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

T H E   U N I V E R S I T Y   O F   T U L S A

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

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Introduction to Attack Graphs

1.2 Application to Cybersecurity and Compliance

1.3 Objectives and Contributions

CHAPTER 2

RELATED WORKS

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.

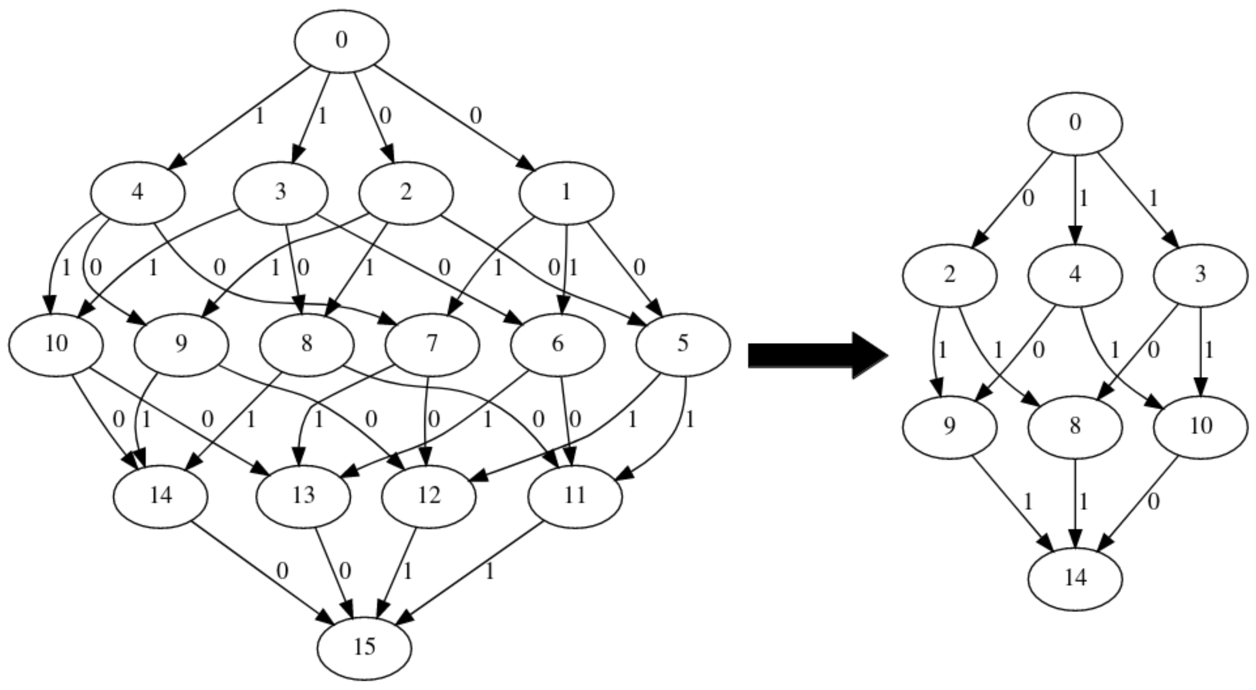

Figure 3.1: Path Walking to State 14

3.2 Compound Operators

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 [3] 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.

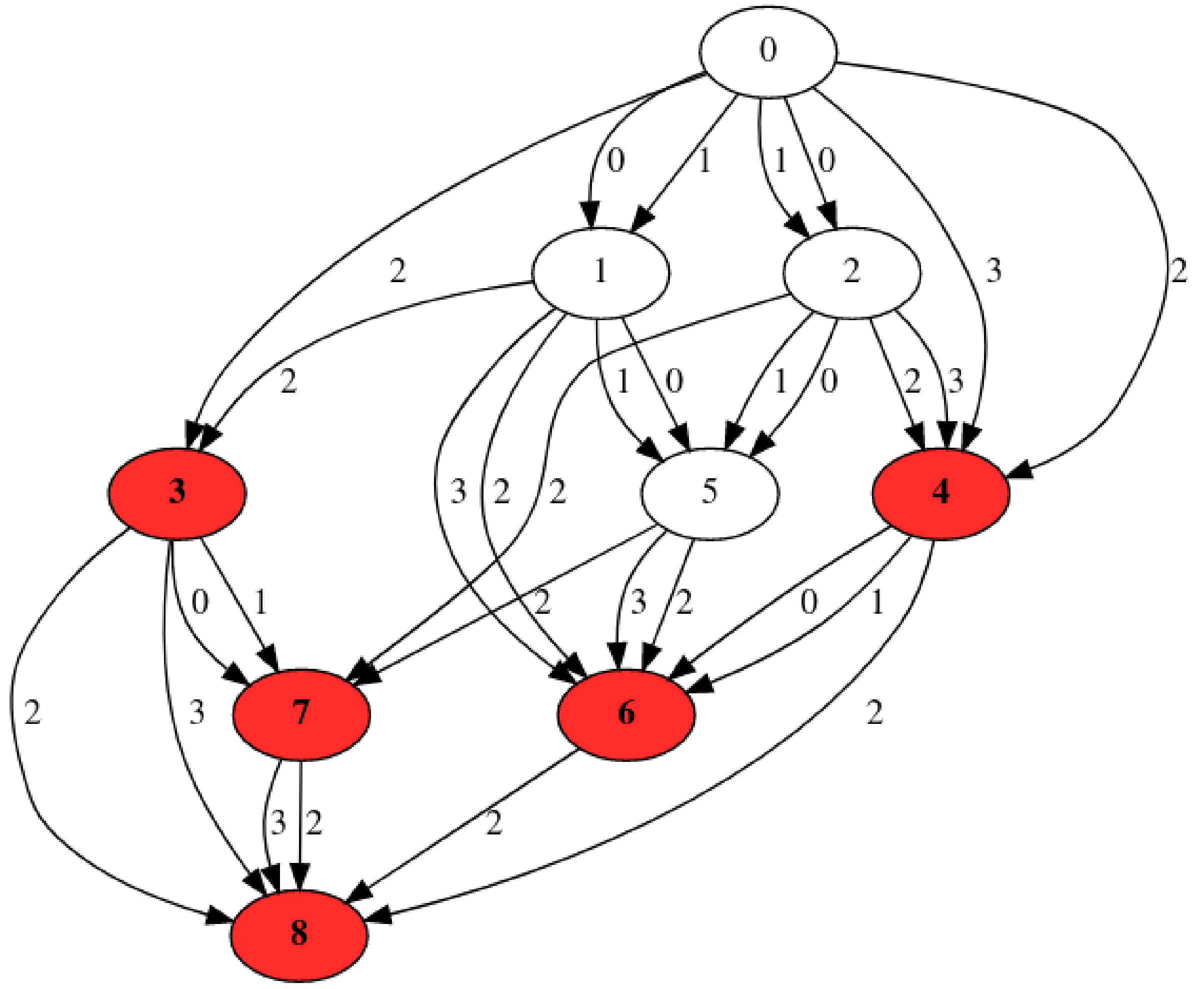

Figure 3.2: Color Coding a Small Network Based on Violations

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 [3], [5], and [4]. 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 [6], [1], [2]. The author of [3] 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 [6], 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, 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

Fifth section of the third chapter.

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

IMPLEMENTATION OF MESSAGE PASSING INTERFACE

5.1 Introduction to MPI Utilization for Attack Graph Generation

5.2 Necessary Components

5.2.1 Serialization

5.2.2 Data Consistency

5.3 Tasking Approach

5.3.1 Introduction to the Tasking Approach

5.3.2 Algorithm Design

Communication Structure:

Task Zero:

Task One:

Task Two:

Task Three:

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

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

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

[5] 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.

[6] 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: