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

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

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 [21]. This work, and other attack tree discussions of this time such as that conducted by the author of [22], would later be referred to as early versions of modern-day attack graphs [20]. By utilizing this graphical approach, cybersecurity postures can be measured 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 initial 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 [23] also discuss the advantage of conciseness of attack graphs, where the final graph only incorporates states that an attacker can leverage; no superfluous states are generated that can clutter analysis. Despite their advantages, attack graphs do suffer from their exhaustiveness. As the authors of [20] examine, even very small networks with only 10 hosts and 5 vulnerabilities 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 [26]. Various efforts that will be discussed in Section 2.3 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 [13], [7], and [4] 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 [13].

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 [13]. 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 [9] 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 for inseparable features while remaining exhaustive and capturing all necessary information
- Extend RAGE to function for heterogeneous distributed computing environments
- Extend and 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 and to graph generation.

2.1 Introduction to Graph Generation

Graph generation as a broad topic has many challenges that prevent full actualization of computation seen from a theoretical standpoint. In actuality, graph generation often achieves only a very low percentage of its expected performance [8]. A few reasons for this occurence lies in the underlying mechanisms of graph generation. The generation is predominantly memory based (as opposed to based on processor speed), where performance is tied to memory access time, the complexity of data dependency, and coarseness of parallelism [8], [26], [3]. The graph generation process is typically quite poor, resulting in lower performance results. Graphs consume large amounts of memory through their nodes and edges, graph data structures suffer from poor cache locality, and memory latency from the processor-memory gap all slow the generation process dramatically [8], [3]. Section 2.2 discusses a few works that can be used to improve the graph generation process, and Section 2.3 discusses a few works specific to attack graph generation improvements.

2.2 Graph Generation Improvements

For architectural and hardware techinques for generation improvement, the authors of [3] discuss the high cache miss rate, and how general prefetching leads does not increase the prediction rate due to non-sequenial graph structures and data-dependent access patterns. However, the authors continue to discuss that the generation algorithm is known in advance, so explicit tuning of the hardware prefetcher to follow the traversal order pattern can lead to better performance. The authors were able to achieve over 2x performance improvement of a breadth-first search approach with this method. Another hardware approach is to make use of accelerators. The authors of [24] present an approach for minimizing the slowdown caused by the underlying graph atomic functions. By using the atomic function patterns, the authors utilized pipeline stages where vertex updates can be processed in parallel dynamically. Other works, such as those by the authors of [26] and [12], leverage field-programmable gate arrays (FPGAs) for graph generation in the HPC space through various means. This includes reducing memory strain, storing repeatedly accessed lists, storing results, or other storage through the on-chip block RAM, or even levering Hybrid Memory Cubes for optimizing parallel access.

From a data structure standpoint, the authors of [5] describe the infeasibility of adjacency matrices in large-scale graphs, and this work and other works such as those by the authors of [25] and [18] discuss the appeal of distibuting a graph representation among systems. The author of [18] discuss the usage of distributed adjacency lists for assinging vertices to workers. The authors of [18] and [6] present other techniques for minimizing communication costs by achieving high compression ratios while maintaining a low compression cost. The Boost Graph Library and the Parallel Boost Graph Library both provide appealing features for working with graps, with the latter library notably having interoperability with MPI, Graphviz, and METIS [2], [1].

2.3 Improvements Specific to Attack Graph Generation

As a means of improving scalability of attack graphs, the authors of [20] present a new representation scheme. Traditional attack graphs encode the entire network at each state, but this representation uses logical statements to represent a portion of the network at each node. This is called a logical attack graph. This approach led to the reduction of the generation process to quadratic time and reduced the number of nodes in the resulting graph to $\mathcal{O}(n^2)$. However, this approach does require more analysis for identifying attack vectors.

Another approach presented by the authors of [10] represent a description of systems and their qualities and topologies as a state, with a queue of unexplored states. This work was continued by the authors of [16] by implementing a hash table among other features. Each of these works demonstrate an improvement in scalability through refining the desirable information.

Another approach for generation improvement is through parallelization. The authors of [16] leverage OpenMP to parallelize the exploration of a FIFO queue. This parallelization also includes the utilization of OpenMP's dynamic scheduling. In this approach, each thread receives a state to explore, where a critical section is employed to handle the atomic functions of merging new state information while avoiding collisions, race conditions, or stale data usage. The authors measured a 10x speedup over the serial algorithm. The authors of [17] present a parallel generation approach using CUDA, where speedup is obtained through a large number of CUDA cores. For a distributed approach, the authors of [14] present a technique for utilizing reachability hyper-graph partitioning and a virtual shared memory abstraction to prevent duplicate work by multiple nodes. This work had promising results in terms of limiting the state-space explosion and speedup as the number of network hosts increases.

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 separate 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 demonstrates 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

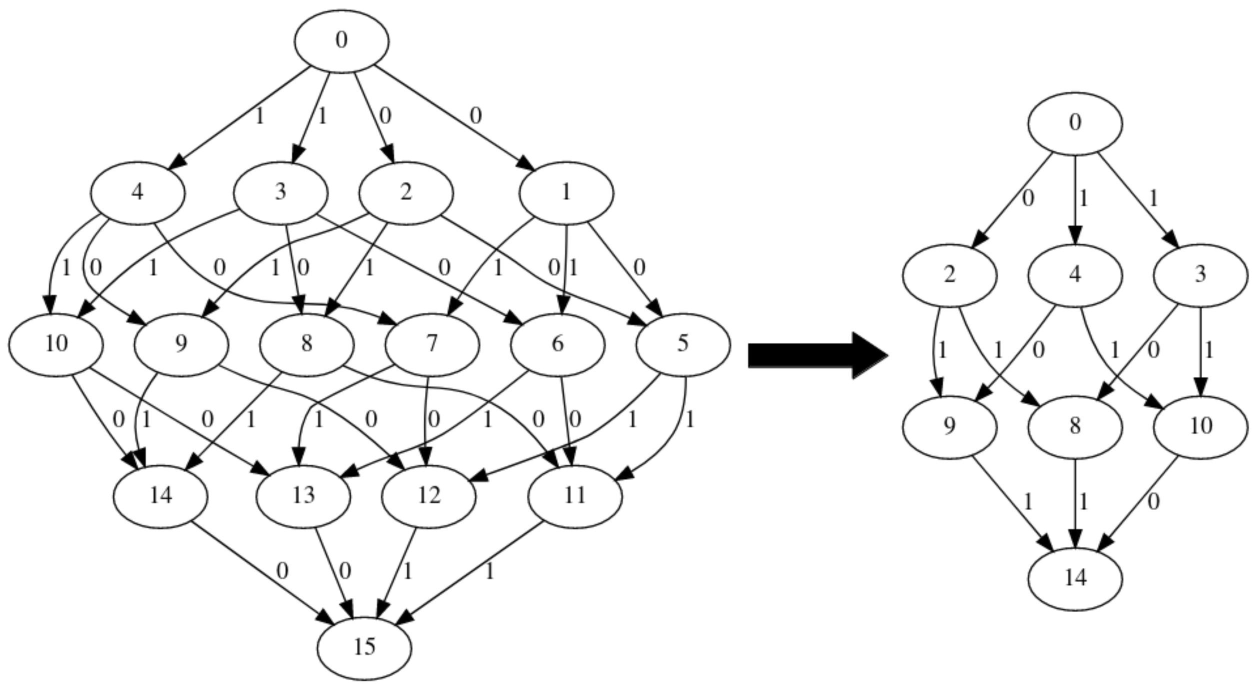


Figure 3.1: Path Walking to State 14

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 [9] 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.

Other changes involved updating classes (namely the Quality, EncodedQuality, ParameterizedQuality, NetworkState, and Keyvalue classes) to include a new member for the operator in question. Auxiliary functions related to this new member, such as prints and getters, were also added. In addition, preconditions were altered to include operator overloads to check the asset identifier, quality name, and quality values for the update process.

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.

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 runtime caused by the state space explosion, such as the works seen by the authors of [9], [16], and [15]. 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 [26], [3], [8]. The author of [9] 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

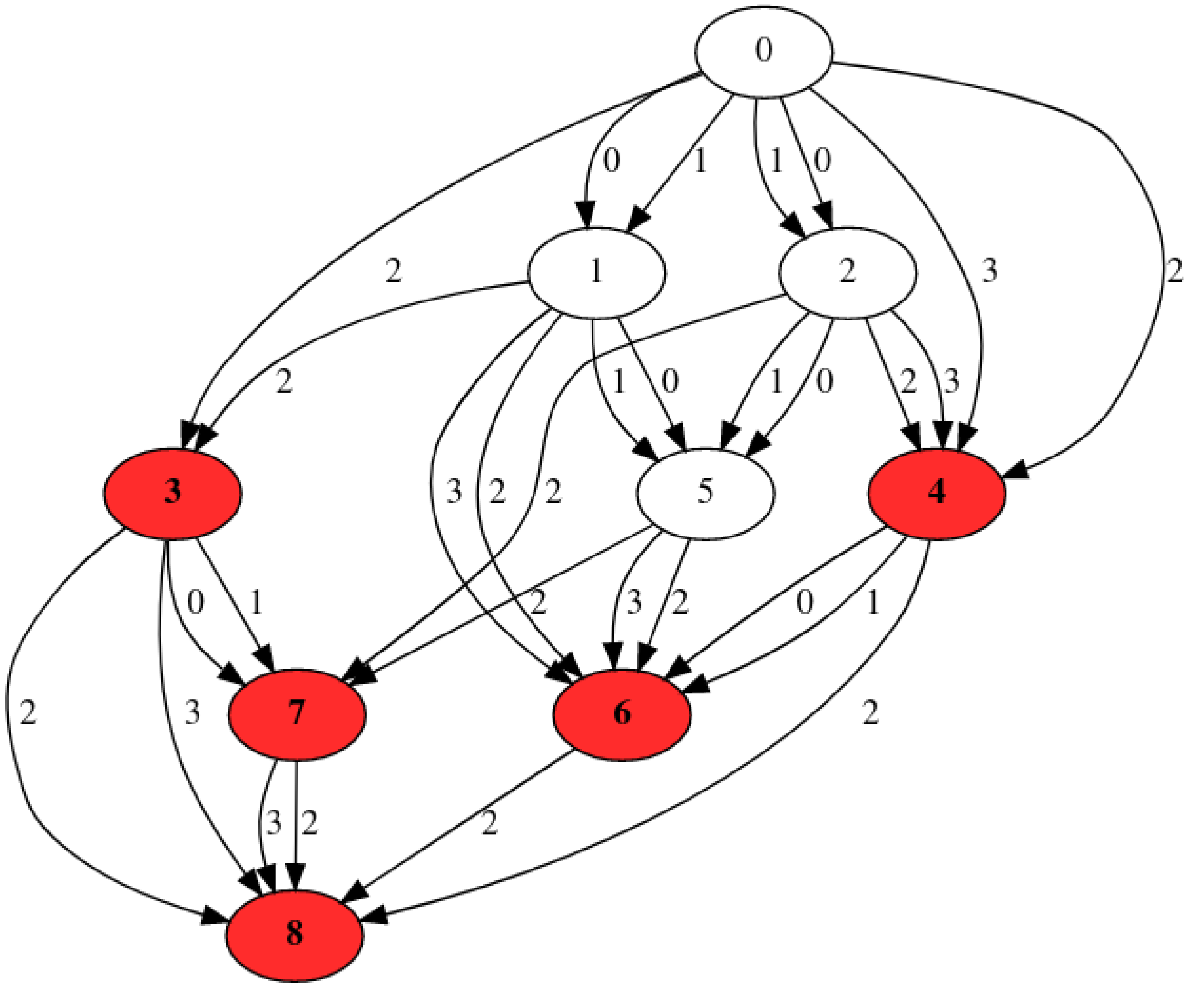


Figure 3.2: Color Coding a Small Network Based on Violations

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 [26], 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-Performance 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 satisfies 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" [11]. 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.

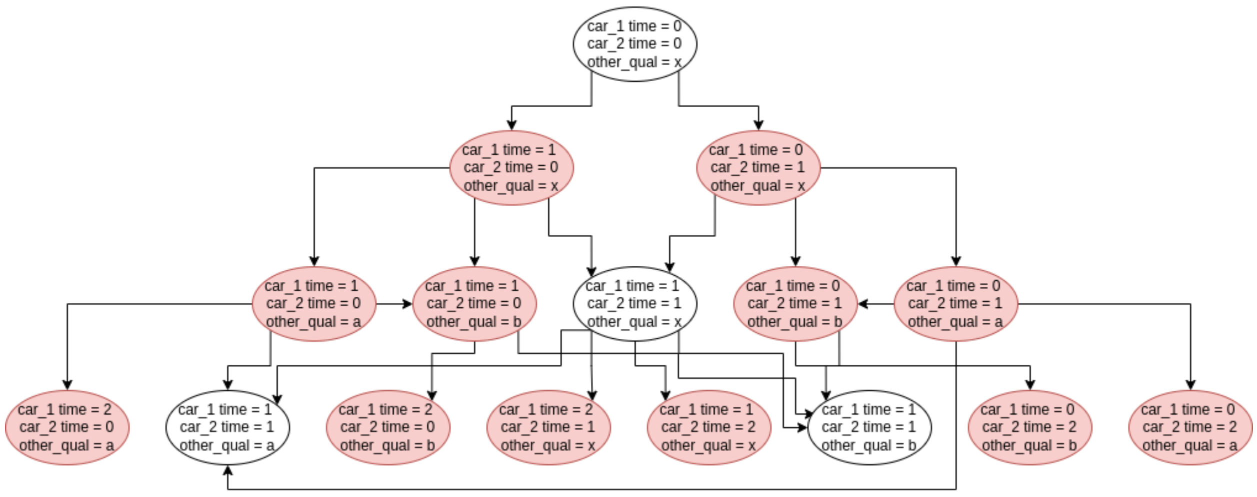
CHAPTER 4

SYNCHRONOUS FIRING

4.1 Introduction

One main appeal of attack graphs and compliance graphs are their exhaustiveness. The ability to generate all permutations of attack chains or to generate all possible ways a system can fall out of compliance is a valuable feature. The disadvantage of this approach is that the generation of the final graph increases in time, as does the analysis. One other disadvantage is that this exhaustiveness can produce states that are not actually feasible. When a system has assets that have inseparable features, the generation process forcibly separates features to examine all permutations, since the generation process only modifies one quality at a time. One example of an inseparable feature is time. If two different assets are identical and no constraints dictate otherwise, the two assets cannot proceed through times at different rates.

For example, if two cars were manufactured at the same moment, one of these cars cannot proceed multiple time steps into the future while the other remains at its current time step - each car must step through time at the same rate. However, the generation of attack graphs and compliance graphs examines the possibilities that one car ages by one time step, while the other car does not, or vice versa. This results in an attack graph that can be seen in Figure 4.1, which is a partial attack graph showing the separation of the time feature. All states shaded as a light red color are considered infeasible, since all of these states comprise of assets that have advanced time at different rates. It is noticeable that not only are the infeasible states a wasteful generation, but that they also lead to the generation of even more infeasible states that will then also be explored. The desired goal of a generation process similar to this is to have a single state transition that updates assets with an inseparable feature simultaneously.



car_1 time = 0 car_2 time = 0 other_qual = x
car_1 time = 1 car_2 time = 0 other_qual = x
car_1 time = 0 car_2 time = 1 other_qual = x
car_1 time = 1 car_2 time = 0 other_qual = a
car_1 time = 1 car_2 time = 0 other_qual = b
car_1 time = 1 car_2 time = 1 other_qual = x
car_1 time = 0 car_2 time = 1 other_qual = b
car_1 time = 0 car_2 time = 1 other_qual = a
car_1 time = 2 car_2 time = 0 other_qual = a
car_1 time = 1 car_2 time = 1 other_qual = a
car_1 time = 2 car_2 time = 0 other_qual = b
car_1 time = 2 car_2 time = 1 other_qual = x
car_1 time = 1 car_2 time = 2 other_qual = x
car_1 time = 1 car_2 time = 1 other_qual = b
car_1 time = 0 car_2 time = 2 other_qual = b
car_1 time = 0 car_2 time = 2 other_qual = a

Figure 4.1: A network without Synchronous Firing generating infeasible states

Post-processing is one option at removing the infeasible states. This process would simplify and reduce the time taken for the analysis process, but the generation process would still suffer from generating infeasible states, as well as exploring the infeasible states. Instead, a new approach called Synchronous Firing can be used to prevent the generation of infeasible states. The goal of the Synchronous Firing feature is to prevent the generation of infeasible states, while also not incurring a greater computational cost. This Chapter will discuss the development of this feature, its mentionings in literature, and examine the results when using this feature in applicable networks.

4.1.1 Synchronous Firing in Literature

Synchronous Firing is discussed by the author of [19], where it is described as grouped exploits. The functionality discussed by the author is similar: where an exploit should be fired on all possible assets simultaneously. This is also described as synchronizing multiple exploits. The methodology is similar to the one implemented in this work, but there are notable differences. The first, is that the work performed by the author of [19] utilizes global features with group features. Using the simultaneous exploit firing necessitated a separation of global and group features, and synchronous firing could not be performed on exploits that could be applicable to both sets. A second difference is that there is no consistency checking in the work by the author of [19], which could lead to indeterminate behavior or race conditions, unless additional effort was put into encoding exploits to use precondition guards. A third difference is that the work of [19] could still lead to a separation of features. The behavior of the work would attempt to fire all exploits on all applicable assets simultaneously, but if some assets were not ready or capable to fire, they would not proceed with the exploit firing but the other assets would. The last difference is that the work by the author of [19] was developed in Python, since that was the language of the generator of the tool at the time. The work by the author of [9] led to new development of RAGE in C++ for performance enhancements, so the synchronous firing feature in this new work was likewise developed in C++.

4.2 Necessary Alterations

For the implementation of the Synchronous Fire Feature, there are four primary changes and/or additions necessary. The first is a change in the lexical analyzer, the second involves multiple changes to PostgreSQL, the third is the implementation of compound operators (as discussed in Section 3.2), and lastly is a change in the graph generation process. This section will compromise of subsections discussing the development of these four alterations.

4.2.1 GNU Bison and Flex

The work conducted by the author of [9] included the introduction of GNU Bison and GNU Flex into RAGE. The introduction of Bison and Flex allows for an easily modifiable grammar to adjust features, the ability to easily update parsers since Bison and Flex are built into the build system, and increases portability since Flex and Bison generate standard C. For the development of the synchronous fire, a similar approach was taken to that of the work performed by the author of [19] with the exploit keywords. However, rather than having both global and group keywords, this work only incorporates the group keyword to prevent a few of the difficulties discussed in Section 4.1.1. The new “group” keyword is intended to be used when creating the exploit files. The design of exploits in the exploit file is developed as:

```
 <exploit>  ::= <group name> "group" "exploit" <identifier> , (<parameter-list>)=
```

where the “<group name>” identifier and “group” keyword is optional. An example of an exploit not utilizing the group feature is:

```
exploit brake_pads(2015_Toyota_Corolla_LE)=
```

and an example of an exploit utilizing the group feature is:

```
time group exploit advance_month(all_applicable)=
```

To implement this feature, a few changes were conducted, where the intention is to detect the usage of the “group” keyword, and have the lexical analyzer code return to the parser implementation file to alert of the presence of the ”GROUP” token. The new token is of type string with the name GROUP, and it is comprised of a leading “IDENTIFIER” of type string or integer token, followed by the GROUP token. This new token also required changes to the processing of the “exploit” keyword. If the group keyword is not detected, the exploit has a group of name “null”. If the group keyword is detected, then the leading IDENTIFIER is parsed, and the exploit is assigned to a group with the parsed name. Various auxiliary functions were also adjusted to include (for instance) support for printing the groups of each exploit. Figure 4.2 illustrates the incorporation of this feature into Bison, Flex, and the overall program.

4.2.2 PostgreSQL

As seen in Figure 4.2, Bison and Flex feed into the Model Database. With the addition of a new group identifier and the group keyword, minor alterations were needed to ensure compatibility with the PostgreSQL database. One adjustment was to alter the exploit

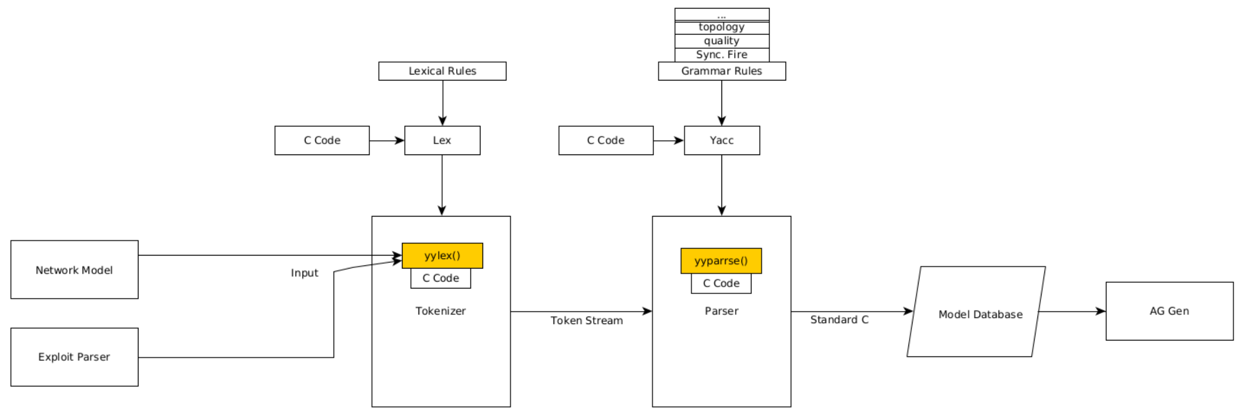


Figure 4.2: Inclusion of Synchronous Firing into GNU Bison, GNU Flex, and the overall program

table in the SQL schema to include new columns of type "TEXT". The second adjustment was to update the SQL builder functions. This included updating the related functions such as exploit creations, exploit parsing, database fetching, and SQL string builders to add additional room for the group identifier.

4.2.3 Compound Operators

While not strictly necessary, compound operators greatly simplify the exploit file creation process.For example, implementing time as a feature into the tool without compound operators would increase its difficulty substantially. For each time interval, a separate exploit would need to be created, with time flags to indicate the current time. If time was increased monthly for a year, 12 different exploits would need to be created, with flags to ensure that time jumps did not occur. Updating qualities without compound operators also relied on flags, and clever, but convoluted methods for incrementing values. Instead, compound operators were implemented, and this addition was discussed in Section 3.2.

4.2.4 Graph Generation

The implementation of synchronous firing in the graph generation process relies on a map to hold the fired status of groups. Previously, each iteration of the applicable exploit vector loop generated a new state. With synchronous firing, all assets should be updating the same state, rather than each independently creating a new state. To implement this, each iteration of the applicable exploit vector checks if the element is in a group and if that group has fired. If the element is in a group, the group has not been fired, and all group members are ready to fire, then all group members will loop through an update process to alter the single converged state. Otherwise, the loop will either continue to the next iteration if group conditions are not met, or will create a single state if it is not in a group. Figure 4.3 displays the synchronous fire approach.

4.3 Example Networks and Results

All data was collected on a 13 node cluster, with 12 nodes serving as dedicated compute nodes, and 1 node serving as the login node. Each compute node has a configuration as follows:

- OS: CentOS release 6.9
- CPU: Two Intel Xeon E5-2620 v3
- Two Intel Xeon Phi Co-Processors
- One FPGA (Nallatech PCIE-385n A7 Altera Stratix V)
- Memory: 64318MiB

All nodes are connected with an Infiniband interconnect.

4.3.1 Example Networks

The example networks for testing the effectiveness of synchronous firing follow the compliance graph generation approach. These networks analyze two assets, both of which are identical 2006 Toyota Corolla cars with identical qualities. The generation examines both cars at their current states, and proceeds to advance in time by a pre-determined amount, up to a pre-determined limit. Each time increment updates each car by an identical amount

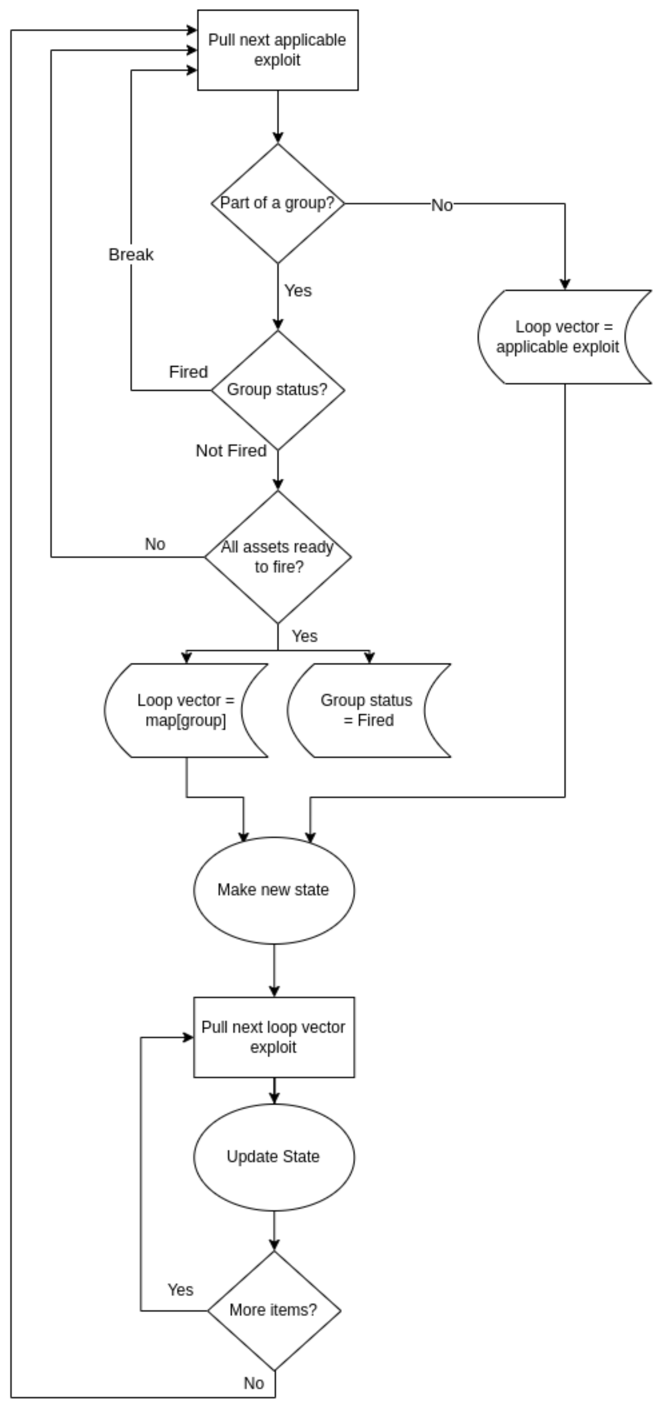


Figure 4.3: Synchronous Firing in the Graph Generation Process

of mileage. During the generation process, it is determined if a car is out of compliance either through mileage or time since last maintenance in accordance with the Toyota Corolla Maintenance Schedule manual.

In addition, the tests employ the use of "services", where if a car is out of compliance, it will go through a correction process and reset the mileage and time since last service. Each test varies in the number of services used. The 1 Service test only employs one service, and it is dedicated to brake pads. The 2 Service test employs two services, where the first service is dedicated to the brake pads, and the second is for exhaust pipes. This process extends to the 3 and 4 Service tests. The testing information is as follows:

- All tests ran for 12 months, with time steps of 1 month.
- All tests had the same number of compliance checks: brake pads, exhaust pipes, vacuum pumps, and AC filters.
- There were 10 base exploits, and an additional 4 exploits were individually added in the form of services for each test.
- All tests used the same network model.
- All tests used the same exploit file, with the exception of the "group" keyword being present in the synchronous firing testing.
- Services must be performed prior to advancing time, if services are applicable.
- Graph visualization was not timed. Only the generation and database operation time was measured.

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 compliance graph generation process 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 [16], [17], and [14] work to parallelize the graph generation using OpenMP, CUDA, and hyper-graph partitioning. This approach, however, utilizes Message Passing Interface (MPI) to distribute the six identified tasks of RAGE to examine the effect on speedup, efficiency, and scalability for attack and compliance graph generation.

5.3.2 Algorithm Design

The design of the tasking approach is to leverage a pipeline structure with the six tasks and MPI nodes. Each stage of the pipeline will pass the necessary data to the next stage through various MPI messages, where the next stage's nodes will receive the data and execute their tasks. The pipeline is considered fully saturated when each task has a dedicated node. When there are less nodes than tasks, some nodes will processing multiple tasks. When there are more nodes than tasks, additional nodes will be assigned to Tasks 1 and 2. Timings were collected in the serial approach for various networks that displayed more time requirements for Tasks 1 and 2, with larger network sizes requiring vastly more time to be taken in Tasks 1 and 2. As a result, additional nodes are assigned to Tasks 1 and 2. Node allocation can be seen in Figure **??**.

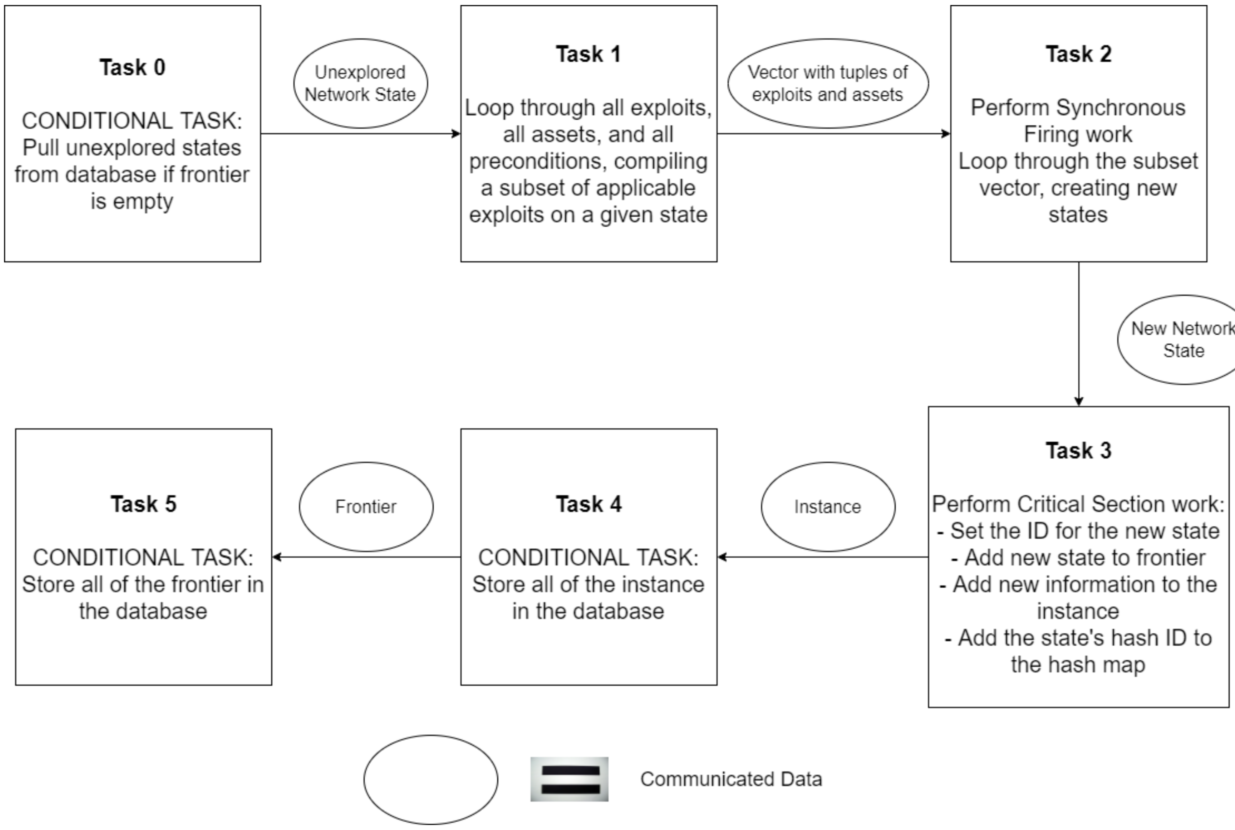

Task 0
CONDITIONAL TASK: Pull unexplored states from database if frontier is empty
Unexplored Network State
Task 1
Loop through all exploits, all assets, and all preconditions, compiling a subset of applicable exploits on a given state
Vector with tuples of exploits and assets
Task 2
Perform Synchronous Firing work
Loop through the subset vector, creating new states
New Network State
Task 3
Perform Critical Section work:
- Set the ID for the new state
- Add new state to frontier
- Add new information to the instance
- Add the state's hash ID to the hash map
Instance
Task 4
CONDITIONAL TASK: Store all of the instance in the database
Frontier
Task 5
CONDITIONAL TASK: Store all of the frontier in the database
Communicated Data

Figure 5.1: Task Overview of the Attack Graph Generation Process

For determining which tasks should be handled by the root note, a few considerations were made. Minimizing communication cost and avoiding unnecessary complexity were the main two considerations. In the serial approach, the frontier queue was the primary data structure for the majority of the execution. Rather than using a distributed queue or passing multiple sub-queues between nodes, the minimal option is to pass states individually. This approach also assists in reducing the complexity. Managing multiple frontier queues would require duplication checks, multiple nodes requesting data from and storing data into the database, and devising a strategy to maintain proper queue ordering, all of which would also increase the communication cost. As a result, the root node will be dedicated to Tasks 0 and 3.

Communication Structure:

Task Zero: Task Zero is performed by the root node, and is a conditional task; it is not guaranteed to be executed at each pipeline iteration. Task Zero is only executed when the frontier is empty, but the database still holds unexplored states. This occurs when there are memory constraints, and database storage is performed during execution to offload the demand, as discussed in Section **??**. After the completion of Task 0, the frontier has a state popped, and the root node sends the state to n_1.

Task One:

Task Two:

Task Three:

Task Four:

Task Five:

5.3.3 Performance Expectations

./Chapter5_img/node-alloc.png

Figure 5.2: Node Allocation for each Task

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] The Boost Graph Library - 1.75.0.

[2] An Overview of the Parallel Boost Graph Library - 1.75.0.

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

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

[5] Shaikh Arifuzzaman and Maleq Khan. Fast parallel conversion of edge list to adjacency list for large-scale graphs. In *HPC '15: Proceedings of the Symposium on High Performance Computing*, pages 17–24, April 2015.

[6] Janani Balaji and Rajshekhar Sunderraman. Graph Topology Abstraction for Distributed Path Queries. In *HPGP '16: Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 27–34, May 2016.

[7] 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.

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

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

[10] Kyle Cook, Thomas Shaw, John Hale, and Peter Hawrylak. Scalable attack graph generation. *Proceedings of the 11th Annual Cyber and Information Security Research Conference, CISRC 2016*, 2016.

[11] 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.

[12] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. FPGP: Graph processing framework on FPGA: A case study of breadth-first search. *FPGA 2016 - Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 105–110, 2016.

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

[14] Kerem Kaynar and Fikret Sivrikaya. Distributed attack graph generation. *IEEE Transactions on Dependable and Secure Computing*, 13(5):519–532, 2016.

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

[16] 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.

[17] Ming Li, Peter J. Hawrylak, and John Hale. Implementing an attack graph generator in cuda. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 730–738, 2020.

[18] Panagiotis Liakos, Katia Papakonstantinopoulou, and Alex Delis. Memory-Optimized Distributed Graph Processing through Novel Compression Techniques. In *CIKM '16:*

Proceedings of the 25th ACM International Conference on Information and Knowledge Management, pages 2317–2322, October 2016.

[19] G Louthan. *Hybrid Attack Graphs for Modeling Cyber-Physical Systems.* PhD thesis, 2011.

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

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

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

[23] 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.

[24] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. An efficient graph accelerator with parallel data conflict management. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, 2018.

[25] Xinjie Yu, Wentao Chen, Jiajia Miao, Jian Chen, Handong Mao, Qiong Luo, and Lin Gu. The Construction of Large Graph Data Structures in a Scalable Distributed Message System. In *HPCCT 2018: Proceedings of the 2018 2nd High Performance Computing and Cluster Technologies Conference*, pages 6–10, June 2018.

[26] 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: