining vulnerable states and measuring cybersecurity postures, the focus can be narrowed to generate graphs with the purpose of examining compliance or regulation statuses. These graphs are known as compliance graphs. Compliance graphs can be especially useful for cyber-physical systems, where a greater need for compliance exists. As the authors of [7], [3], and [2] discuss, cyber-physical systems have seen greater usage, especially in areas like critical infrastructure and Internet of Things. The challenge of cyber-physical systems lies not only in the demand for cybersecurity of these systems, but also the concern for safe, stable, and undamaged equipment. The industry in which these devices are used can lead to additional compliance guidelines that must be followed. Compliance graphs are promising tools that can aid in minimizing the difficulties of these systems.

A few alterations are needed to attack graph generators to function as compliance graph generators, and these alterations are discussed in Section 1.2.2. Compliance requirements are broad and varying, and can function as safety regulations, maintenance compliance, or any other regulatory compliance. In the same fashion as attack graphs, compliance graphs are exhaustive, and future system states can be analyzed to determine appropriate steps that need to be taken for preventative measures [7].

1.2.2 Defining Compliance Graphs

The common features of attack graphs serve separate purposes in compliance graphs. The nodes of an attack graph typically represent the system state that includes the qualities and topologies of all assets in the network as they pertain to cybersecurity postures. Nodes of a compliance graphs also represent the system state, however they include the qualities and topologies of all assets in the network as they pertain to compliance regulation. For instance, a quality for a vehicle's maintenance compliance could be described as: *car:months_since_oil_change=6*, or *car:miles_since_oil_change=10,000*. Edges represent changes to a system state that inserted, modified, or deleted a quality or topology. Using the car example, an edge could represent the addition of more mileage or more time since the last oil change. One large differentiation of attack graphs and compliance graphs can be seen through topologies. For assets in attack graphs, topologies typically represent a connection of assets through a digital medium. For compliance graphs, topologies not only need to represent the digital connections of assets, but also need extensions to incorporate hardware devices such as sensors, actuators, or other equipment [7]. In addition, rather than using applicable exploits or vulnerabilities, compliance violation detections should be used. An attack graph generation engine would need to use compliance parameters rather than exploit files, but would otherwise function similarly in the generation process.

1.2.3 Difficulties of Compliance Graphs and Introduction to Thesis Work

Like attack graphs, compliance graphs suffer from the state space explosion problem. Since compliance graphs are also exhaustive, the resulting networks can grow to incredibly large sizes. Compliance regulations that need to be checked at each system state such as SOX, HIPAA, GDPR, PCI DSS, or any other regulatory compliance in conjunction with a large number of assets that need to be checked can very quickly produce these large resulting graphs. The creation of these graphs through a serial approach likewise becomes increasingly infeasible. Due to this, the high-performance computing space presents itself as an appealing approach. This work aims to extend the attack graph generator engine RAGE presented by the author in [5] to begin development for compliance graph generation. The example networks in this work will also be in the compliance graph space, specifically examining vehicle maintenance compliance. This work will also examine approaches to leverage high-performance computing to aid in the generation of compliance graphs.

1.3 Objectives and Contributions

The objectives of this thesis are:

- Extend the utility of RAGE to:
    1. Reduce the complexity required for network model and exploit file creation
    2. Expand the complexity of attack modeling
    3. Allow for the creation of an infinite sized Attack Graph, assuming infinite storage
    4. Split Attack Graphs into subgraphs to simplify analysis of individual clusters
- Implement solutions to reduce state space explosion while remaining exhaustive and capturing all necessary information
- Extend RAGE to function for heterogeneous distributed computing environments
- Utilize RAGE for compliance graph generation

CHAPTER 2

RELATED WORKS

Many authors and researchers have developed or extended attack graphs since their beginning as attack trees. This Chapter reviews a few of their efforts as they relate to this work.

2.1 Introduction to Graph Generation

2.2 Improvements to Attack Graph Generation

2.3 Attack Dependency Graphs

2.4 Compliance Graphs

CHAPTER 3

UTILITY EXTENSIONS TO THE RAGE ATTACK GRAPH GENERATOR

3.1 Path Walking

Due to the large-scale nature of Attack Graphs, analysis can prove difficult and time-consuming. With some networks reaching millions of states and edges, analyzing the entire network can be overwhelming complex. As a means of simplifying analysis, a potential strategy could be to consider only small subsets of the network at a time, rather than feeding the entire network into an analysis algorithm. To aid in this effort, a Path Walking feature was implemented as a separate program, and has two primary modes of usage. The goal of this feature is to provide a subset of the network that includes all possible paths from the root node to a designated node. The first mode is a manual mode, where a user can input the desired state to walk to, and the program will output a separate graph of all possible paths to the specified state. The second mode is an automatic mode, where the program will output seperate subgraphs to all states in the network that have qualities of "$compliance\_vio = true$" or "$compliance\_vios > 0$". This often produces multiple subgraphs, that can then be separately fed into an analysis program.

Figure 3.1 demonstates an output of the Path Walking feature when walking to state 14. In this figure, the primary observable feature is that the network was reduced from 16 states to 6 states, and 32 edges to 12 edges. The reduction from the original network to the subset varies on the overall connectivity of the original Attack Graph, but the reduction can aid in simplifying the analysis process if only certain states of the network are to be analyzed.

3.2 Compound Operators

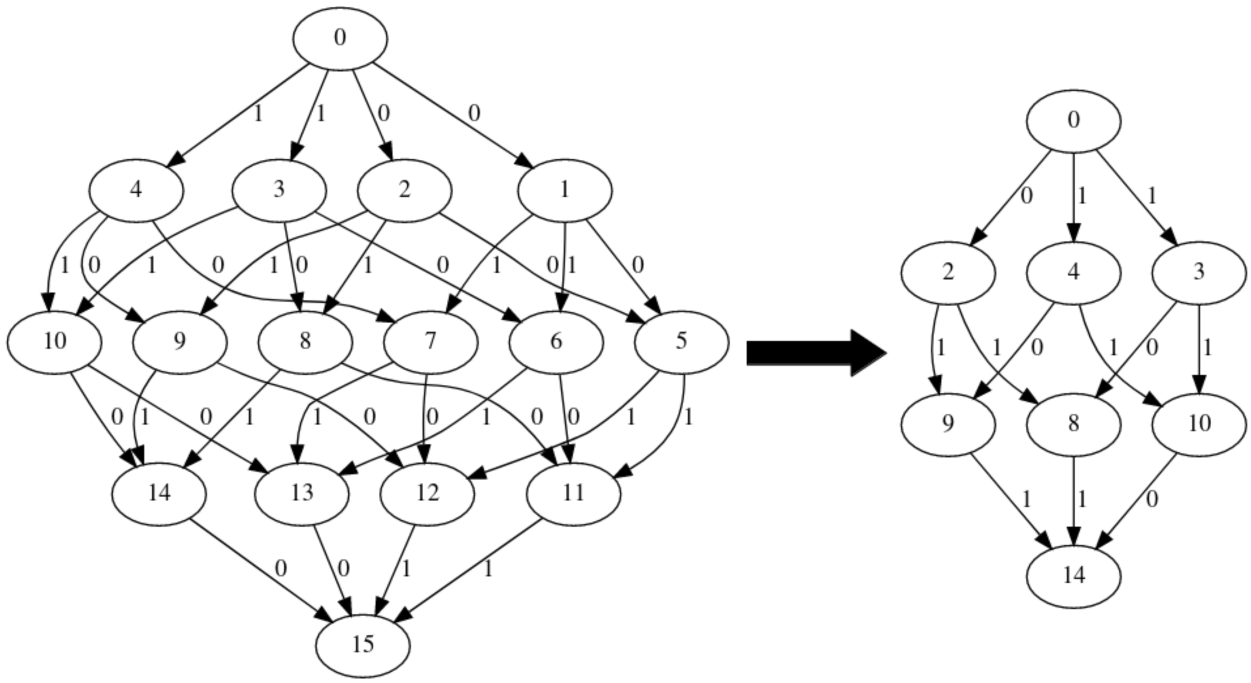

Figure 3.1: Path Walking to State 14

Many of the networks previously generated by RAGE compromise of states with features that can be fully enumerated. In many of the generated networks, there is an established set of qualities that will be used, with an established set of values. These typically have included "$compliance\_vio = true/false$", "$root = true/false$", or other general "$true/false$" values or "$version = X$" qualities. To expand on the types and complexities of networks that can be generated, compound operators have been added to RAGE. When updating a state, rather than setting a quality to a specific value, the previous value can now be modified by an amount specified through standard compound operators such as $+=$, $-=$, $*=$, or $/=$.

The work conducted by the author of [5] when designing the software architecture included specifications for a quality encoding scheme. As the author discusses, qualities have four fields, which include the asset ID, attributes, operator, and value. The operator field is 4 bits, which allows for a total of 16 operators. Since the only operator in use at the time was the " = " operator, the addition of four compound operators does not surpass the 16 operator limit, and no encoding scheme changes were necessary. This also allows for additional compound operators to be incorporated in the future.

A few changes were necessary to allow for the addition of compound operators. Before the generation of an Attack Graph begins, all values are stored in a hash table. For previous networks generated by RAGE, this was not a difficulty, since all values could be fully enumerated and all possible values were known. When using compound operators however, not all values can be fully known. The concept of approximating which exploits will be applicable and what absolute minimum or maximum values will be prior to generation is a difficult task, so not all values can be enumerated and stored into the hash table. As a result, on-the-fly updates to the hash table needed to be added to the generator. The original key-value scheme for hash tables relied on utilizing the size of the hash table for values. Since the order in which updates happen may not always remain consistent (and is especially true in distributed computing environments), it is possible for states to receive different hash values with the original hashing scheme. To prevent this, the hashing scheme was adjusted so that the new value of the compound operator is inserted into the hash table values if it was not found, rather than the size of the hash table. Previously, there was no safety check for the hash table, so if the value was not found, the program would end execution. The assumption that this value can be inserted into the hash table is safe to make, since compound operators are conducted on numeric values, and matches the numeric type of the hash table.

3.3 Color Coding

As a visual aid for analysis purposes, color coding was another feature implemented as a postprocessing tool for RAGE. When viewing the output graph of RAGE, all states are originally identical in appearance, apart from number of edges, edge IDs, and state IDs. To allow for visual differentiation, color coding can be enabled in the run script. Color coding currently functions by working through the graph output text file, but it can be extended to read directly from Postgres instead. The feature scans through the output file, and locates states that have "*compliance_vios* $= X$" (where X is a number greater than 0), or "*compliance_vio* $=$ *true*". For states that meet these properties, the color coding feature will add a color to the graphviz DOT file through the [*color* $=$ *COL*] attribute for the given node, where *COL* is assigned based on severity. For this version of color coding, severity is determined by the total number of compliance violations, but future versions can alter the severity measure through alternative means. Figure 3.2 displays an example graph that leverages color coding to easily identify problem states.

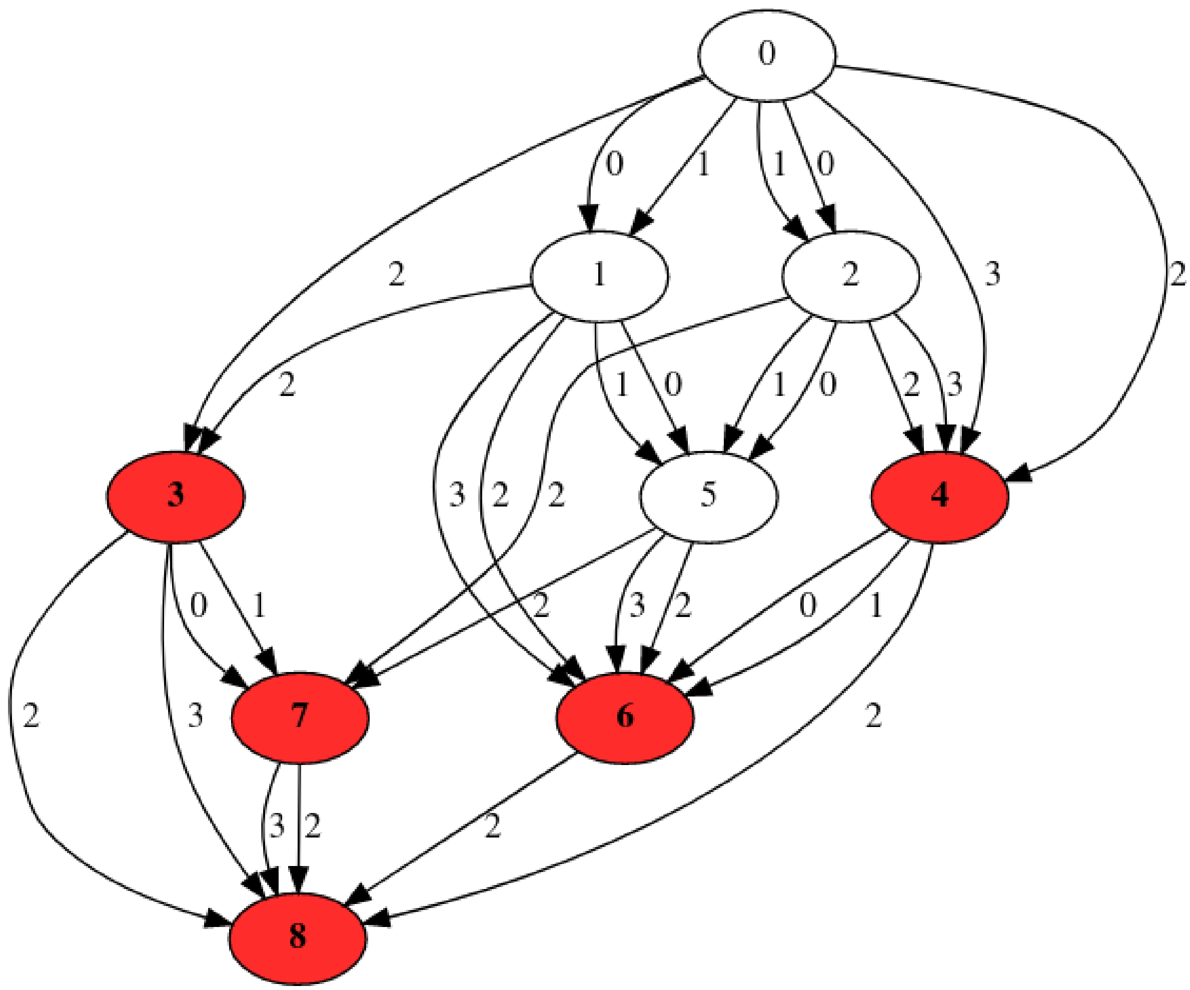

Figure 3.2: Color Coding a Small Network Based on Violations

3.4 Intermediate Database Storage

3.4.1 Memory Constraint Difficulties

Previous works with RAGE have been designed around maximizing performance to limit the longer runtimes caused by the state space explosion, such as the works seen by the authors of [5], [9], and [8]. To this end, the output graph is stored in memory during the generation process to minimize disk writing and reading, as well as leverage the performance benefits of memory operations since graph computation relies less on processor speed than that of data dependency complexity, parallelism coarseness, and memory access time [14], [1], [4]. The author of [5] does incorporate PostgreSQL as a final storage mechanism to write the resulting graph information, but no intermediate storage is otherwise conducted.

While the design decision to not use intermediate storage maximizes performance for graph generation, it does suffer from a few complications. When generating large networks, the system runs the risk of running out of memory. This typically does not occur when generation is conducted on small graphs, and is especially true when relatively small graphs are generated on a High Performance Computing system with substantial memory. However, when running on local systems, or when the graph is large, memory can quickly be depleted due to state space explosion. The memory depletion is due to two primary memory consumption points: the frontier which contains all of the states that still need to be explored, and the graph instance, which holds all of the network states and their state information as well as all of the edges.

The frontier quickly becomes a problem point with large networks that contain many layers before reaching leaf nodes. During the generation process, RAGE works on a Breadth-First Search approach, and new states are continuously discovered each time a state from the frontier is explored. In almost all cases, this means that for every state that is removed from the frontier, several more are added, leading to an ever-growing frontier that can not be adequately reduced for large networks. Simultaneously, the graph instance is ever-growing as states are explored. When the network contains numerous assets, each with their own large sets of qualities, the size of each state becomes noticeably larger. With some graphs containing millions of nodes and billions of edges, like those mentioned by the authors of

[14], it becomes increasingly unlikely that the graph can be fully contained within system memory.

3.4.2 Maximizing Performance with Intermediate Database Storage

Rather than a static implementation of storing to the database on disk at a set interval or a set size, the goal was to dynamically store to the database only when necessary. Since there is an associated cost with preparing the writes to disk, the communication cost across nodes, the writing to disk itself, and with retrieving items from disk, it is desirable to store as much in memory for as long as possible and only write when necessary. When running RAGE, a new argument can be passed *(-a <double>)* to specify the amount of memory the tool should use before writing to disk. This argument is a value between 0 and 0.99 to specify a percentage. This double is immediately reduced by 10%. For instance, if 0.6 is passed, it is immediately reduced to 0.5. This acts as a buffer for PostgreSQL. Since queries will consume a variable amount of memory through parsing or preparation, an additional 10% is saved as a precaution. This can be changed later as needed or desired for future optimizations. Specific to the graph data, the statement is made that the frontier is allowed to consume half of the allocated memory, and that the instance is allowed to consume the other half.

To decide when to store to the database instead of memory, two separate checks are made. The first check is for the frontier. If the size of the frontier consumes equal to or more than the allowed allocated memory, then all new states are stored into a new table in the database called "unexplored states". Each new state from this point forward is stored in the table, regardless of if room is freed in the frontier. This is to ensure proper ordering of the FIFO queue. The only time new states are stored directly into the frontier is when the unexplored states table is empty. Once the frontier has been completely emptied, new states are then pulled from the database into the frontier. To pull from the database, the parent loop for the generator process has been altered. Instead of a while loop for when the frontier is not empty, it has been adjusted to when the frontier is not empty or the unexplored states table is not empty. Due to C++ using short-circuit evaluation, some performance is gained since no SQL statement must be passed to disk to check the size of the unexplored states table unless the frontier is empty. The original design was to store new states into the frontier during the critical section to avoid testing on already-explored states. As a result, writing new states to the database is also performed during the critical section.

For the instance, a check in the critical section determines if the size of the instance consumes more than its allocated share of the memory. If it does, the edges, network states, and network state items are written to the database, and are then removed from memory.

However, a new issue arose with database storage. The original design was to save staging, preparation, and communication cost by writing all the data in one query (as in, writing all of the network states in one query, all the network state items in one query, and all the edges in one query). While this was best in terms of performance, it was also not feasible. Building the SQL queries themselves quickly began depleting the already constrained memory with large storage requests. As a result, the storage process would consume too much memory and crash the generator tool. To combat this, all queries had to be broken up into multiple queries. As previously mentioned, an extra 10% buffer was saved for the storage process. SQL query strings are now built until they consume the 10% buffer, where they are then processed by PostgreSQL, cleared, and the query building process resumes.

3.4.3 Portability

The intermediate database storage is greatly advantageous in increasing the portability of RAGE across various systems, while still allowing for performance benefits. By allowing for a user-defined argument, users can safely assign a value that allows for other processes and for the host OS to continue their workloads. While the "total memory" component currently utilizes the Linux *sysconf()* function, this is not rigid and is easily adjustable. When working on a High-Perfomance Computing cluster, using this function could lead to difficulties since multiple users may be working on the same nodes, which prevents RAGE from fully using all system memory. This could be prevented by using a job scheduler argument such as Slurm's "–exclusive" option, but this may not be desirable. Instead, a user could pass in the amount of total memory to use (and can be reused from a job scheduler's memory allocation request option), and the intermediate database storage process would function in the same fashion.

3.5 Relational Operators

As discussed in Section 3.2, many of the networks previously generated by RAGE compromise of states with an established set of qualities and values. These typically have included "$compliance\_vio = true/false$", "$root = true/false$", or other general "$true/false$" values or "$version = X$" qualities. To further expand the dynamism of attack graph generation, it is important to distinguish when a quality has a value that satisifies a relational comparison to an exploit. An example application can be seen through CVE-2019-10747, where "set-value is vulnerable to Prototype Pollution in versions lower than 3.0.1" [6]. Prior to the implementation of relational operators, to determine whether this exploit was applicable to a network state, multiple exploit qualities must be enumerated for all versions prior to 3.0.1. This would mean that the exploit needed to check if *version=3.0.0*, or *version=2.0.0*, or *version=1.0.0*, or *version=0.4.3*, etc. This becomes increasingly tedious when there are many versions, and not only reduces readability, but is also more prone to human error when creating the exploit files. As a result, relational operators were implemented.

To implement the relational operators, operator overloads were placed into the Quality class. At the time of writing, the following are implemented: $==$, $<$, $>$, $\leq$, $\geq$. However, these operators do not take up room in the encoding scheme, so additional operators can be freely implemented as needed. The overloads ensure that the Quality asset IDs and Quality names match, and then compares the Quality values based on the operator in question.

CHAPTER 4

SYNCHRONOUS FIRING

4.1 Introduction

4.1.1 Synchronous Firing in Literature

4.2 Necessary Components

4.3 Example Networks and Results

4.3.1 Example Networks

4.3.2 Results

CHAPTER 5

UTILIZATION OF MESSAGE PASSING INTERFACE

5.1 Introduction to MPI Utilization for Attack Graph Generation

5.2 Necessary Components

5.2.1 Serialization

In order to distribute workloads across nodes in a distributed system, various types of data will need to be sent and received. Support and mechanisms vary based on the MPI implementation, but most fundamental data types such as integers, doubles, characters, and Booleans are incorporated into the MPI implementation. While this does simplify some of the messages that need to be sent and received in the MPI approaches of attack graph generation, it does not cover the vast majority of them.

RAGE implements many custom classes and structs that are used throughout the generation process. Qualities, topologies, network states, and exploits are a few such examples. Rather than breaking each of these down into fundamental types manually, serialization functions are leveraged to handle most of this. RAGE already incorporates Boost graph libraries for auxiliary support, so this work extended this further to utilize the serialization libraries also provided by Boost. These libraries also include support for serializing all STL classes, and many of the RAGE classes have members that make use of the STL classes. One additional advantage of the Boost library approach is that many of the RAGE class members are nested. For example, the NetworkState class has a member vector of Quality classes. When serializing the NetworkState class, boost will recursively serialize all members, including the custom class members, assuming they also have serialization functions.

When using the serialization libraries, this work opted to use the intrusive route, where the class instances are altered directly. This was preferable to the non-intrusive approach, since the class instances were able to be altered with relative ease, and many of the class instances did not expose enough information for the non-intrusive approach to be viable.

5.2.2 Data Consistency

5.3 Tasking Approach

5.3.1 Introduction to the Tasking Approach

The high-level overview of the RAGE Data Flow Diagram was presented by the author of [5], and can be seen in Figure 5.1. This diagram includes an attack graph generation block that can be broken down into six main tasks. These tasks are described in Figure 5.2. Prior works such as that seen by the authors of [9] work to parallelize the attack graph generation using OpenMP by dividing the frontier. This approach, however, utilizes Message Passing Interface (MPI) to distribute the six tasks to examine the effect on speedup, efficiency, and scalability for attack graph generation.

5.3.2 Algorithm Design

Communication Structure:

Task Zero:

Task One:

Task Two:

Task Three:

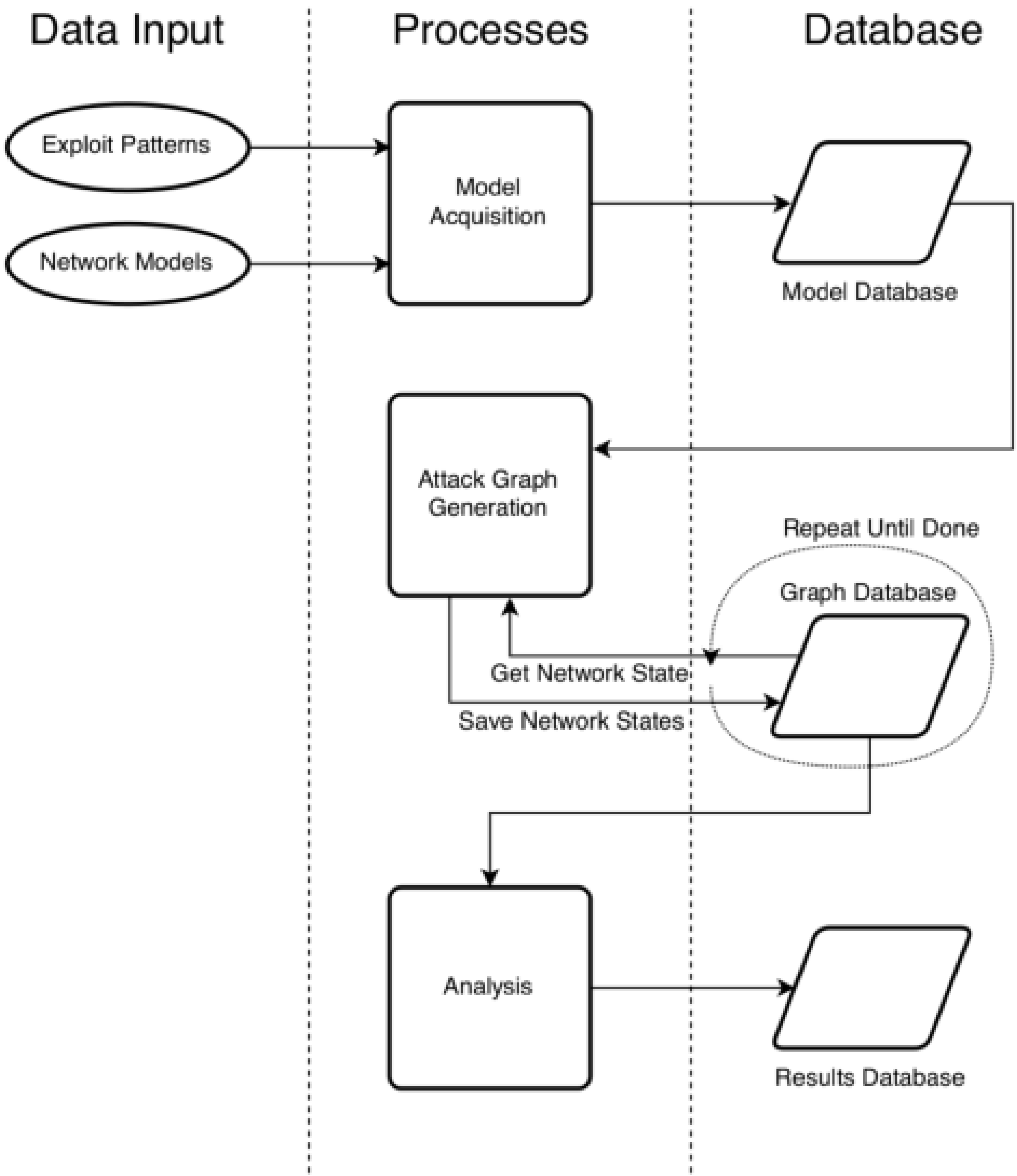

Figure 5.1: Generation Flowchart of RAGE

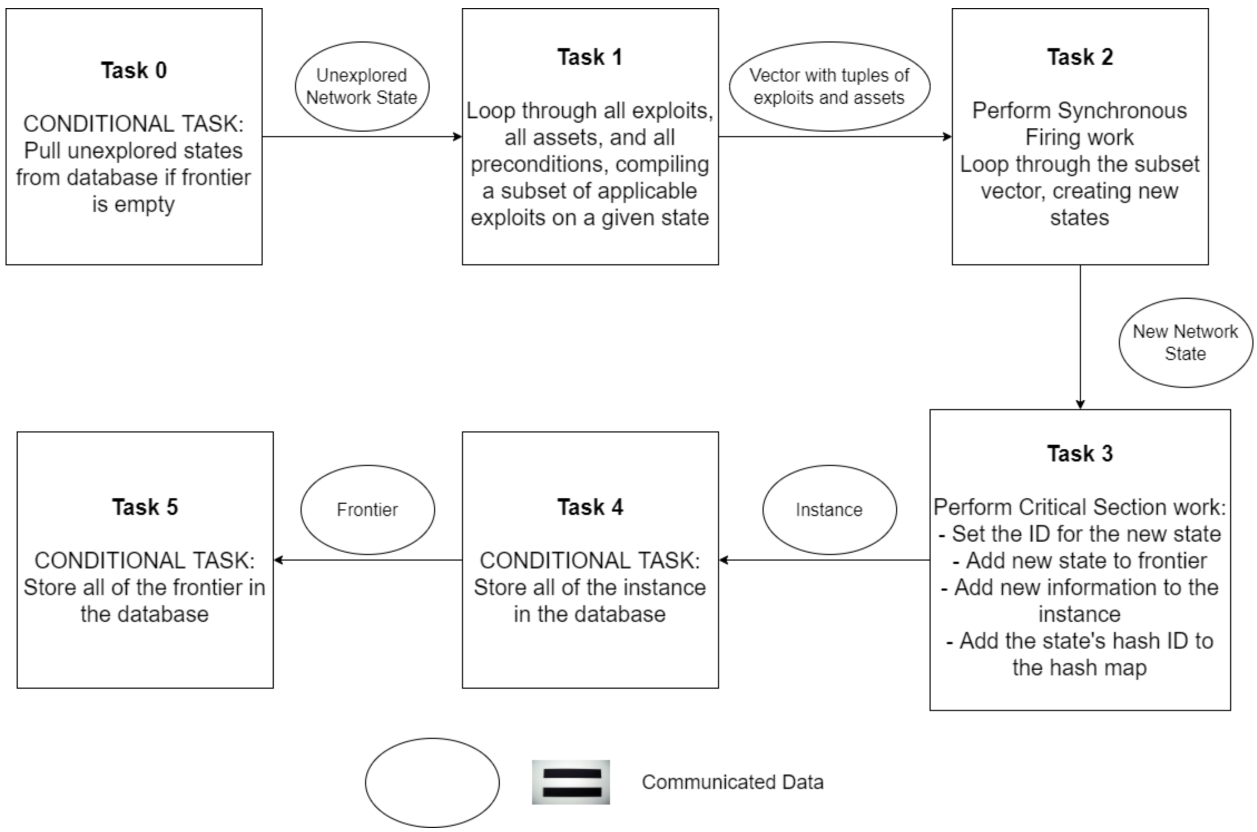

Figure 5.2: Task Overview of the Attack Graph Generation Process

Task Four:

Task Five:

5.3.3 Performance Expectations

5.4 Subgraphing Approach

5.4.1 Introduction to the Subgraphing Approach

5.4.2 Algorithm Design

Communication Structure:

Worker Nodes:

Root Node:

Database Node:

5.4.3 Performance Expectations

CHAPTER 6

PERFORMANCE ANALYSIS

6.1 Small Networks

6.1.1 Test Information

6.1.2 Results

6.1.3 Analysis

6.2 Large Networks

6.2.1 Test Information

6.2.2 Results

6.2.3 Analysis

6.3 Large Exploit Lists

6.3.1 Test Information

6.3.2 Results

6.3.3 Analysis

6.4 Distributed Hash Tables

6.4.1 Test Information

6.4.2 Results

6.4.3 Analysis

CHAPTER 7

CONCLUSIONS AND FUTURE WORKS

7.1 Future Work

BIBLIOGRAPHY

[1] Sam Ainsworth and Timothy M. Jones. Graph prefetching using data structure knowledge. *Proceedings of the International Conference on Supercomputing*, 01-03-June, 2016.

[2] Eric Allman. Complying with Compliance: Blowing it off is not an option. *ACM Queue*, 4(7), 2006.

[3] Ntsako Baloyi and Paula Kotzé. Guidelines for Data Privacy Compliance: A Focus on Cyberphysical Systems and Internet of Things. In *SAICSIT '19: Proceedings of the South African Institute of Computer Scientists and Information Technologists 2019*, Skukuza South Africa, 2019. Association for Computing Machinery.

[4] Jonathan Berry and Bruce Hendrickson. Graph Analysis with High Performance Computing. *Computing in Science and Engineering*, 2007.

[5] Kyle Cook. *RAGE: The Rage Attack Graph Engine*. PhD thesis, 2018.

[6] set-value is vulnerable to Prototype Pollution in versions lower than 3.0.1. The function mixin-deep could be tricked into adding or modifying properties of Object.prototype using any of the constructor, prototype and _proto_ payloads. National Vulnerability Database, August 2019.

[7] J. Hale, P. Hawrylak, and M. Papa. Compliance Method for a Cyber-Physical System.

[8] Ming Li, Peter Hawrylak, and John Hale. Combining OpenCL and MPI to support heterogeneous computing on a cluster. *ACM International Conference Proceeding Series*, 2019.

[9] Ming Li, Peter Hawrylak, and John Hale. Concurrency Strategies for Attack Graph Generation. *Proceedings - 2019 2nd International Conference on Data Intelligence and Security, ICDIS 2019*, pages 174–179, 2019.

[10] Xinming Ou, Wayne F Boyer, and Miles A Mcqueen. A Scalable Approach to Attack Graph Generation. pages 336–345, 2006.

[11] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. *Proceedings New Security Paradigms Workshop*, Part F1292:71–79, 1998.

[12] Bruce Schneier. Modeling Security Threats, 1999. Publication Title: Dr. Dobb's Journal.

[13] O. Sheyner, J. Haines, S. Jha, R.. Lippmann, and J. Wing. Automated Generation and Analysis of Attack Graphs. *Proceeding of 2002 IEEE Symposium on Security and Privacy*, pages 254–265, 2002.

[14] Jialiang Zhang, Soroosh Khoram, and Jing Li. Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search. *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 207–216, 2017.

APPENDIX A

THE FIRST APPENDIX

APPENDIX B

THE SECOND APPENDIX

B.1 A Heading in an Appendix

B.1.1 A Subheading in an Appendix

A Sub-subsection in an Appendix: