IEEE INSTRUMENTATION & MEASUREMENT SOCIETY

Received 09 November, 2022; revised XX Month, XXXX; accepted XX Month, XXXX; Date of publication XX Month, XXXX; date of current version XX Month, XXXX.

Digital Object Identifier 10.1109/OJIM.2022.1234567

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

NOAH L. SCHRICK*, MEMBER, IEEE, AND PETER J. HAWRYLAK†, SENIOR MEMBER, IEEE.

[1]Department of Computer Science, University of Tulsa, Tulsa, OK 74104 USA

CORRESPONDING AUTHOR: Noah L. Schrick (e-mail: noah-schrick@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 and Regulation; Compliance Graph; Cybersecurity; High-Performance Computing; Speedup; Synchronous Firing;

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]. These attack graphs take the form of Directed Acyclic Graphs (DAGs), where the root node is the initial state of the environment, and each subsequent node represents the new state of the environment after changes have occurred. 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 have also been extended to Cyber-Physical Systems (CPS) and the Internet of Things (IoT), and their usage can be seen in works such as that presented by the authors of [4], [5]. Various analysis metrics can then be performed, such as Bayesian attack graphs [6], maximum flow [7], and centrality-based ranking measures [8].

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 [9]–[11] discuss, cyber-physical systems have seen greater usage, especially in areas such as critical infrastructure and IoT. 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 and compliance 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 [12] 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 [13]. 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. Since every change in the network is examined individually, and no two changes can occur simultaneously, some nodes in the graph are created despite being infeasible. A leading cause of this is when examining an environment over time. Assets must undergo a time progression in the graph generation process, and by firing the time change separately for each asset, there is a synchronization problem where assets may progress through time disjointly from other assets. This work introduces a solution to this problem with synchronous exploit firing, which mitigates state space explosion for applicable scenarios while maintaining accuracy of the resulting graph, and discusses the performance results of its use.

II. Related Work

Multiple works have introduced various approaches for mitigating state space explosion. The authors of [14] 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 [15] 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 [16], 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 [16] 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 [16], 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 [16] 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 [16] 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 [17]. 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. Another 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 Fig. 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.

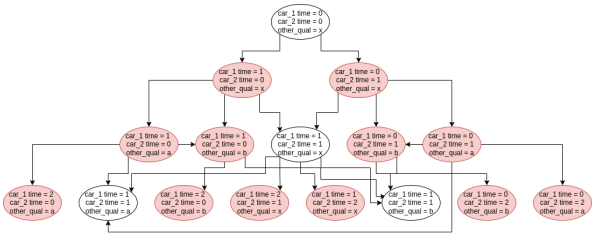

FIGURE 1. A network without Synchronous Firing generating infeasible states

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 incurring no 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

Synchronous exploit firing aims to eliminate the generation of infeasible states during the generation process, rather than needing an additional post-processing step. Using Fig. 1 as an example, the goal of synchronous firing is to only generate the 3 valid, unshaded nodes, rather than generate all 16 nodes since 13 of the nodes represent an impossibility where two cars are progressing through time at different rates. Synchronous firing is accomplished through new grouping keywords in the input exploit file, which propagate through the modified attack and compliance graph engine, and prevent the firing of exploits if they are part of a group where all members are not yet available to fire. For the implementation of the synchronous firing feature, there were four primary changes and additions that were required. The first is a change in the lexical analyzer, which handles the model and exploit input. Users indicate features that should fire simultaneously in these input files, and the lexical analyzer is responsible for parsing and passing that information to the graph generator. The second involves multiple changes to PostgreSQL to support the storage of group features and information. The third is the implementation of compound operators in RAGE, since enumeration of all exploits may not be possible, especially when modeling a system over time. The fourth is a change in the graph generation process, which checks to see if all exploits in a group are able to fire simultaneously before firing. The subsections in this Section describe these four alterations in greater detail.

A. GNU Bison and Flex

The work conducted by the author of [17] 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 [16] with the exploit keywords. This work implements the “group” keyword. The new 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. Fig. 2 illustrates the incorporation of this feature into Bison, Flex, and the overall program.

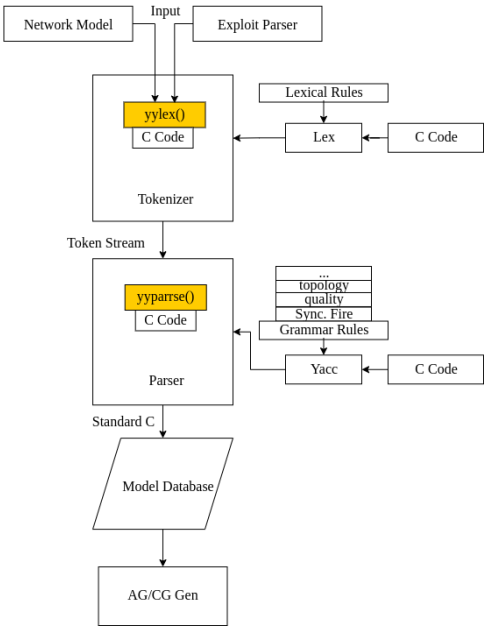

FIGURE 2. Inclusion of Synchronous Firing into GNU Bison, GNU Flex, and the overall program

B. PostgreSQL

As seen in Fig. 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. Additional care was taken to ensure that the normalization form of the database was not altered. Before adding the group identifier to its appropriate table, additional checking was performed to ensure there would be no partial functional dependencies or transitive dependencies.

C. 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 [18]. 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 [17], and compound operators were not included in the latest version of RAGE.

The work conducted by the author of [17] when designing the software architecture of RAGE included specifications for a quality encoding scheme. As they discuss, 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, Network-State, and Keyvalue classes) to include a new member for the

operator in question. In addition, preconditions were altered to include operator overloads to check the asset identifier, quality name, and quality values for the update process.

D. 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. Fig. 3 displays the synchronous fire approach.

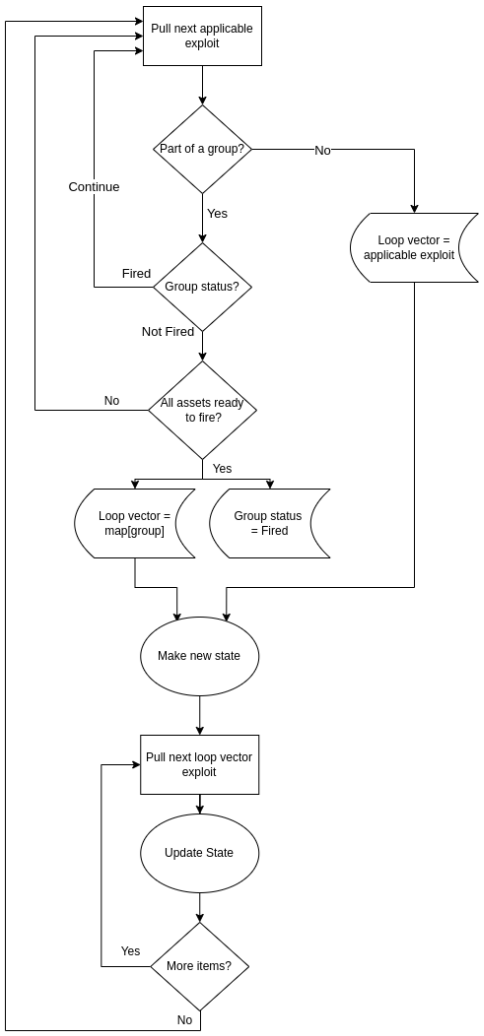

FIGURE 3. Synchronous Firing in the Graph Generation Process

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 8-core Intel Xeon E5-2620 v3
  - With hyperthreading: 2 threads/process per core
- 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.

1) Automobile Maintenance

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 test varies in the number of services used. The 1 Service case only employs one service, and it is dedicated to brake pads. The 2-Service case 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 cases. The experimental setup is as follows:

- All cases ran for 12 months, with time steps of 1 month.
- All cases 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 cases used the same network model.
- All cases used the same exploit file, with the exception of the “group” keyword being present in the synchronous firing testing.
- All 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

2) DMCA Takedown

A second example of synchronous firing is illustrated through a DMCA Takedown for a fictitious organization [19]. In this example, a DMCA Takedown is issued to an organization after a group of employees were found to be engaging in online piracy with torrenting software on company devices and while using company resources. Detection and removal of illicit data, such as through means presented by the authors of [20] for Windows or [21] for company-supplied Android mobile devices, can be incorporated into and represented by a compliance graph.

For this example, various graphs are generated based on the permutations of employees present. In one graph, only Employee A is present in the network. In another graph, Employees B and C are present in the network. All permutations are tested and are show in 3. The graph generation process walks through as a system administrator removes the torrenting software and the illicit data from the company devices. Typically when removing torrenting software, the data associated with the torrenting program can be removed at the same time as the uninstall automatically; an administrator does not need to remove the torrenting program and then separately remove the data. Without the use of synchronous firing, attack and compliance graphs must individually remove all data and all programs individually. This example highlights the capability of synchronous firing by grouping the removal of software and data together through "uninstall" groups, as opposed to traditional attack and compliance graphs requiring multiple steps to remove the software and data.

This experimental setup is as follows:

- Employee A has torrenting software, and is actively uploading and downloading 3 programs.
- Employee B has torrenting software, and is actively uploading and downloading 4 programs.
- Employee C has torrenting software, and is actively uploading and downloading 5 programs.
- If synchronous firing is not enabled, the administrator removes each illicit program one-by-one after the removal of the torrenting software.
- If synchronous firing is enabled, the administrator removes the torrenting software and all programs off a single device simultaneously.
- Graph visualization was not timed. Only the generation and database operation time was measured.

The compliance checks are as follows:

- Does an employee have torrenting software
- Does an employee have illicit data

B. Results and Analysis

1) Results for the Theoretical Automobile Environment

Using the experimental setup described in Section A on the platform described at the beginning of Section A, results were collected in regards to the effect of synchronous firing on both state space and runtime. The graphs' edge to state ratio (E/S Ratio) was computed as well. The inclusion of this ratio allows for a comparison to be drawn regarding the usage of the synchronous firing feature. Examining this ratio can provide additional insight on how the graph's underlying topological structures change when using or not using synchronous firing. The results can be seen in Figures 4 and 5. The respective tables are seen in Tables 1 and 2. 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. Fig. 6 displays the speedup (according to Amdahl's Law) obtained when using synchronous firing instead of non-synchronous firing for identical setups, as well as the state space reduction factor.

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.

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 Automobile Environment

The environment and resulting graphs presented in Section 1 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

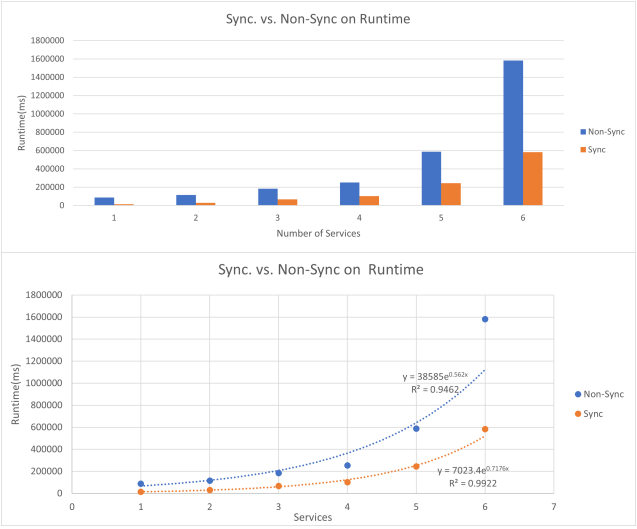

FIGURE 4. Bar Graph and Line Graph Representations of Synchronous Firing on Runtime

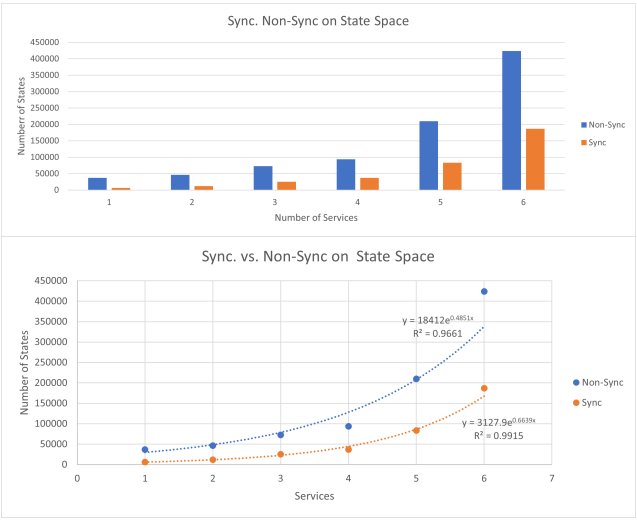

FIGURE 5. Bar Graph and Line Graph Representations of Synchronous Firing on State Space

TABLE 1. Results for the Non-Synchronous Firing Testing

Non-Synchronous Firing				
Number of Services	**Number of States**	**Number of Edges**	**Runtime (ms)**	**E/S Ratio**
1	37001	202920	87366.65	5.484
2	46361	259400	115929.97	5.595
3	72489	405236	184634.34	5.590
4	93525	546280	252959.511	5.841
5	209944	1254784	588336.01	5.977
6	423940	2712165	1581697.61	6.398

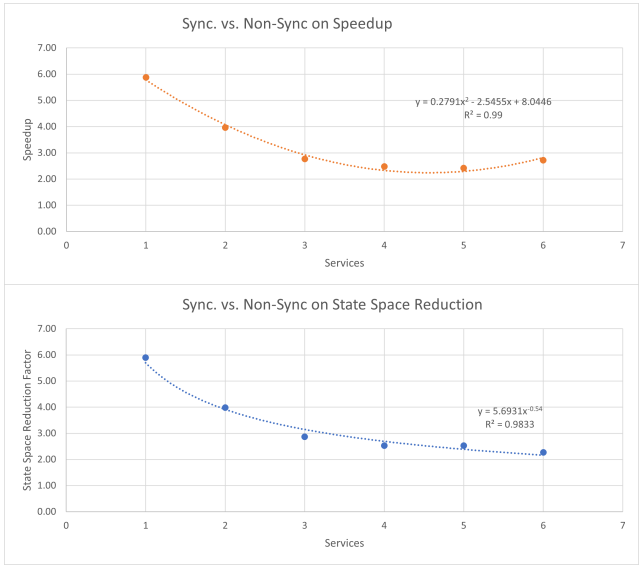

FIGURE 6. Speedup (Amdahl's) and State Space Reduction Factor Obtained When Using Synchronous Firing

TABLE 2. Results for the Synchronous Firing Testing

Synchronous Firing						
Services	**States**	**Edges**	**Runtime (ms)**	**E/S Ratio**	**Speedup**	**State Space Reduction**
1	6277	3.46E04	1.48E04	5.507	5.87	5.89
2	11653	6.94E04	2.92E04	5.954	3.96	3.98
3	25317	1.60E05	6.68E04	6.321	2.76	2.86
4	36949	2.42E05	1.02E05	6.538	2.47	2.53
5	83134	5.71E05	2.44E05	6.868	2.42	2.53
6	186679	1.35E06	5.82E05	7.209	2.72	2.27

same time, it 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 'comprehensive services", where a car would undergo multiple services simultaneously. With three services used, there are a total of three permutations: all three services are done individually, two services are performed simultaneously while the other is performed later, and all three services are performed simultaneously.

For this set of examples, all compliance checks have the same time requirements. This work does not introduce any heuristics or methodologies for intentionally performing services early or late. If Service A was required no later than every 6 months, but Service B was required no later than every 8 months, then joining Service A and Service B

together would either mean: 1. Service B was completed 2 months earlier than it needed to be, or 2. Service A was completed 2 months later than it needed to be. This was considered out of scope for this approach, but this is noted in the Future Works Section (Section VI).

These results are seen in Table 4 for the synchronous firing enabled generation, and Table 3 for the non-synchronous firing generation. The corresponding figures for the runtime can be seen in Fig. 7, and the corresponding figures for state space can be seen in Fig. 8. It is noticeable that there is a state space reduction achieved through synchronous firing in this set of examples, along with a runtime improvement. When all three services were conjoined, synchronous firing provided a 5.09x speedup over non-synchronous firing. Using comprehensive services on their own also provided a reduction in state space and an improvement in runtime. When synchronous firing was enabled and comprehensive services were used, the total number of states could be reduced from 25,317 to 3,774, providing a a 6.7x reduction in state space solely from combining services.

Leveraging comprehensive services with synchronous firing enables users to significantly reduce the size of the resulting attack or compliance graphs. Comprehensive services also enable users to introduce heuristics to analyze and identify optimal service plans for compliance, or attack mitigation strategies for attack graphs. Coupled with synchronous firing, analysis of these optimal plans can be performed quicker due to the inexistence of superfluous, unattainable states.

TABLE 3. Results for the Comprehensive Services without Synchronous Firing

Comprehensive Services with Non-Synchronous Firing				
Permutation	**States**	**Edges**	**Runtime (ms)**	**E/S Ratio**
All Disjoint	72489	405236	184634.34	5.590
Any Two Services, One Disjoint	50052	241176	125176.22	4.819
All Conjoined	19764	87024	47126.42	4.403

3) Results for the DMCA Takedown Environment

Using the experimental setup described in Section A on the platform described at the beginning of Section A, results were collected in regards to the effect of synchronous firing on both state space and runtime. The graphs' edge to state ratio (E/S Ratio) was computed as well. The respective tables are seen in Tables 5 and 6. The associated figure (Fig. 10) shows 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

TABLE 4. Results for the Comprehensive Services with Synchronous Firing

Comprehensive Services with Synchronous Firing					
Permutation	**States**	**Edges**	**Runtime (ms)**	**E/S Ratio**	**Speedup**
All Disjoint	25317	160041	66799.18	6.321	2.76
Any Two Services, One Disjoint	10398	55354	26042.85	5.324	4.81
All Conjoined	3774	18370	9261.03	4.868	5.09

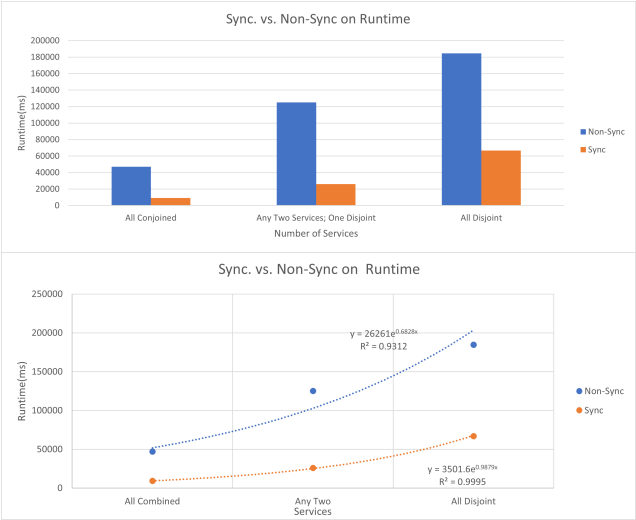

FIGURE 7. Bar Graph and Line Graph Representations of Synchronous Firing with Comprehensive Services on Runtime

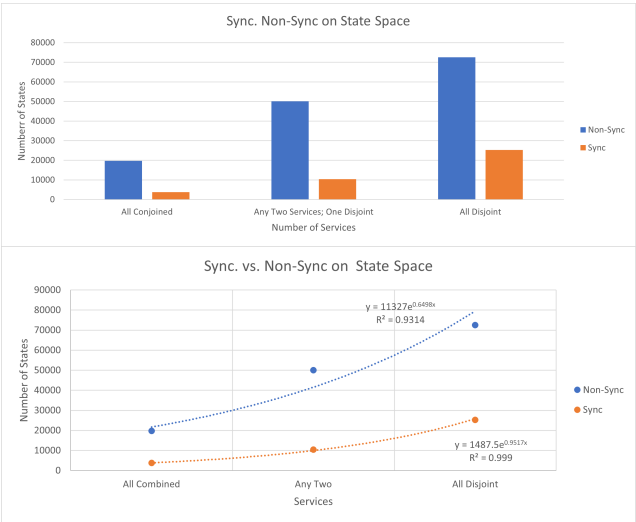

FIGURE 8. Bar Graph and Line Graph Representations of Synchronous Firing with Comprehensive Services on State Space

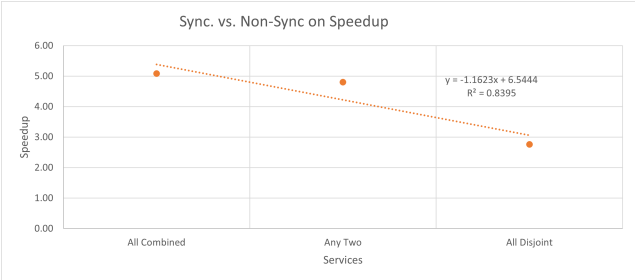

FIGURE 9. **Speedup (Amdahl's) Obtained When Using Synchronous Firing with Comprehensive Services**

states, there is no meaningful information loss that occurs in the graphs generated with the synchronous firing feature. Fig. 11 displays the speedup (according to Amdahl's Law) obtained when using synchronous firing instead of non-synchronous firing for identical setups, as well as the state space reduction factor.

In this example, the synchronous firing approach mitigates the state space explosion by an increasing factor. With traditional attack and compliance graph generation, the uninstall process is required to be broken into individual steps, causing an unnecessarily large growth in the resulting graph. This is exacerbated due to the presence of multiple employees transmitting multiple pieces of illicit data, all of which must be captured individually. Using synchronous firing allows for better modeling of real systems, where features, processes, or tasks are often combined into single steps.

TABLE 5. **Results for the Non-Synchronous Firing Testing**

Non-Synchronous Firing				
Employees Present	**Number of States**	**Number of Edges**	**Runtime (ms)**	**E/S Ratio**
A	16	32	11.109	2.000
B	32	80	26.202	2.500
C	64	192	64.012	3.000
AB	512	1984	906.578	3.875
AC	1024	4032	2154.862	3.938
BC	2048	8128	5154.858	3.969
ABC	32768	131008	105675.709	3.998

TABLE 6. **Results for the Synchronous Firing Testing**

Synchronous Firing						
Employees Present	**States**	**Edges**	**Runtime (ms)**	**E/S Ratio**	**Speedup**	**State Space Reduction**
A	2	1	3.810	0.500	2.92	8.00
B	2	1	5.214	0.500	5.03	16.00
C	2	1	9.063	0.500	7.06	32.00
AB	4	4	67.560	1.000	13.42	128.00
AC	4	4	117.833	1.000	18.29	256.00
BC	4	4	108.544	1.000	47.49	512.00
ABC	8	12	705.016	1.500	149.89	4096.00

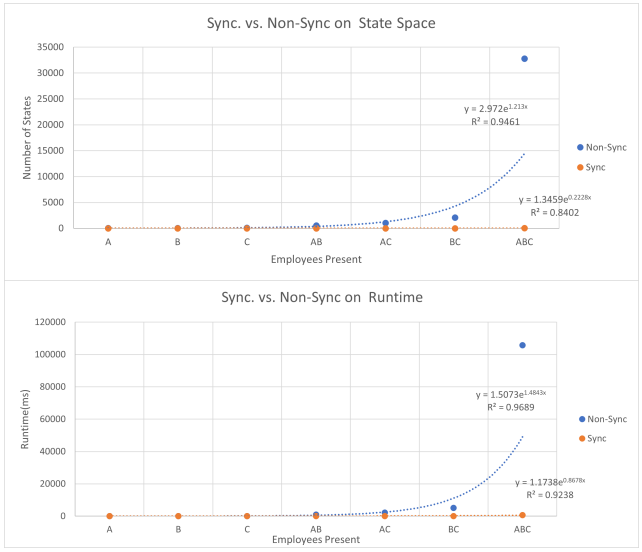

FIGURE 10. **Synchronous Firing on State Space and Runtime for the DMCA Takedown Environment**

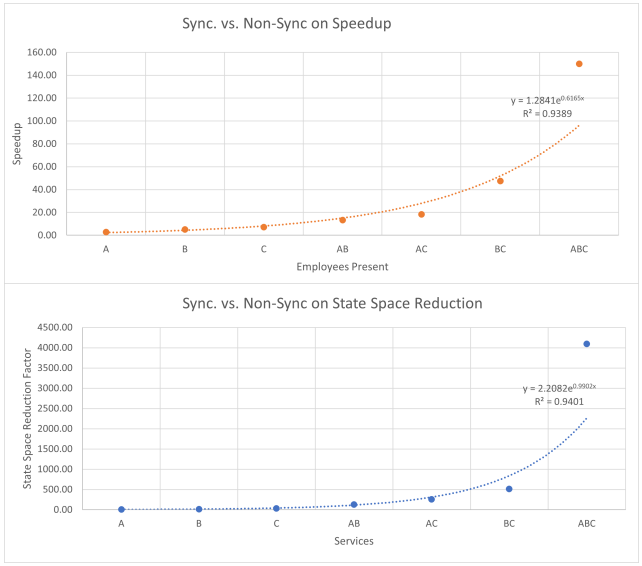

FIGURE 11. **Speedup (Amdahl's) and State Space Reduction Factor Obtained When Using Synchronous Firing**

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 scenario to have a greater number of

assets, and a standard set of exploits more akin to an attack graph rather than a compliance graph. Other future works could include measuring the performance of synchronous firing when multiple groups of inseparable 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 work 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 of 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.

Introducing service heuristics could improve the characteristics of synchronous firing. If services are performed too early, then additional states would be generated in the resulting graph. If synchronous firing was not used, these additional states could compound into more states due to the separation of features. Likewise, if services are performed too late, then additional states could be generated to represent the compliance violation, and these states may also compound into more statues without synchronous firing. Examining the impact of synchronous firing when various heuristics are implemented could reveal interesting results.

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 Fig. 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] A. T. Al Ghazo, M. Ibrahim, H. Ren, and R. Kumar, "A2g2v: Automated attack graph generator and visualizer," in *Proceedings of the 1st ACM MobiHoc Workshop on Mobile IoT Sensing, Security, and Privacy*, Mobile IoT SSP'18, (New York, NY, USA), Association for Computing Machinery, 2018.

[5] M. Li, P. Hawrylak, and J. Hale, "Strategies for practical hybrid attack graph generation and analysis," *Digital Threats*, oct 2021. Just Accepted.

[6] L. Muñoz González, D. Sgandurra, A. Paudice, and E. C. Lupu, "Efficient attack graph analysis through approximate inference," *ACM Trans. Priv. Secur.*, vol. 20, jul 2017.

[7] H. Wang, Z. Chen, J. Zhao, X. Di, and D. Liu, "A vulnerability assessment method in industrial internet of things based on attack graph and maximum flow," *IEEE Access*, vol. 6, pp. 8599–8609, 2018.

[8] T. Gonda, T. Pascal, R. Puzis, G. Shani, and B. Shapira, "Analysis of attack graph representations for ranking vulnerability fixes," 09 2018.

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

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

[19] "H.r.2281 - Digital Millennium Copyright Act." Pub. L. No. 105-304. 1998 [Online]. Available: https://www.govinfo.gov/content/pkg/PLAW-105publ304/html/PLAW-105publ304.htm.

[20] Y. Kim, J. Moon, S. J. Cho, M. Park, and S. Han, "Efficient identification of windows executable programs to prevent software piracy," in *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 236–240, 2014.

[21] N. Kumari and M. Chen, "Malware and piracy detection in android applications," in *2022 IEEE 5th International Conference on Multimedia Information Processing and Retrieval (MIPR)*, pp. 306–311, 2022.