Application-Level Checkpoint/Restart for Large-Scale Attack and Compliance Graphs

1st Noah L. Schrick
Tandy School of Computer Science
University of Tulsa
Tulsa, USA
noah-schrick@utulsa.edu
ORCID: 0000-0003-0875-8927

2nd Peter J. Hawrylak
Tandy School of Computer Science
University of Tulsa
Tulsa, USA
peter-hawrylak@utulsa.edu
ORCID: 0000-0003-3268-7452

***Abstract*—Attack graphs and compliance graphs are graphical representations of systems used to analyze their standings in regards to cybersecurity or compliance postures. These graphs are designed to examine all permutations and combinations of changes that can occur to systems, and represent those changes through the topological information of the resulting graph. Due to their exhaustive nature, these graphs are often large-scale. Various efforts have shown success in the parallelization and generation deployment on High-Performance Computing (HPC) systems. Despite the additional compute power that HPC systems provide, these large-scale graphs still posses lengthy runtimes and require large amounts of system memory to fully contain the resulting graph. This work presents a Checkpoint/Restart (C/R) implementation specific to attack and compliance graphs that aims to provide fault-tolerance and memory relief throughout the generation process. This implementation has been designed to be built within the graph generation source code, and provides a lightweight alternative to other C/R implementations. The results indicate that this implementation is successful at providing fault-tolerance and memory relief while having low impact to the generation process and to the source code.**

***Index Terms*—Attack Graph; Compliance Graph; MPI; High-Performance Computing; Checkpoint/Restart;**

I. Introduction

In order to predict and prevent the risk of cyber attacks, various modeling and tabletop approaches are implemented to best prepare for attack scenarios. One approach is through the use of attack graphs, originally presented by the author of [1]. Attack graphs represent possible attack scenarios or vulnerability paths in a network. These graphs consist of nodes and edges, with various information encoded at the topological level as well as within the nodes themselves. Similarly, compliance graphs are used to predict and prevent violations of compliance or regulation mandates [2]. These graphs are now generated through the use of attack or compliance graph generators, rather than by hand. The generator tool used for the implementation of this work is RAGE (the RAGE Attack Graph Engine) [3].

Despite their advantages, graph generation has many challenges that prevent full actualization of computation seen from a theoretical standpoint, and these challenges extend to attack and compliance graphs. In practice, graph generation often achieves only a very low percentage of its expected performance [4]. A few reasons for this occurrence lie 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 [4], [5], [6]. 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 [4], [6].

The author of [3] discusses the challenges of attack graph generation in regards to its scalability. Specifically, the author of [3] displays results from generations based on small networks that result in a large state space. The authors of [7] also present the scalability challenges of attack graphs. Their findings indicate that small networks result in graphs with total edges and nodes in the order of millions. Generating an attack or compliance graph based on a large network with a multitude of assets and involving a more thorough exploit or compliance violation checking will prevent the entire graph from being stored in memory as originally designed.

Due to the runtime requirements and scalability challenges imposed by graph generation, fault-tolerance is critical to ensure reliable generation. These difficulties highlight the need for fault-tolerance and non-volatile memory (memory) relief approaches. The ability to safely checkpoint and recover from a system error is crucial to avoid duplicated work or needing to request more cycles on an HPC cluster. In addition, having the ability to minimize the memory strain without requesting excess RAM on an HPC cluster assists in reducing incurred cost. This work presents an application-level checkpoint/restart (C/R) approach tailored to large-scale graph generation. This work illustrates the advantages in having a C/R system built into the generation process itself, rather than using alternative libraries. By having native C/R, performance can be maximized and runtime interruption and overhead can be minimized. This C/R approach allows the user to ensure fault-tolerance for graph generation without the reliance on a system-level, HPC cluster implementation of C/R.

II. Related Work

Numerous efforts have been presented for C/R techniques with various categories available. The authors of [8] and [9] discuss three categories of C/R, which include application-level, user-level, and system-level. Each approach draws upon advantages that appeal toward different aspects of reliability. User-level checkpointing, though has greater simplicity, results in larger checkpoints. System-level requires compatibility with the operating system and any libraries used for the application. Application-level checkpointing requires additional work for the implementation, but resuls in smaller, faster C/R. The authors of [10] present the SCR (Scalable Checkpoint/Restart) library, which has seen widespread adoption due to its minimal overhead. DMTCP (Distributed MultiThreaded Checkpointing) [11] and BLCR (Berkely Lab Checkpoint/Restart) [12] are two other commonly-used C/R approaches.

Other investigations into attack and compliance graphs attempt to improve performance and scalability to mitigate state space explosion or lengthy runtimes, rather than focus on C/R. These investigations include the works by the authors of [13], which implement attack graph methodologies using Neo4j for efficient storage techniques. This approach has seen other implementations, such as that shown by the authors of [14]. Other attack graph scalability studies involve the alteration of the representation schemes. The authors of [7] make use of logical statements for logical attack graphs. This approach has seen continued investigations, and similar logic-based attack graphs can be seen in the work presented by the authors of [15]. These logical based attack graphs aim to improve scalability by minimizing the resulting information. Other representation schemes include those seen by the authors of [16] and the authors of [17], which make use of qualities and topologies through graph states. Scalability improvements have also been examined through sampling, such as the approach presented by the authors of [18]. Parallelization techniques have been investigated for runtime improvement, and successful results have been seen in the work by the authors of [19].

III. Methodology

A. Checkpointing

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 [3], [17], and [20]. To this end, the output graph is contained in memory during the generation process to minimize disk writing and reading. RAGE does incorporate PostgreSQL as an initial and final storage mechanism to write the starting and resulting graph information, but no intermediate storage is otherwise conducted. Based on the inclusion of PostgreSQL in RAGE, the C/R approach was based around this dependency. Fig. 1 shows an image of an attack or compliance graph that is undergoing the generation process. All nodes and edges within the "instance" box have been fully explored, and all information is stored in memory. All nodes within the "frontier" box have their information stored in memory, but they have not yet undergone exploration. To checkpoint at this point in time, both the instance and the frontier need to be saved. Additionally, since the instance will no longer be used, it can be fully removed from memory. Section III-A1 highlights the advantages and necessities of this removal.

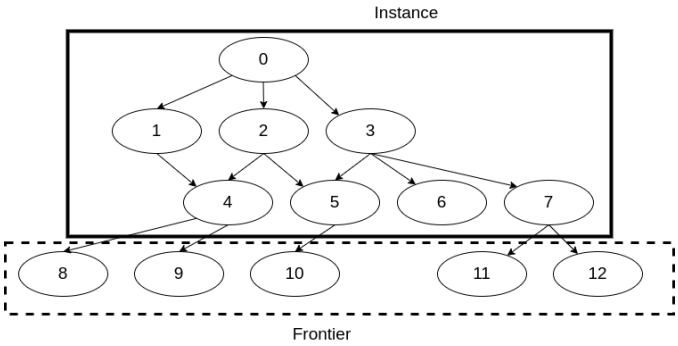

Fig. 1. An Attack or Compliance Graph Undergoing Generation

1) Memory Constraint Difficulties: While the design decision to store all graph generation information in memory maximizes performance, it introduces a few complications. When generating large graphs, the system runs the risk of running out of memory. This typically does not occur when generation is conducted on small graphs, and is especially true when relatively small graphs are generated on an HPC system with substantial amounts of 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 states and their information, as well as all of the edges.

The frontier rapidly strains memory with large graphs that have a large height value and contain many layers before the 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 a rapidly increasing frontier that can not be adequately reduced for large networks. Simultaneously, the graph instance continues to grow 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 [5], it becomes increasingly unlikely that the graph can be fully contained within system memory. Checkpointing provides an additional benefit to the generation process to relieve its memory strain.

2) Implementation: Rather than only a static implementation of storing to the database on disk at a

set interval or a set size, the goal was to also allow for dynamically storing to the database only when necessary. This would allow for proper utilization of systems with greater memory, and would reduce fine-tuning of a maximum size variable before database writes on different systems. Since there is an associated cost with preparing the writes to disk, the communication cost across nodes, the writing to disk itself, and a cost for retrieving items from disk, it may be desirable to store as much in memory for as long as possible and only checkpoint 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. Alternatively, an integer greater than or equal to 1 can be passed, which allows for a discrete number of states to be held in memory before checkpointing. If the value passed is between 0 and 1, it is immediately reduced by a static subtraction of 10%. For instance, if 0.6 (60%) is passed, it is immediately reduced by a static subtraction of 10% to yield 0.5 (0.6-0.1=0.5, or 60%-10%=50%). 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 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 graph instance is allowed to consume the other half.

To decide when to checkpoint due to memory capacity, 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, if the frontier is empty, if the unexplored states table is not empty. The original generation design stored new states into the frontier during an OpenMP critical section to avoid testing on already-explored states. To follow this design decision, writing new states to the database is also performed during the critical section.

For the graph instance, a check in the critical section determines if the size of the graph 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.

The original design was to save staging, preparation, and communication cost by writing all the data in one query (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 option in terms of performance, it was also not feasible when the amount of data to store was large in relation to system memory. 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 invoke the Out-of-Memory Killer. 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) Portability: The checkpointing process 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, additional functionality can be built for alternative operating systems. When working on an HPC 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 checkpointing process would function in the same fashion. Since PostgreSQL is used for the checkpointing, no file system dependencies are necessary for the cluster.

B. Restarting

The restarting process for attack and compliance graph generation requires only a limited set of information. In order for a proper generation restart, the generator first needs to pull the unexplored queue ("frontier") database table into memory. After the frontier is loaded, the generator tool needs to know at which integer to begin tagging new states. This is accomplished by checking the ID of the latest state in the frontier. Since the instance has already been explored, it does not need to be retrieved from disk. In addition, the frontier is a queue of unexplored nodes. Since these nodes have yet to be explored, no edge information is available, and as a result, no edge information is required. At this point, the generator tool is able to pop the first node from the queue and resume the generation process.

IV. Results

Results were collected to measure the checkpointing time as a function of the instance size in bytes, as well as the frontier size in bytes. The restart results were collected to measure the time taken to restart as a function of the frontier size in bytes. In order to mitigate the non-deterministic effects of networking as much as possible, the checkpointing and restarting processes

were called by a compute node that was also running the PostgreSQL instance.

Fig. 2 displays the time taken to checkpoint as the instance size grows. The size of the instance was measured in bytes to allow for abstracting the number of items in the instance and the size of each item. This figure demonstrates that as the instance grows, the time taken to checkpoint remains linear. This figure includes two trendlines - one for a linear fit, and one for a logarithmic fit. Though the logarithmic fit appears to match the data better than the linear fit when the instance size is less than 20MB, the accuracy begins to decrease after this size. Based on the timing data obtained, the logarithmic fit predicts higher than the actual data for smaller instances, and the linear fit predicts lower than the actual data.

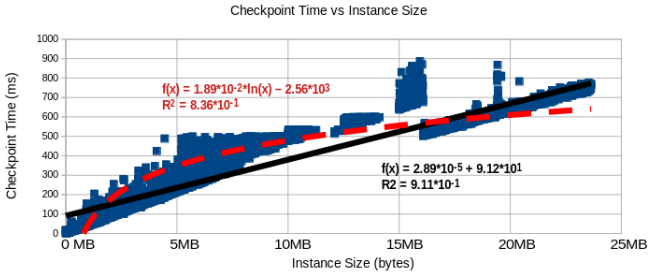

Fig. 2. Time Taken to Checkpoint as the Size of the Instance Grows

Fig. 3 displays the time taken to checkpoint as the frontier size grows. Similar to the instance size, the frontier was also measured in bytes for consistency. This figure illustrates that as the frontier grows, the checkpoint time also remains linear. Compared to the time taken to checkpoint the instance, the frontier requires less time. Since the graph instance contains various topological information such as edges and edge labels, additional time is required compared to the frontier.

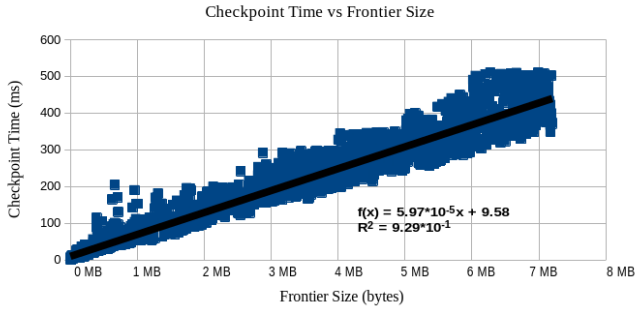

Fig. 3. Time Taken to Checkpoint as the Size of the Frontier Grows

Fig. 4 displays the time taken to restart as the frontier size grows. The frontier was again measured in bytes. This figure illustrates that the restart time remains linear as the size of the frontier grows. This figure shows that there is less time taken to restart due to the limited amount of information needed to restart compared to the information needed to checkpoint. Since the checkpointing process involves saving the graph instance and the frontier, extra time is required compared to the restart process which only requires the queue of unexplored states.

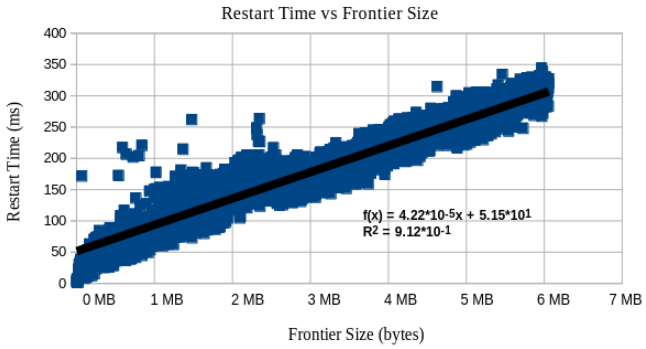

Fig. 4. Time Taken to Restart as the Size of the Frontier Grows

V. Conclusions and Future Work

This work presents an application-level approach at C/R. This approach was built into RAGE itself, and has no dependencies on C/R libraries. In addition, it does not need support from the operating system, allowing for fault-tolerance on HPC clusters that may not support C/R. The results highlight the minimal time requirement to both checkpoint and restart the generation process. Since only the necessary information is stored and retrieved, there are no lengthy function calls or snapshots that are required. The C/R implemented also serves as a form of memory relief. Due to the size of large-scale attack and compliance graphs, there is increased difficulty in storing all information in memory. This approach allows users to abstract away the memory constraint difficulties while maximizing performance.

Future work includes performance and size comparisons to available C/R libraries. This would include implementing SCR, BLCR, and/or DMTCP and measuring their respective checkpoint times and sizes, as well as time taken to restart. This timing information can then be compared to the checkpoint and restart time presented by this work. Future work could also include new optimization techniques. Additional investigations can provide insight on techniques for reducing the runtime of database queries, PostgreSQL database settings to alter or enable, or communication strategies between distributed nodes. Other work can involve alternative approaches at C/R. This work made use of PostgreSQL since it was already incorporated into RAGE, but future work can look toward alternatives. Filesystem C/R approaches can be investigated to determine possible advantages over a dedicated database. One other route for future work involves identifying the checkpointing interval. The work presented by the authors of [21] discusses a multitude of options for determining an optimal interval. Various options presented in their work can be implemented, along with their closed-form solution for identifying a default checkpointing interval that can be built into RAGE.

REFERENCES

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

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

[3] K. Cook, "RAGE: The Rage Attack Graph Engine," Master's thesis, The University of Tulsa, 2018.

[4] J. Berry and B. Hendrickson, "Graph Analysis with High Performance Computing.," *Computing in Science and Engineering*, 2007.

[5] 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.

[6] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," *Proceedings of the International Conference on Supercomputing*, vol. 01-03-June, 2016.

[7] 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.

[8] F. Shahzad, M. Wittmann, T. Zeiser, G. Hager, and G. Wellein, "An evaluation of different i/o techniques for checkpoint/restart," in *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, pp. 1708–1716, 2013.

[9] J. Hursey, *Coordinated checkpoint/restart process fault tolerance for MPI applications on HPC systems*. Indiana University, 2010.

[10] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2010.

[11] J. Ansel, K. Arya, and G. Cooperman, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," 2009.

[12] J. Duell, P. H. Hargrove, and E. S. Roman, "Requirements for linux checkpoint/restart," 2 2002.

[13] G. Simon-Nagy, R. Fleiner, and A. Bánáti, "Attack graph implementation in graph database," in *2022 IEEE 20th Jubilee International Symposium on Intelligent Systems and Informatics (SISY)*, pp. 000127–000132, 2022.

[14] B. Yuan, Z. Pan, F. Shi, and Z. Li, "An attack path generation methods based on graph database," in *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, vol. 1, pp. 1905–1910, 2020.

[15] F. Yi, H. Y. Cai, and F. Z. Xin, "A logic-based attack graph for analyzing network security risk against potential attack," in *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–4, 2018.

[16] K. Cook, T. Shaw, J. Hale, and P. Hawrylak, "Scalable attack graph generation," *Proceedings of the 11th Annual Cyber and Information Security Research Conference, CISRC 2016*, 2016.

[17] M. Li, P. Hawrylak, and J. Hale, "Concurrency Strategies for Attack Graph Generation," *Proceedings - 2019 2nd International Conference on Data Intelligence and Security, ICDIS 2019*, pp. 174–179, 2019.

[18] O. Subasi, S. Purohit, A. Bhattacharya, and S. Chatterjee, "Impact-driven sampling strategies for hybrid attack graphs," in *2022 IEEE International Symposium on Technologies for Homeland Security (HST)*, pp. 1–7, 2022.

[19] K. Kaynar and F. Sivrikaya, "Distributed attack graph generation," *IEEE Transactions on Dependable and Secure Computing*, vol. 13, no. 5, pp. 519–532, 2016.

[20] M. Li, P. Hawrylak, and J. Hale, "Combining OpenCL and MPI to support heterogeneous computing on a cluster," *ACM International Conference Proceeding Series*, 2019.

[21] N. El-Sayed and B. Schroeder, "Checkpoint/restart in practice: When 'simple is better'," in *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*, pp. 84–92, IEEE Computer Society, 2014.