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

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 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 generators, rather than by hand. The generator tool used by 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 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 handle 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. Notably, application-level requiring additional work for the implementation but resulting in smaller, faster C/R, user-level with its simplicity, but resulting in larger checkpoints, and system-level requiring compatibility with the operating system and any libraries used for the application. 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-seen C/R approaches.

Rather than using C/R, investigations into attack and compliance graphs attempt to improve performance and scalability to mitigate state space explosion or lengthy runtimes. As a means of improving scalability of attack graphs themselves, the authors of [7] present a new representation scheme. Traditional attack graphs encode the entire network at each state, but the representation presented by the authors uses logical statements to represent a portion of the network at each node. This is called a logical attack graph. This approach led to the reduction of the generation process to quadratic time and reduced the number of nodes in the resulting graph to $\mathcal{O}(n^2)$. However, this approach does require more analysis for identifying attack vectors. Another approach presented by the authors of [13] represents a description of systems and their qualities and topologies as a state, with a queue of unexplored states. This work was continued by the authors of [14] by implementing a hash table among other features. Each of these works demonstrates an improvement in scalability through refining the desirable information output.

III. Implementation

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], [14], and [15]. 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.

A. Memory Constraint Difficulties

While the design decision to not use intermediate storage maximizes performance for graph generation, 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.

B. Checkpointing

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 10%. For instance, if 0.6 is passed, it is immediately reduced to 0.5. This acts as a buffer for PostgreSQL. Since queries will consume a variable amount of memory through parsing or preparation, an additional 10% is saved as a precaution. This can be changed 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 the unexplored states table is not empty. Due to C++ using short-circuit evaluation where the first argument is completely evaluated before processing the second, some performance is gained. The performance gained is due to not having to pass a SQL statement to disk to check the size of the unexplored states table unless the frontier is empty. The original generation design stored new states into the frontier during the 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.

1) 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.

C. Restarting

IV. Results

V. Conclusions and Future Work

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*. PhD 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] 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.

[14] 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.

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