Parallel Model for Complex Attack Detection

Abstract

This investigation extends previous work on a Traffic-based Reasoning Intrusion Detection System using Ontology (TRIDSO). TRIDSO uses semantic expression to give reason to network traffic patterns to detect complex attacks. Although achieving complex attack detection the overhead of populating the application's knowledge base prevents it from real-time usage. Concurrent programming benefits include increased run-time efficiency. Two different parallel models were created in an attempt to reduce the application overhead. Solutions to issues in refactoring TRIDSO for concurrency are presented, including handling of volatile resources. Run-time and scalability analysis describing application throughput improvement is provided.

Introduction

Computer networks and the number of devices with connectivity to networks continue to grow. Access to network applications and information can increase productivity. Relying on network connectivity carries a risk. As the size of a network grows, the dependent devices, resources, and information become vulnerable to attack. The implementation of necessary network security protocols are based on the probability of the occurrence of a specific attack and the cost to recoup the damage caused by the attack.

Intrusion detection systems (IDS) are used to ensure the protection and availability of network resources. Amoroso and Kwampniewski [1] identified IDS requirements to be short time to attack detection, knowledge of attacks, profiling of system activity and multi-source information correlation. Snort [2] is an open source IDS which satisfies this selection criteria including the ability to quickly and accurately detect simple attacks. As simple attack detection become more efficient, complex attacks will become more prevalent. Continued research in complex attack detection is becoming more important.

Frye, Cheng and Heflin [3] developed a Traffic-based Reasoning Intrusion Detection System using Ontology (TRIDSO), which focused on detection of complex attacks. Detection of complex attacks requires the recognition of the combination of multiple simple attacks. TRIDSO uses semantic expression to reduce redundant matching of multiple simple attacks in complex attack recognition. This methodology was shown to detect complex attacks that Snort overlooked.

TRIDSO uses an ontology model developed by Frye, Cheng and Kaplan [4] to detect complex attacks. The ontology model was developed using Jena [5], a Java based framework used by the application as a knowledge base. The semantic web and linked data abilities are used to construct the detection hierarchy. As is common with other reason-based IDSs, TRIDSO suffers from a knowledge base population overhead. The reduced run-time efficiency restricts the application from deployment in a real-time system, but the complex attack detection results provide useful information for developing security protocols for network managers.

Today, many computer systems include multi-core processors that allow for multiple sets of instructions to be completed at the same time or concurrently. Concurrent programming practices developing applications where separate sets of instructions are executed simultaneously or in parallel. Concurrent programs run more efficiently on multi-processor systems as each sub-process can be run in parallel on a separate processor. Further, each processor may contain multiple cores, each of which is capable of executing its own set of instructions. The benefit of concurrency is parallel execution, which often leads to increased application throughput when compared with similar sequential applications.

KU USER 3/8/15 11:31 AM
Comment [1]: I am writing this comment last, after reading the document. I think at a very high, system level of description, before getting into the details, it would be useful to have some high level diagram that identifies the serial bottlenecks in the initial system and its successors. Take a look at the series of dataflow diagrams in http://faculty.kutztown.edu/parson/spring2015/csc543spring2015assn1.pdf , especially the first two. The comment card in the first, non-threaded, handout code discusses the two components in the first diagram that will be bottlenecks in a multithreaded variant of that handout code; the subsequent diagrams remove the bottlenecks. This assignment strictly used BlockingQueues. There is a long Comment below on possibly boosting performance using a nonblocking ConcurrentLinkedQueue. The current point, though, is to use some high level diagramming technique to help to identify the bottlenecks in the system architecture.

KU USER 3/8/15 10:29 AM
Deleted: Previous

KU USER 3/8/15 10:30 AM
Deleted: was completed

KU USER 3/8/15 10:31 AM
Deleted: ,

KU USER 3/8/15 10:32 AM
Comment [2]: Glenn Blank always hammered me with the requirement to change passive to active sentences in my PhD papers & dissertation. I walk in his technical writing shadow every time I review a paper. ☺

KU USER 3/8/15 10:31 AM
Deleted: was attained

Many of the risks associated with concurrency are based on contention for shared resources between processes. A concurrent application design should control access to shared or volatile resources between multiple processes to prevent an indeterminate outcome.

This paper addresses the scalability problem associated with the run-time efficiency of TRIDSO by adapting the application for concurrency. The goal is to reduce the overhead of populating the application's knowledge base to increase application throughput. The multi-core, multi-processor system where the concurrent models were tested is outlined. Volatile resource management and transaction classification are discussed. Design patterns of two parallel implementations are explained. Finally, a run-time and scalability analysis is given.

Related Work

Mehra [6] provides a comparison of two freely available open source IDSs named Snort [2] and BRO [7]. Both systems monitor network traffic by completing deep payload packet inspection. Snort uses string and regular expression matching against a rule set to determine malicious signatures. BRO uses application-level semantics and event pattern matching to detect attacks. Mehra explains that BRO has better network adaptability being designed for customization and experimentation. Although Snort is more commonly used due to its ease of deployment, it is not suitable for high speed networks.

Mirtra, Najjar, and Bhuyan [8] used advances in computer hardware to implement a faster version of Snort. Since Snort is open source, additional rules are continually added to the applications. As network speeds increase and packet volume grows, more computation cycles are required to step through the application's rule set. The hardware model offsets computational cost by handing off rule matching to dedicated hardware. Compiling each regular expression in the Snort rule set into hardware based non-deterministic finite state automata creates an improvement in run-time efficiency.

Although hardware solutions are achievable, scalability and portability are unresolved issues when adapting such a model to new systems. Xiang and Zhou [9, 10] suggest using multi-core, multi-processor systems to support packet processing in real-time at the application level. This approach is used to address the scalability of TRIDSO by implementing two parallel models for execution on a multi-core, multi-processor machine.

Sequential System Analysis

The sequential version of TRIDSO was analyzed to determine necessary adjustments to adapt the model for parallel execution. A UML sequence diagram [11] was useful in identifying the Jena ontology model as a volatile resource. This ontology model serves as the knowledge base for complex attack detection. Multiple components of the application use SPARQL [12] queries to commit updates by inserting instances into the knowledge base. The knowledge base is eventually queried to determine if a complex attack has occurred.

KU USER 3/8/15 10:38 AM
Comment [3]: Could we have the sequence diagram in this paper? You know how I love diagrams. I am itching for one at this point. It would really help me to visualize the software architecture at a high level.

Unfortunately, Jena does not support internal concurrent updates. This functionality could be useful as multiple classes commit potentially overlapping updates to the knowledge base. Instead of replacing Jena, the approach of implementing program control for updating the knowledge base was taken. The first step was to identify where and what processes were committing updates in order to synchronize their access. The sequential implementation uses a chain of responsibility design pattern [13] with static methods called from library classes. Four separate classes create multiple query strings for committing updates, which include *Traffic Streams*, *Packet Collections*, *Alert Attacks* and *Simple Attacks.* Multiple functions within each class generate query strings and commit updates through a common library function call. Each class creates

KU USER 3/8/15 10:39 AM
Deleted: h

KU USER 3/8/15 10:41 AM
Comment [4]: Again, diagramming the chain would help. Some readers won't know what Chain of Responsibility is, and won't want to go running for a reference to read. I'd show how COR applies to this problem, thereby giving them an intro to COR at the same time as illustrating this problem.

a layer of semantic knowledge used toward detecting a complex attack. Each class or layer is dependent on the preceding layer and therefore cannot be executed in parallel. While the instances inserted within a class share an association to the same layer of semantics, they do not retain a dependency upon each other. Therefore, the updates committed within a class can be executed in parallel whereas each class must be treated atomically.

The system used for testing in this research contained eight processors where each processor has eight cores. As threads are spawned in the application, each processor is assigned a thread of execution, which is run in one of the eight cores local to the processor. Each of the processors is assigned a single thread of execution and none of the processors is assigned a second thread until each of the processors is executing at least one.

To allow for parallel execution, classes were refactored to fit object oriented design patterns. Due to Jena not supporting internal concurrency, improvements for application throughput focused on reducing the overhead of generating the update query strings. The developed Symmetric model follows a builder design pattern [13] allowing the operating system to distribute the workload of generating query strings and committing updates across multiple cores and processors. The Asymmetric model follows a factory design pattern [13] with a producer-consumer model [14]. The Asymmetric model dedicates knowledge base updates to a single core on a single processor while distributing the workload of query string generation. The two parallel models were implemented to benchmark run-time improvements. The implementations of Symmetric and Asymmetric parallel models are described in the following two sections.

Symmetric Implementation

The Symmetric implementation allows the operating system to distribute the load over multiple cores and processors of a system. The implementation used to achieve this model uses an adapted builder design pattern shown in Figure 1. The pattern is applied to each of the classes responsible for generating queries and committing updates. The example presented is for the *TrafficStreams* class, although all four aforementioned classes follow the same pattern.

The instantiation of each class object is a bootstrap for adding the respective layer of semantics to the ontology. The sequential implementation used multiple methods, each of which generated a unique query string for populating the knowledge base. Here, when the initial object is constructed, multiple threads are generated. The initial class object is the runnable target of each thread, allowing the thread to inherit the knowledge base reference. Each thread generates a unique query string and commits an update by pairing an assigned thread id with a class method. Each of the threads is joined before the constructor returns. Thus, instantiation of each class object, corresponding to a layer of semantic knowledge, executes its internal updates concurrently while the class instructions are treated atomically. Each layer of semantics is then added by simply instantiating each class in sequential order to build the layering dependency needed for complex attack detection.

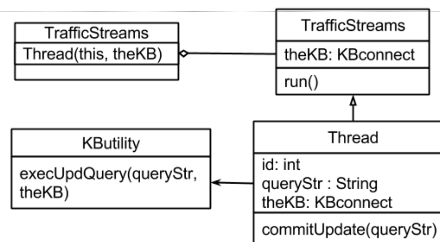

Figure 1: Adapted Builder Pattern

In the Symmetric model, the operating system is responsible for distributing the

KU USER 3/8/15 11:23 AM

Comment [5]: Again, unless the reader is versed in O-O design patterns, these sentences will be lost on them. Maybe the best approach would be to explain (with diagrams as appropriate) what is going on, and mention something to the effect that "this represents the factory design pattern []14]", rather than leading with the factory design pattern, which will be a mystery to many readers. (I do like the patterns myself, by the way. We immerse ourselves in them as part of csc520.) This comment applies to subsequent citation of design patterns with which the reader may not be familiar.

KU USER 3/8/15 10:51 AM

Comment [6]: It looks from the diagram that there are two distinct classes called TrafficStreams, with one of them containing an aggregation link to the other. Is that correct? It is confusing to see both with the same name. Also, I assume that Thread is not the same as java.lang.Thread. Is my assumption correct? That library class does not contain these fields and methods, so they must differ. Maybe it would help to spell out the full package.class path to disambiguate that, since anyone working regularly with Java concurrency will wonder about java.lang.Thread, the minute they see this diagram.

workload over the cores and processors of the system. This implementation addresses scalability by leveraging the operating system to distribute the work load as opposed to pigeon-holing work based on system hardware. This implementation is portable, although run-time improvements will be platform dependent. Parallel query string generation and instance insertion within each class should yield an increase in application throughput, although a lack of internal knowledge base concurrency may result in unwanted and unpredictable thread pre-emption resulting from contention for the knowledge base writing lock. These issues are discussed in the run-time analysis section.

Asymmetric Implementation

The Asymmetric implementation was developed to address two potential issues in the Symmetric implementation. One problem with the Symmetric design was assuming that generating a query string and committing an update was an atomic instruction. For example, although updates committed by *PacketCollections* cannot take place before all *TrafficStreams* updates are complete, the query strings for *PacketCollections* could be generated concurrently while *TrafficStreams* updates where being completed. Second, sharing the knowledge base across multiple threads could create additional overhead. Each time the knowledge base is updated, the reference must be shared to each thread containing a reference. Assuming that two processes responsible for committing an update are executing on cores of separate processors, the knowledge base must be written out to Java Main Memory. As the knowledge base grows, the cost of sharing this reference increases. The Asymmetric model addresses each of these design flaws.

By following a parallel producer/consumer model, a solution to both problems is achieved. First, a factory design pattern is applied to all four classes for generating query strings. A *QueryGenerator* interface is implemented as the query generation factory. Each class implements the *QueryGenerator* interface, becoming a specialized factory for generating query strings specific for adding their respective layer of semantics. Each class is instantiated with a reference to a blocking queue specific to the class. Each class generates unique query strings in parallel and inserts them into the blocking queue. The factories become the producers in the producer/consumer model and all four classes are executed in parallel as the query strings do not have dependency upon layers of semantic knowledge. The knowledge base is retained in the main thread of execution and becomes the consumer in the model. The main thread is responsible for committing updates with query strings removed from the blocking queue associated with each class. A blocking queue is used to ensure the consumer does not move to the next queue before all queries associated with the class have been consumed. Each queue is consumed fully before moving to the next queue to preserve the semantic layering of the knowledge base.

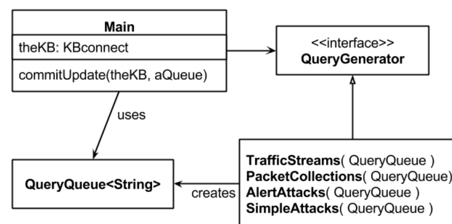

Figure 2: Asymmetric Factory Design Pattern

In this design, the knowledge base follows a Singleton design pattern [13] by restricting the knowledge base to the main thread of execution. This allows possible local caching of the knowledge base and reduces the potential of multiple writes to Java Main Memory. This concept parallels an Asymmetric model as updates to the knowledge base will be limited to one process, which will execute on one core of one processor assuming the thread

KU USER 3/8/15 11:55 AM
Comment [7]: I don't know whether the run-time dynamics of this system lend themselves to the following approach, but Dylan & I applied the following to a search problem that had been using BlockingQueue, and we are applying them again this semester in CSC543 to a parallel state machine (a cellular automaton, triggered partially by my advising of Dan Rassler's senior seminar project last semester, including some of my ideas that he never got around to implementing). The idea is this: If the worker threads that are reading the queues typically find data in the queues during the work cycle, then use a non-blocking ConcurrentLinkedQueue that the threads poll() to get their next work item. If the threads have non-busy times, have them block somewhere else (we are using a CyclicBarrier in the current CSC543 sequence of projects) until the next round of work. Also, in both Dylan/my case and this semester, I have been able to get another doubling of performance by giving each worker thread its own nonblocking ConcurrentLinkedQueue from which to poll() work. This eliminates contention among the worker threads for access to a shared ConcurrentLinkedQueue. See Graph 4 in http://faculty.kutztown.edu/parson/pubs/BidSearchPDPTA2012ParsonSchwesinger.pdf . Just going to multiple nonblocking queues cuts time in half from the single nonblockingqueue approach. Also, Graph 3 compares blocking queue to a single ConcurrentLinkedQueue. The dotted "minimum blocking" in Graphs 3 & 4 is the same curve; it uses a single ConcurrentLinkedQueue that is poll()ed for work. The "blocking queue" uses LinkedBlockingQueue and blocks for work, but is otherwise the same algorithm. The "multiple queues" in Graph 4 is the same as "minimum blocking" except that its uses one ConcurrentLinkedQueue for each worker thread. It is actually engineered to minimize thread contention at each side of each ConcurrentLinkedQueue. The writes into each ConcurrentLinkedQueue happen in round-robin order, controlled by an AtomicInteger index that we increment using (+1) % numberofQueues, for two reasons: in avoid thread contention for a ConcurrentLinkedQueue in enqueuing, and to distribute the workload evenly. The reads from each ConcurrentLinkedQueue take place only by the since worker thread feeding from that queue in the "multiple queues" approach, so there is absolutely 0 contention for access to a given ConcurrentLinkedQueue object. See also the activity diagram on page 7 of http://faculty.kutztown.edu/parson/spring2015/csc543spring2015assn2.pdf . It shows busy poll()ing of a queue during busy work times (queue has a high probability of having a work item when polled). I needed to outline the algorithm without giving it away; pseudocode would have been too close to the Java solution, so I used an activity diagram. H ... [1]

KU USER 3/8/15 11:18 AM
Comment [8]: If this is a serial bottleneck, and if it is in your code, we should look at parallelizing it. It is not necessarily a bottleneck, though. And, it may not be parallelizable.

is not pre-empted. In the Symmetric model, generating queries for the last three layers of semantics are dependent on successful insertion of instances from the first layer of semantics. In

The Asymmetric model queries for all four classes are generated in parallel. The removed dependence should yield an increase in run-time efficiency for the Asymmetric model.

Run-time Analysis

A series of test runs for each of the three execution models was planned to benchmark gains in run-time efficiency. The same dataset containing a PingScan network attack was used for each test. Four test runs where completed for each model over two separate days with one test run starting in the morning and one starting in the evening. The application was allowed to complete running before a new test run would begin and was the only application running on the test machine. A limit of two test runs per day was used to ensure that exceedingly long run-times could complete before a new test began. Also, test runs on different days where added to determine if latency in accessing accompanying project files located in the network file system was effecting run-time efficiency.

Table 1 shows the run-time results from Sequential, Symmetric and Asymmetric models. The first four rows of the table show the application run-time for each of the four test runs completed for each model. The fifth row shows the average run-time for each model with the two columns at the far right displaying the percent increase in run-time achieved. The last two rows show the differences in the fastest and slowest run-times.

The results show that both the Symmetric and Asymmetric models increase application throughput by 12.88 percent and 34.97 percent respectfully. Percentage gains for the Symmetric and Asymmetric are calculated respective to the Sequential implementation. The Asymmetric model had a 34.97 percent gain on average and a 21.38 percent gain during the fastest observed test runs. The gain for each model is attributed to the reduced overhead by generating query strings in parallel.

	Run-Time			Percent Increase	
	Sequential	Symmetric	Asymmetric	Symmetric	Asymmetric
AM day1	9:10	2:20	3:40	-----	-----
PM day1	5:16	2:55	4:48	-----	-----
AM day2	4:55	8:10	1:54	-----	-----
PM day2	2:25	5:30	3:49	-----	-----
Overall Average	5:26	4:44	3:32	12.88%	34.97%
Fastest run-time	2:25	2:20	1:54	3.45%	21.38%
Slowest run-time	9:10	8:10	4:48	10.91%	47.64%

Table 1. Benchmark results with time in hours

The Asymmetric model is a more scalable concurrent application than the Symmetric model for several reasons. First, this model has a greater increase in application throughput attributed to constraining the knowledge base to one thread of execution. By localizing the knowledge base, the overhead of writing its reference to Java Main Memory after each update is eliminated. Now, the knowledge base can be cached locally and only needs to be rewritten when its size exceeds its local cache.

Run-time analysis shows separating query string generation and knowledge base

KU USER 3/8/15 11:18 AM
Deleted: t

KU USER 3/8/15 11:19 AM
Deleted: ,

KU USER 3/8/15 11:20 AM
Comment [9]: Line graphs similar to those in http://faculty.kutztown.edu/parson/pubs/BidSearchPDPTA2012ParsonSchwesinger.pdf would be nice ☺

KU USER 3/8/15 11:21 AM
Deleted: determined to be

updating with a factory design pattern reduced application overhead from the Symmetric to Asymmetric implementations. The outlying slowest run-time observed is nearly half the slowest run-time of the Sequential and Symmetric models.

Another result of testing shows a large amount of variance application run-time. It is unclear whether these outlying results stem from latency in accessing the network file system for the ontology definitions or pre-emption caused by the order of committed updates. The calculated variance for the Sequential and Symmetric models combined is nearly 7.5 hours when compared to the Asymmetric model variance of nearly 1.5 hours. The reduced variance is important for a scalable application as a predictable run-time is necessary for network managers to be able to efficiently use the application. Although a significant reduction in application throughput was achieved, the improvements have not been great enough to allow the application to detect complex attacks in real-time.

Conclusion

Two parallel implementations were developed to address the scalability of a sequential version of a Traffic-based Reasoning Intrusion Detection System using Ontology. The specific goal of the development was to increase application throughput such that the application could be deployed for real-time attack detection. Reducing application overhead for populating the knowledge base resulted in increased application throughput of up to 35 percent. Although this is a significant improvement, more research is needed to further reduce the application's run-time to be deployed for real-time complex attack detection.

Although the database used for the ontology model did not support internal concurrency, two areas could be addressed to seek further improvements. The first is possible restructuring of the update query strings to create greater parallelism. The second is to implement an update scheduler based on the pre-emption duration caused by each update. We believe that these options could further reduce overhead and created better run-time predictability resulting in a more scalable application.

A second proposal for continued research is to use an alternate semantic model to represent the attack definitions currently defined for the ontology. Replacing Jena as the database for the current ontology model with a database that supports internal concurrency could mitigate the query restructuring and scheduler implementation previously described.

Computer networks will continue to provide availability and access to information. As networks continue to develop, vulnerabilities in these networks will be exposed. Continued research in both simple and complex attack detection and prevention is needed. Finally, continuing development of scalable systems capable of detecting zero day attacks is needed to prevent information loss.

References

[1] E. Amoroso and R. Kwampniewski. Selection Criteria for Intrusion Detection Systems. 14th Annual Computer Security Applications Conference (ACSAC '98)**.** December 10, 1998.

[2] SNORT. http://www.snort.org/. Accessed November 15, 2013.

[3] L. Frye, L. Cheng, and J. Heflin. 2013. TRIDSO: Traffic-based Reasoning Intrusion Detection System using Ontology. Special Collection on Ontologies and Semantics in Communication Systems and Networks of the Journal of Research and Practice in Information Technology (JRPIT), 44 (4).

[4] L. Frye, L. Cheng, and R. Kaplan. *A Methodology to Identify Complex Network*

Attacks. The 2011 International Conference on Security and Management (SAM'11) at The 2011 World Congress in Computer Science, Computer Engineering, and Applied Computing (WORLDCOMP'11). Las Vegas, NV. July 18-21, 2011.

[5] JENA. http:// jena.apache.org/. Accessed November 15, 2013.

[6] Mehra, P. A Brief Study and Comparison of Snort and Bro Open Source Network Intrusion Detection Systems. International Journal of Advanced Research in Computer and Communication Engineering. 6(1), 2012, pp. 383-386.

[7] BRO. http://www.bro.org/. Accessed November 15, 2013.

[8] Mitra, A., Najjar, W., Bhuyan, L. Compiling PCRE to FPGA for acceleration SNORT IDS. Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems. (2007), pp. 127-136.

[9] Xiang, Y., Zhou, W. Using Multi-core Processors to Support Network Security Applications. 12th IEEE International Workshop on Future Trends of Distributed Computing Systems. (2008) pp. 213-218.

[10] Tian, D. Xiang, Y. A Multi-core Supported Intrusion Detection System. IFIP International Conference on Network and Parallel Computing. (2008), 50-55.

[11] Fowler, Martin. (2004). *UML Distilled.* Westford, MA: Addison-Wesley. 53-61.

[12] SPARQL QUERY LANGUAGE FOR RDF. http://www.w3.org/TR/rdf-sparql-query/. Accessed December 8, 2013.

[13] Object Oriented Design. http://www.oodesign.com/. Accessed December 8, 2013.

[14] Producer-consumer problem. http://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem/. Accessed December 8, 2013.