Scalable Attack Graph Generation

Kyle Cook
The University of Tulsa
kylecook@utulsa.edu

Thomas Shaw
The University of Tulsa
thomas-shaw@utulsa.edu

Dr. Peter Hawrylak
The University of Tulsa
peter-hawrylak@utulsa.edu

Dr. John Hale
The University of Tulsa
john-hale@utulsa.edu

ABSTRACT

Attack graphs are a powerful modeling technique with which to explore the attack surface of a system. However, they can be difficult to generate due to the exponential growth of the state space, often times making exhaustive search impractical. This paper discusses an approach for generating large attack graphs with an emphasis on scalable generation over a distributed system. First, a serial algorithm is presented, highlighting bottlenecks and opportunities to exploit inherent concurrency in the generation process. Then a strategy to parallelize this process is presented. Finally, we discuss plans for future work to implement the parallel algorithm using a hybrid distributed/shared memory programming model on a heterogeneous compute node cluster.

CCS Concepts

•**Security and privacy** → *Formal security models; Vulnerability management;* •**Computing methodologies** → Parallel algorithms;

Keywords

Attack Graphs, Vulnerability Analysis, Attack Modeling

1. INTRODUCTION

Modeling networks to check for security vulnerabilities is not a new problem. However, current networks have grown substantially in size, and modeling systems in a reasonable amount of time remains a significant challenge. Comprehensive security analysis of a network mandates a full accounting of exposures and the potential interactions between them. Attack graphs describe how a series of actions (e.g., attacks) can alter a system with the ultimate goal of compromising some security property of the system [4, 7, 11]. The large size of modern networks makes complete attack graphs difficult to generate due to their size and complexity.

CISRC '16, April 05-07, 2016, Oak Ridge, TN, USA

DOI: http://dx.doi.org/10.1145/2897795.2897821

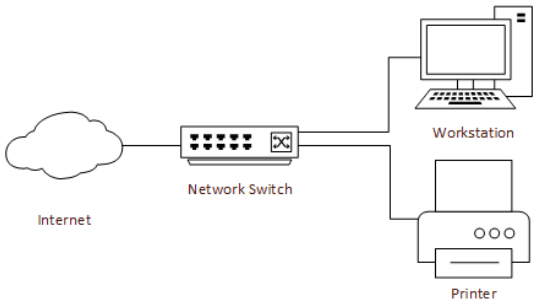

Figure 1: A Simple Network State ©Kyle Cook

In this paper, we present a scalable approach for generating attack graphs of large networks. First, a serial algorithm is presented along with a complexity analysis that reveals bottlenecks and opportunities for parallelism. Then a parallel algorithm is presented and its relative performance characteristics are compared against the serial approach.

2. ATTACK GRAPHS

Modeling exploit methods is vital for security. As systems grow in size, the number of possible exposures grow with it. Modern systems employ very large networks which must be maintained on a regular basis, but typically only a few people are able to manage them, leaving some areas vulnerable. Several methods of modeling the security of these networks have been proposed [6, 10, 12]. One such method is the **attack graph**, which is a directed graph that describes a particular network and how exploits can be chained together to compromise a complex system [8, 11].

2.1 Overview

Attack graphs characterize the attack surface of a system through a functional treatment of exploit patterns. When checking for exposures in a network, each possible exploit must be checked against all items in the network. In our attack graph vernacular, these items are called **assets**. Assets have **facts** associated with them. Facts are either **qualities**, which describe properties of the asset, or **topologies**, which describe the asset's relation to other assets. Collectively, assets and facts comprise a **network state**.

An **exploit** consists of preconditions (requirements for the exploit) and postconditions (results of the exploit). After an exploit has been applied to a network state, a new state

```
Network Model
    Assets:
        Workstation
        Printer
        Network Switch
    Facts:
        Quality: Workstation, "Windows 7"
        Quality: Workstation,
                    adobe_reader_version,
                    15.010.20056
        Quality: Workstation, anti-virus, false
        Quality: Workstation, root, false
        Quality: Printer, firmware, 6.5.4
        Quality: Printer compromised, false
        Quality: Switch, firmware, "7.4.1.34"
        Quality: Switch, manufacturer, "Cisco"
        Quality: Switch, compromised, false
        Topology: Switch, Workstation, connected
        Topology: Switch, Printer, connected
        Topology: Switch, Internet, connected
        Topology: Workstation, Printer, trust
```

Figure 2: Schema of Figure 1 ©Kyle Cook

is generated by applying the postconditions to the current network state. In this sense, an exploit pattern operates like a function. New states are subsequently checked against exploits to reveal new viable attacks. As each exploit is applied to each network state, a graph is generated that describes all possible chains of attacks on a given network.

The resulting graph can be processed to determine state reachability, probability of exploits, and other risk metrics [4, 9]. Heuristics, acceleration techniques, and even HPC hardware can be used in order to conduct practical and timely analysis on larger and larger networks.

2.2 Example

Consider the network in Figure 1. The topology is as follows: The switch connects to the workstation, the switch connects to the printer, and the internet connects to the switch. This characterizes the initial network state. Figure 2 shows the textual representation, or network model, of the simple network state in Figure 1.

Figure 3 gives a simple attack graph example. The e denotes an exploit. There is a vulnerability in the network state susceptible to a particular virus. The network is now in a state where an attack on the printer can be executed to gain root access. Alternatively, an attacker could exploit a printer firmware rollback vulnerability, allowing a firmware attack on the printer to be executed. In this way, multiple paths can be taken to achieve the same network state.

2.3 Serial Generation Algorithm

The generation algorithm attempts to explore all possible chain of attacks on a given network state. A network state is a collection of assets, facts about those assets, and asset relationships. We are given a list of exploits which, when its postconditions are applied to a network state, may generate new states to which other exploits can then be applied.

The serial algorithm repeatedly expands network states until none remain, at which point the entire state space has been explored and no exploits apply to any of the current network states. This admits the potential of an infinite graph. Alternative implementations limit state explosion by depth, distance, or some constant value, such as generating nodes up to x nodes away from the initial state.

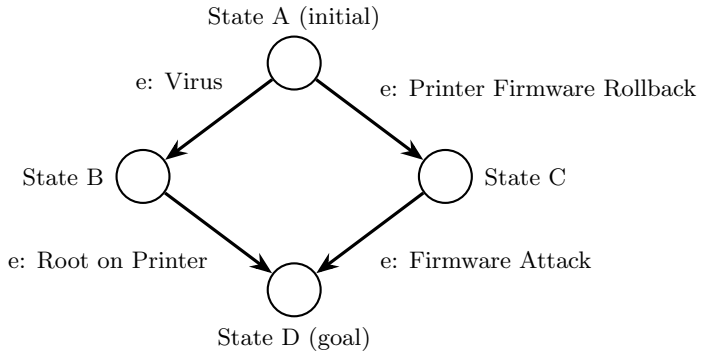

Figure 3: A Simple Attack Graph ©Kyle Cook

Algorithm 1 begins by loading the initial network state into the *unexpanded* queue. We then iterate over each unexpanded network state (which is, at first, just the initial state), and apply each exploit to that network state.

To find which assets in the network have an applicable exploit, all permutations (with repetitions) of the assets in the exploit parameter list must be checked. Calculating all permutations with repetition of an exploit parameter list is $O(A^P)$ where A is the number of assets in the network state and P is the number of parameters in the exploit.

Algorithm 1 Serial Attack Graph Generation

```
1: procedure SERIAL_AG_GEN(assets, exploits)
2:     unexpanded ← initial_network_state
3:     for exploit e in exploits do
4:         bindings_e ← permute_r(assets, e.params)
5:     while not done do
6:         network_state ← dequeue(unexpanded)
7:         for exploit e in exploits do
8:             new_states ← match(
                   bindings_e,
                   network_state)
9:             enqueue(unexpanded, new_states)
10:        move network_state to expanded
```

Algorithm 2 Serial Match Function

```
1: function MATCH(bindings, state)
2:     for binding in bindings do
3:         new_state ← check(binding, exploit.preconds)
4:         if new_state then
5:             enqueue(new_states, new_state)
6:     return new_states
```

Each permutation is considered an **asset binding**. Each asset is bound to a exploit parameter in the exploit pattern, defining a bound exploit. The pattern is evaluated by checking each fact in the bound exploit and comparing this to the facts in the network state. If the facts match, then the bound exploit can be applied to the network state.

The *match* function (Algorithm 2) compares an exploit to a network state to determine if new states can be generated. If any new states are able to be generated, then the original network state can be exploited successfully. A new list of states is returned, which is then enqueued into the *unexpanded* queue. The current network state is then moved from *unexpanded* to *expanded*. If the graph is finite and the unexplored queue is empty, the algorithm terminates.

Because we generated all bindings earlier, we pass only the relevant list of bindings to match. We iterate over each binding and call the *check* function. *check* applies the asset binding to the parameters in the exploit and determines if the preconditions in the exploit match the assets. If they match, a new state is generated. If they do not match, we move on to the next binding and check again.

2.4 Complexity Analysis

The time complexity of the serial algorithm reveals bottlenecks and opportunities to exploit concurrency for scalable generation. Let all possible network states in the state space be S, all assets in a network model be A, all exploits be E, all facts in an exploit be F, and finally all parameters of some E be denoted as P.

In this algorithm, we assume that the entire state space should be explored. However, it is possible that S converges to infinity, which suggests that the algorithm is $O(\infty)$. Alternative formulations of S consider that the state space S is finite and explorable in a finite amount of time.

In the *match* function, line 3 shows a call to *check*, which is essentially a search routine. *check* tries to find each fact in the current exploit's factlist against those in the asset binding. Assuming these facts are in sorted order, *check* is done in $O(\log F)$. Line 2 shows a **for** loop which iterates over all A^P bindings applicable to this exploit. Therefore, the *match* function has a complexity of $O(\log FA^P)$.

In the *serial_ag_gen* procedure, line 8 calls *match*, which runs in $O(B \log F)$ (line 9 is constant time). These operations are inside a **for** loop (line 7). Thus the complexity of the code from 7-9 is $O(E \log FA^P)$. Lines 6 and 10 are constant time operations, and the **while** loop executes as long as unexplored states remain. Absent any heuristic truncating execution, this implies the **while** loop will explore every state in the state space, so the time for lines 5-10 is $O(SE \log FA^P)$. Line 4 calls *permute_r*, which has a complexity of $O(FA^P)$ (every binding permutation with repeats applied to F exploit preconditions and postconditions) and is executed E times, so lines 3-4 have a complexity of $O(EFA^P)$. Line 2 is constant. Combining these, *serial_ag_gen* has a complexity of $O(SEFA^P)$.

3. SCALABLE GRAPH GENERATION

We now consider opportunities to exploit the latent concurrency within the serial algorithm. The most obvious candidates for parallelism are found in the **for** and **while** loops of the algorithm. Successfully transforming these into parallel operations requires ensuring that loop-carried dependencies do not exist and that other obstacles are addressed.

3.1 Parallel Algorithm

Looking at Algorithm 1, the generation of asset bindings – the asset-bound preconditions and postconditions associated with each exploit – does not encumber any loop-carried dependencies. This process entails ranging over each exploit, while effectively creating all permutations with repeats of assets bound to exploit parameters. Thus, this loop can be parallelized, processed as independent tasks distributed to nodes under a distributed or a shared memory model.

The **while** loop can be parallelized as well. However, it does depend on bindings, each of which are generated beforehand. Each unexplored network state can be processed independently of the others. One issue that must be addressed is the potential creation of duplicate *new_states* from two or more *network_states* that are being expanded in parallel. Message passing must also be kept to a minimum as constantly updating states (and searching for duplicates) could be a computationally expensive task.

The **for** loop at line 7, which ranges over exploits, is another candidate for applying a shared memory model for parallel programming. Here, each node manages network state expansion, wherein threads under their supervision would check an assigned set of exploits. The **for** loop at line 2 of the match algorithm, which compares bound preconditions and processes postconditions, is also a candidate for parallelism under a shared memory model. To simplify matters, we choose the latter for engaging threads in this way.

Algorithm 3 Parallel Attack Graph Generation

```
1:  procedure PARALLEL_AG_GEN(assets, exploits)
2:      unexpanded ← initial_network_state
3:      parallel for exploit e in exploits do
4:          bindings_e ← permute_r(assets, e.params)
5:      parallel while not done
6:          network_state ← dequeue(unexpanded)
7:          for exploit e in exploits do
8:              new_states ← match(
                    bindings_e,
                    network_state)
9:              enqueue(unexpanded, new_states)
10:         move network_state to expanded
```

Algorithm 4 Parallel Match Function

```
1:  function MATCH(bindings, state)
2:      parallel for binding in bindings do
3:          new_state ← check(binding, exploit.preconds)
4:          if new_state then
5:              enqueue(new_states, new_state)
6:      return new_states
```

3.2 Complexity Analysis

An approach to deriving the complexity of the parallel algorithm begins with the serial algorithm and pursues a systematic transformation of it according the algorithmic transformations induced through parallel operations. Fundamentally, we scale the serial complexity down by a factor of either n or t, where n is the number of nodes and t is a number of threads per node, depending on the nature of the parallel transformation. We incorporate two terms designed to account for the overhead incurred by (i) communication between nodes in a distributed memory programming model (c_n) and (ii) explicit coordination of tasks between threads in a shared memory programming model (c_t). That is, the complexity of the each parallel algorithm element here can be generally characterized as $O(\frac{serial}{n} + c_n)$

time and $O(\frac{serial}{t} + c_t)$ time for the distributed and shared memory model components, respectively.

The loop in line 3 of *parallel_ag_gen* has a serial complexity of $O(EFA^P)$. Distributing the processing across n nodes results in a parallel time complexity of $O(\frac{EFA^P}{n} + c_n)$. Recall that the serial implementation of the match function runs in $O(\log FA^P)$. The parallel version of the match function has no loop-carried dependencies, and its complexity is $O(\frac{\log FA^P}{t} + c_t)$ time. As we did not parallelize the loop in line 8 of *parallel_ag_gen*, its time complexity is $O(E(\frac{\log FA^P}{t} + c_t))$.

The while loop explores every state in the ultimate state space S, but in a distributed fashion that partitions the set to be processed by n nodes. The parallel time complexity for lines 5-10 accordingly is $O(\frac{SE(\frac{\log FA^P}{t} + c_t)}{n} + c_n)$. Thus, *parallel_ag_gen* has a complexity of $O(\frac{SE(FA^P + c_t)}{n} + c_n)$.

3.3 Discussion and Future Work

Practical generation and analysis of attack graphs for large networks in realistic threat landscapes require techniques that conquer scalability challenges. Enterprise networks consist of thousands or millions of diverse and complex assets. The National Vulnerability Database (NVD) has approximately 75,000 unique CVE vulnerability descriptions [2].

Our approach to scalable attack graph generation exploits concurrency inherent in the serial algorithm. A similar approach, relying on a multi-agent platform, engages a distributed memory programming solution to generate attack graphs [5]. The result relies on an implementation strategy minimizing performance penalties introduced by communication and coordination between computing elements. Distributed memory models of parallelism use message passing to synchronize and coordinate program execution, but at a cost in node-to-node communication. The Message Passing Interface (MPI) is a low level programming library that implements the distributed memory model for parallel computing [1]. Shared memory programming models incur overhead in the explicit management and avoidance of race conditions and other synchronization hazards. OpenMP extends the C programming language to implement shared memory programming while handling locking and mutexes automatically [3]. The implementation we pursue blends MPI and OpenMP for a hybrid parallel computing solution.

Hardware is also influential in the implementation of a parallel computing solution. We plan to deploy our parallel attack graph generation software on a heterogeneous compute node cluster (under development), each of 12 nodes consisting of a CPU, two Many Integrated Core (MiC) processors, and a Field Programmable Gate Array (FPGA). Experimentation with configurations and deployment across the nodes' compute platforms will provide insights on how to exploit the differential capabilities of each node.

Finally, this effort sets the stage for a similar one in attack graph analysis. Several techniques have been proposed to reveal unsafe states, viable attack chains and critical paths in attack graphs. Where attack graphs contain millions of nodes, analysis will confront similar, if not more severe, scalability issues. Lessons learned in scalable attack graph generation must be explored and translated to attack graph analysis to fulfill the promise of using attack graphs to derive meaningful security intelligence for large networks.

4. CONCLUSIONS

Scalable attack graph generation is a challenge for large networks. State space explosion in the presence of high asset and exploit counts mandates an approach engaging parallelism and high performance computing. Opportunities exist to exploit concurrency in generation algorithms. A parallel algorithm that combines distributed and shared memory models offers improved time complexity characteristics.

5. ACKNOWLEDGEMENT

This material is based on work supported by the National Science Foundation under Grant No. 1524940. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

6. REFERENCES

[1] Message Passing Interface (MPI) Forum Home Page. http://mpi-forum.org/.

[2] National Vulnerability Database. http://nvd.nist.gov/.

[3] OpenMP.org. http://www.openmp.org/.

[4] 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, pages 217–224, New York, NY, USA, 2002. ACM.

[5] K. Kaynar and F. Sivrikaya. Distributed attack graph generation. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2015.

[6] M. E. Kuhl, J. Kistner, K. Costantini, and M. Sudit. Cyber Attack Modeling and Simulation for Network Security Analysis. In *Proceedings of the 39th Conference on Winter Simulation: 40 Years! The Best is Yet to Come*, WSC '07, pages 1180–1188, Piscataway, NJ, USA, 2007. IEEE Press.

[7] X. Ou, W. F. Boyer, and M. A. Mcqueen. A scalable approach to attack graph generation. In *In Proc. of the Conference on Computer and Communications Security (CCS). ACM*, 2006.

[8] C. Phillips and L. P. Swiler. A Graph-based System for Network-vulnerability Analysis. In *Proceedings of the 1998 Workshop on New Security Paradigms*, pages 71–79, New York, NY, USA, 1998. ACM.

[9] L. Pièvre-Cambacédès and M. Bouissou. Beyond Attack Trees: Dynamic Security Modeling with Boolean Logic Driven Markov Processes (BDMP). In *Dependable Computing Conference (EDCC), 2010 European*, pages 199–208, Apr. 2010.

[10] S. Roy, C. Ellis, S. Shiva, D. Dasgupta, V. Shandilya, and Q. Wu. A Survey of Game Theory as Applied to Network Security. In *2010 43rd Hawaii International Conference on System Sciences*, pages 1–10, Jan. 2010.

[11] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of 2002 IEEE Symposium on Security and Privacy*, pages 273–284. IEEE, 2002.

[12] T. Tidwell, R. Larson, K. Fitch, and J. Hale. Modeling internet attacks. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and security*, volume 59, 2001.