State Space Explosion Mitigation for Large-Scale Attack and Compliance Graphs Using Synchronous Exploit Firing

Noah L. Schrick
Tandy School of Computer Science
The University of Tulsa
Tulsa, USA
noah-schrick@utulsa.edu

Peter J. Hawrylak
Tandy School of Computer Science
The University of Tulsa
Tulsa, USA
peter-hawrylak@utulsa.edu

***Abstract*—Attack and compliance graphs are useful tools for cybersecurity and regulatory or compliance analysis. These graphs represent the state of a system or a set of systems, and can be used to identify all current or future ways the systems are compromised or at risk of violating regulatory or compliance mandates. However, due to their exhaustiveness and thorough permutation checking, these graphs suffer from state space explosion - the graphs rapidly increase in the total number of states, and likewise, their generation time also rapidly increases. This state space explosion in turn also slows the analysis process. This work introduces a mitigation technique called synchronous firing, where graph users and designers can prevent the generation of infeasible states by firing exploits simultaneously through joining inseparable features like time. This feature does not invalidate the integrity of the resulting attack or compliance graph by altering the exhaustiveness or permutation checking of the generation process, but rather jointly fires exploits through their defined inseparable features.**

***Index Terms*—Attack Graph; Compliance Graph; Synchronous Firing; High-Performance Computing; Cybersecurity; Compliance and Regulation; Speedup;**

I. Introduction

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 [1]. This work, and other attack tree discussions of this time such as that conducted by the author of [2], would later be referred to as early versions of modern-day attack graphs [3]. 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.

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 [4], [5], and [6] discuss, cyber-physical systems have seen greater usage, especially in areas such as 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, increasing the complexity required for examining compliance statuses. Compliance graphs are promising tools that can aid in minimizing the overhead caused by these systems and the regulations they must follow.

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 [7] 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 well. As the authors of [3] 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, attack graph generation quickly becomes infeasible. Similar difficulties arise in related fields, where social networks, bioinformatics, and neural network representations result in graphs with millions of states [8]. This state space explosion is a natural by-product of the graph generation process, and removing or avoiding it entirely undermines the overall goal of attack and compliance graphs. However, there

are some scenarios in which the state space explosion can be mitigated when certain features are inseparable. This work discusses the application of synchronous exploit firing which mitigates state space explosion for applicable scenarios, and discusses the results of its use.

II. Related Work

Multiple works have introduced various approaches for mitigating state space explosion. The authors of [9] propose that attack graphs encapsulate excessive information that lead to difficulties in scalability. They discuss the concept of monotonicity, where attackers do not need to backtrack. If a previous exploit was achieved, its preconditions and postconditions should not be revoked through another, future exploit firing. The authors of [10] use monotonicity in their tool, TVA, along with various node and edge representations based on sets and dependency graphs that can likewise mitigate the state space explosion challenge. The authors of [3] also take the approach of using alternate representations of the underlying graph structure through logical attack graphs. In this representation, each node only encompasses a portion of the network in a logical statement format, as opposed to encoding the entire system information at each node. This approach is able to limit the total number of nodes to $O(N^2)$, with N representing the total number of nodes in the system.

A form of synchronous firing is discussed by the author of [11], where it is described as grouped exploits. The functionality discussed by the author is similar: firing an exploit should be performed on all possible assets simultaneously. This was 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 [11] utilizes global features with group features. Using the simultaneous exploit firing necessitated a separation of global and group features, and grouped exploits 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 [11], 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 [11] could still lead to a separation of features. The grouped exploit feature would attempt to fire all exploits on all applicable assets simultaneously, but if some assets were not ready or capable to fire, these assets would not proceed with the exploit firing but the applicable assets would. The last difference is that the work by the author of [11] was developed in Python, since that was the language of the generator of the tool at the time. This work relies on RAGE (The RAGE Attack Graph Engine) for the feature development and result collection [12]. RAGE is developed in C++ for performance enhancements, so the synchronous firing feature in this new work was likewise developed in C++.

III. Inseparable Features

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 attainable or realistic, as briefly mentioned in Section II. 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 should not, and realistically cannot, proceed through time 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 1, which is a partial attack graph showing the separation of the time feature. All shaded states are considered unattainable, since all of these states comprise of assets that have advanced time at different rates. It is noticeable that not only are the unattainable states themselves a wasteful generation, but they also lead to the generation of even more unattainable states that will then also be explored. A better procedure for a generation process similar to this example is to have a single state transition that updates assets with an inseparable feature simultaneously.

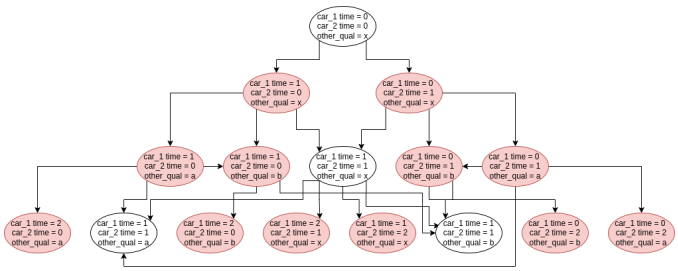

Fig. 1. A network without Synchronous Firing generating infeasible states

Post-processing is one option at removing the unattainable states. This process would simplify and reduce the time taken for the analysis process, but the generation process would still suffer from generating and exploring the unattainable states, and would still need to go through a post-processing step. Instead, a new feature called synchronous firing can be used to prevent the generation of these states. The goal of the synchronous firing feature is to prevent the generation

of unattainable states, while also not incurring a greater computational cost. Section IV will discuss the development of this feature, and Section V will examine the results when using this feature in applicable networks.

IV. Implementing Synchronous Firing

A. Base Generator Description

For the implementation of the synchronous firing feature, there were four primary changes and additions that were 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, and lastly is a change in the graph generation process. This Section will consist of subsections discussing the development of these four alterations.

B. GNU Bison and Flex

The work conducted by the author of [12] 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 firing feature, a similar approach was taken to that of the work performed by the author of [11] 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 II. 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 the keyword recognition and group name parsing, a few changes were made, where the intention was 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 2 illustrates the incorporation of this feature into Bison, Flex, and the overall program.

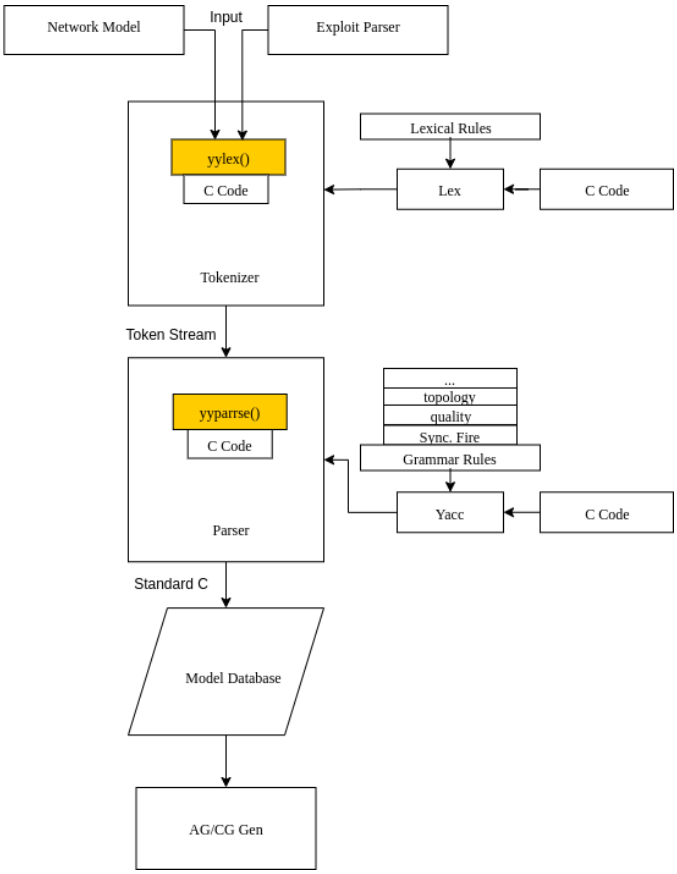

Fig. 2. Inclusion of Synchronous Firing into GNU Bison, GNU Flex, and the overall program

C. PostgreSQL

As seen in Figure 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 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.

D. Compound Operators

Many of the graphs previously generated by RAGE comprise of states with features that can be fully enumerated. In many of these generated graphs, there was an established set of qualities that was 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 graphs that can be generated and to allow for synchronous firing, 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 $/=$. Previous work on an attack graph generator included the implementation of compound operators, as seen by the author of [13]. However, this work was conducted on the previous iteration of an attack graph generator written in Python. This attack graph generator has since been rewritten in C++ by the author of [12], and compound operators have not been included in the latest version of RAGE.

The work conducted by the author of [12] when designing the software architecture of RAGE 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 concern 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 task of approximating which exploits will be applicable and what absolute minimum or maximum value bounds will be prior to generation is difficult, and not all values can be enumerated and stored into the hash table. As a result, real-time 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 assertion that the new 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.

E. 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 current loop 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 3 displays the synchronous fire approach.

V. Results

A. Experimental Networks and Computing Platform

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 a 10Gbps Infiniband interconnect.

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 of mileage. During the generation process, it is determined if a car is out of compliance either through mileage or time since its 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

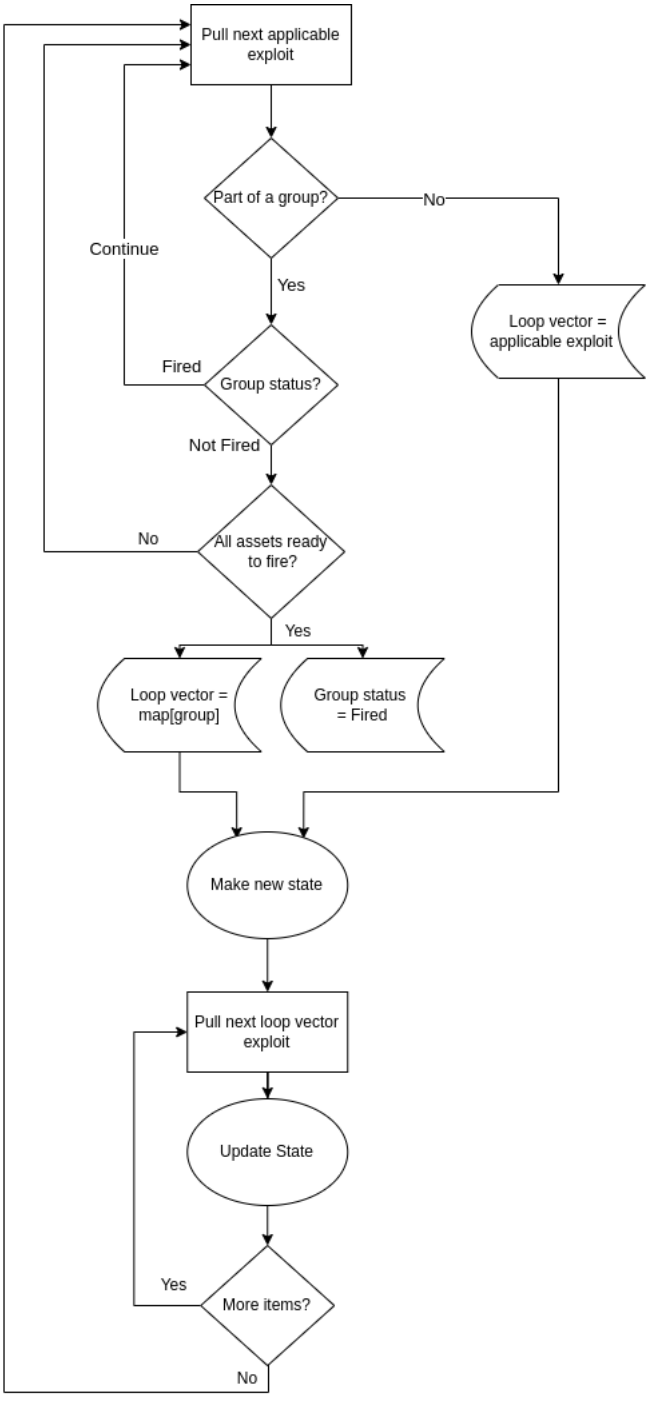

Fig. 3. Synchronous Firing in the Graph Generation Process

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, 4, 5, and 6 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, AC filters, oil changes, and driveshaft boots.
- There were 12 base exploits, and an additional 6 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.

The compliance checks are as follows:

- Brake pads: to be checked every 6 months
- Exhaust pipes: to be checked every 12 months
- AC filter: to be checked every 12,000 miles
- Vacuum pump: to be checked every 120,000 miles
- Engine oil: to be checked every 6,000 miles
- Driveshaft boots: to be checked every 12,000 miles

B. Results and Analysis

1) Results for the Theoretical Environment: Using the testing setup described in Section V-A on the platform described at the beginning of Section V-A, results were collected in regards to the effect of synchronous firing on both state space and runtime. There was also a collection of the graphs' edge to state ratio (E/S Ratio). These results can be seen in Figures 4 and 5. The respective tables are seen in Tables I and II. Both figures show a decrease in the number of states and a decrease in the runtime when synchronous firing is utilized. Since synchronous firing prevents the generation of unattainable states, there is no meaningful information loss that occurs in the graphs generated with the synchronous firing feature. Since the resulting number of states was also reduced, there will be increased justification for the synchronous firing approach due to a reduced runtime for the analysis process. Figure 6 displays the speedup (according to Amdahl's Law) obtained when using synchronous firing instead of non-synchronous firing for identical setups.

When examining the E/S ratio for the non-synchronous graphs, it is both expected and observed that the ratio slightly increases as the number of services increases. When more applicable exploits are used during the generation process, the number of permutations increases, which corresponds with the growing number of states and edges. However, the increase in the number of services also increases the relation between states and the new permutations.

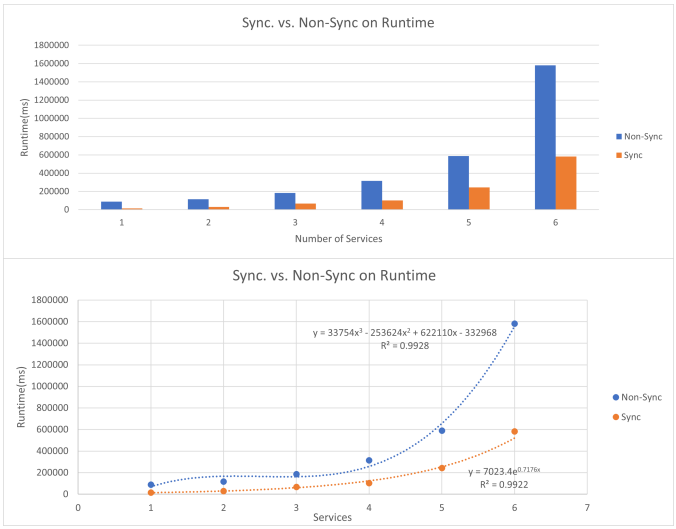

Fig. 4. Bar Graph and Line Graph Representations of Synchronous Firing on Runtime

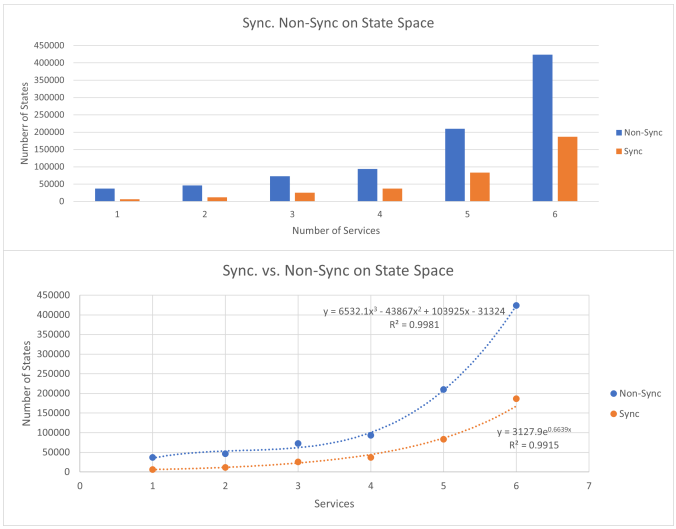

Fig. 5. Bar Graph and Line Graph Representations of Synchronous Firing on State Space

When comparing the E/S ratio for the non-synchronous graphs to the E/S ratio for the synchronous graphs, it is observed that the ratio does not remain constant. For example, for the 5 service case, the non-synchronous graph has an E/S ratio of 6.398, and the synchronous graph has an E/S ratio of 7.209. While the number of states and the number of edges is reduced when using synchronous firing, the ratio of edges to states is not necessarily constant or reduced.

2) Results for a Grouped Environment: The environment and resulting graphs presented in Section **??** depict the possible states of the two cars in compliance graph formats. While these graphs demonstrated accurate, exhaustive depictions of the cars and their compliance standings, they may not be realistic representations of the most likely outcomes. If a car was due for two compliance checks at the same time, it

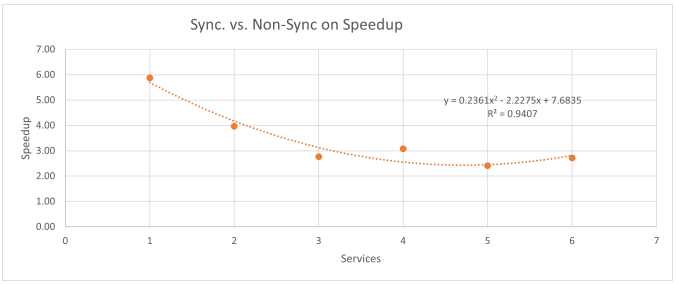

Fig. 6. Speedup (Amdahl's) Obtained When Using Synchronous Firing

Non-Synchronous Firing			
Number of Services	**Number of States**	**Runtime (ms)**	**E/S Ratio**
1	37001	87366.65	5.484
2	46361	115929.97	5.595
3	72489	184634.34	5.590
4	93525	314470.16	5.841
5	209944	588336.01	5.977
6	423940	1581697.61	6.398

TABLE I
TABLED RESULTS FOR THE NON-SYNCHRONOUS FIRING TESTING

is unlikely that the car would be taken for one maintenance, returned to its original destination, then driven immediately back for maintenance, and finally to its original destination once more. The more realistic scenario is that the car is taken for maintenance, both services are performed at the same visit, and then the car is returned to its original destination.

Another set of graphs were generated using only the 3 service case. These services were for a driveshaft boot check, an AC filter change, and an oil change. This set of graphs used 'super services", where a car would undergo multiple services simultaneously. These results are seen in Table III for the synchronous firing enabled generation, and Table IV for the non-synchronous firing generation.

VI. FUTURE WORKS

As seen and discussed in Section III, when unattainable states are generated, there is a compounding effect. Each unattainable state is explored, and is likely to generate additional unattainable states. Future works include examining the effect of synchronous firing when more assets are utilized. It is hypothesized that the synchronous firing approach will lead to an increased runtime reduction and state space reduction due to the increased number of unattainable state permutations. This work had a limited number of assets, but generated upwards of 400,000 states due to repeated applications of the exploit set due to the services corresponding with the compliance graph. Future work could alter the test scenario to have a greater number of assets, and a standard set of exploits more akin to an attack graph. Other future works could include measuring the performance of synchronous firing when multiple groups of inseparable

Synchronous Firing				
Number of Services	**Number of States**	**Runtime (ms)**	**E/S Ratio**	**Speedup (Amdahl's)**
1	6277	14872.86	5.501	5.87
2	11653	29251.56	5.954	3.96
3	25317	66799.18	6.321	2.76
4	36949	102216.30	6.538	3.08
5	83134	243612.05	6.868	2.42
6	186679	581840.76	7.209	2.72

TABLE II
TABLED RESULTS FOR THE SYNCHRONOUS FIRING TESTING

Super Services with Synchronous Firing			
Permutation	**Number of States**	**Runtime (ms)**	**E/S Ratio**
All Disjoint			
Any Two Services One Disjoint			
All Three Services			

TABLE III
TABLED RESULTS FOR THE SUPER SERVICES WITH SYNCHRONOUS FIRING

Super Services with Non-Synchronous Firing			
Permutation	**Number of States**	**Runtime (ms)**	**E/S Ratio**
All Disjoint			
Any Two Services One Disjoint			
All Three Services			

TABLE IV
TABLED RESULTS FOR THE SUPER SERVICES WITHOUT SYNCHRONOUS FIRING

features are used. This work used a single group, but multiple groups be added to examine the performance of the feature.

Another avenue for future works would be to take a network science approach. There may be features of interest from examining the topology of the resulting graphs with and without synchronous firing. Various centrality metrics could be examined, as well as examining transformations such as dominant trees or transitive closures derived from the original graphs. Each approach could compare each graph when using or not using synchronous firing to determine if there are possible points of interest. Taking a network science approach could also examine and analyze the E/S ratio differences between the graphs when using or not using synchronous firing, and attempt to provide further insight on what those differences mean in terms of usability of the graphs.

VII. CONCLUSION

This work implemented a state space explosion mitigation technique called synchronous firing. This feature is able to fire exploits simultaneously among a group of assets through a single state transition. By firing exploits across multiple assets, it is able to prevent the separation of features that should normally be inseparable (such as time), and successfully reduces the number of total states in the resulting attack or compliance graph. This feature does not alter the procedure of the generation process in a way that undermines the integrity of the resulting attack or compliance graph, and only groups assets through defined inseparable features. This feature is also toggleable, and the generation process seen in Figure 3 does not change if the feature is disabled. This feature successfully reduced the total number of states, reduced the runtime of the generation process, and can lead to a reduced analysis process due to a smaller resulting graph.

REFERENCES

[1] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," *Proceedings New Security Paradigms Workshop*, vol. Part F1292, pp. 71–79, 1998. doi: 10.1145/310889.310919.

[2] B. Schneier, "Modeling Security Threats," *Dr. Dobb's Journal*, 1999. vol. 24, no.12.

[3] X. Ou, W. F. Boyer, and M. A. Mcqueen, "A Scalable Approach to Attack Graph Generation," *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pp. 336–345, 2006.

[4] J. Hale, P. Hawrylak, and M. Papa, "Compliance Method for a Cyber-Physical System." U.S. Patent Number 9,471,789, Oct. 18, 2016.

[5] N. Baloyi and P. 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), Association for Computing Machinery, 2019.

[6] E. Allman, "Complying with Compliance: Blowing it off is not an option.," *ACM Queue*, vol. 4, no. 7, 2006.

[7] 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*, pp. 254–265, 2002.

[8] J. Zhang, S. Khoram, and J. 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*, pp. 207–216, 2017.

[9] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, (New York, NY, USA), p. 217–224, Association for Computing Machinery, 2002.

[10] S. Jajodia and S. Noel, *Topological Vulnerability Analysis*, vol. 46, pp. 139–154. 09 2010.

[11] G. Louthan, *Hybrid Attack Graphs for Modeling Cyber-Physical Systems*. PhD thesis, The University of Tulsa, 2011.

[12] K. Cook, *RAGE: The Rage Attack Graph Engine*. PhD thesis, The University of Tulsa, 2018.

[13] W. M. Nichols, *Hybrid Attack Graphs for Use with a Simulation of a Cyber-Physical System*. PhD thesis, The University of Tulsa, 2018.