Betweeness Centraility performance…

Observations using Cray Graph Engine and Apache GraphX on computer network data

Eric Dull, Felix Flath, Brian Sacash, John Zachary
Cyber Risk Services
Deloitte and Touche, LLP
Arlington, VA
{edull, fflath, bsacash, jzachary}@deloitte.com

***Abstract*— Graph analytics are useful for overcoming real-world analytic challenges such as detecting cyber threats. Our Urika-GX system is configured to use both the Cray Graph Engine (CGE) and Apache Spark for developing and executing hybrid workflows utilizing both Spark and Graph analytic engines. Spark allows us to quickly process data, stored in HDFS, powering flexible analytics in addition to graph analytics for cyber threat detection. Apache Spark's GraphX library offers an alternative graph engine to CGE. In this paper, we will compare the available algorithms, challenges, and performance of both CGE and GraphX engines in the context of a real-world client use case utilizing 40 billion RDF triples. *(Abstract)*

Keywords: Graph engines, Cray Graph Engine, Apache GraphX, computer network security, BRO, Betweenness centrality, Urika-GX

I. Introduction

Detecting intruders inside of a computer network requires a variety of security tools and analytic techniques used together along lines of analysis thought. Signature-based security tools detect previously-identified malicious files, websites and network traffic. These tools cannot identify currently-unknown malicious files, websites, and network traffic which are used by adversaries to vary their tactics, techniques, and procedures. Detecting these variations in files, websites, and network traffic used by adversaries is enabled by the use of behavioral analytic data science tools and engines with data science techniques, statistic methods and models of intruder behaviors. Once security professionals detect anomalous or suspicious files, websites, and network traffic that could be in use by intruders to infiltrate and exploit a target network, they perform manual analysis to validate that the identified files, websites, or network traffic are malicious. Security professionals then use streaming analytic engines and signature-based security tools to detect the validated files, websites, or network traffic going forwards.

There are a number of analytic engines that can be used in performing these behavioral analytic, including interactive, batch, and streaming engines. Each engine is architecturally suited to perform a different type of analysis. These engines can be used together in a "hybrid architecture" to execute different parts of a larger data science workflow. Using "hybrid architectures" allow multi-step analytic workflows to be more efficiently executed over more data in less time, enabling security professionals to identify more-sophisticated behaviors potentially used by more sophisticated intruders that are likely to cause greater damage to the target network, its contents, or its owners.

Graph engines are one of the engine types available to security professionals performing behavioral analysis. Graph engines have been implemented using different computational architectures, including supercomputing shared-system image and "cloud" or distributed cluster. Each of these implementations has different performance characteristics when using different types of graph algorithms on various sizes and types of graphs.

Graph algorithms are applicable to computer network data analysis[1] and identifying behaviors present in the computer networks. One of these algorithms, Betweenness Centrality[2], identifies the central nodes in a network component. Central nodes are the nodes that have the most paths passing through them. These central nodes are the ones that most "connect" the component together. Identifying these central nodes enables the identification of key files, websites, or network traffic nodes that correspond a number of intruder behavioral models.

This paper outlines graph analysis, its suitability to computer network data analysis, the expected models of intruder behavior, a large computer network and its data, betweenness centrality as an applicable algorithm, its performance on Cray Graph Engine and Apache GraphX on a Cray Urika-GX, and the validated security outcomes enabled by this analysis.

II. Background

This section outlines graph analysis, intruder behavior models, betweenness centrality, and the potential roles of graph nodes identified by betweenness centrality in intruder behavior models.

A. Graph analysis

Graph theory supports advanced analytics that uncover insights within complex networks, especially in the domain of computer and network security. Connectivity, centrality, and structural similarity provide deep insight into network traffic behavior that other analytical methods cannot provide. The standard representation is nodes represent network

entities and edges represent a semantic networking relationship between hosts. This topology makes it easy to represent pairwise relationships in the network. Further analysis can lead to uncovering insights about relationships through analytical techniques.

Graph processing is a common computational framework for modeling and analyzing complex relationships. Recent advances in social media, inexpensive storage, distributed data processing, and cloud computing created a demand for graph processing frameworks that scale to extremely large graphs with more than 10^6 nodes. The Facebook social graph reportedly is over an order of magnitude larger [3].

Several efforts produced software libraries, tools, and databases that promise to provide graph processing at scale. Some notable examples are Sandia's MTGL[4], Apache Giraph, Google's Pregel, Apache Spark's GraphX, Titan, and Neo4j. Concurrently, high performance computing companies and organizations developed parallel systems to solve highly complex scientific problems in intelligence analysis, physics, and genomics.

B. The Cyber Kill-Chain

Network behavioral analysis looks for indicators of network intrusion, malware, scanning, and other malicious or anomalous events. A human in the loop identifies these behaviors through analysis of network traffic indicators. Behavioral analysis differs from signature-based analysis because it supports detection of previously unidentified traffic patterns and behaviors associated with anomalous or malicious activity.

Sophisticated cyberattacks are analyzed in the context of a framework known as the cyber kill chain [5]. The cyber kill chain segments attacks into distinct stages that occur in sequence:

1. Reconnaissance. The attacker researches and identifies a target. Activities in this phase may include mapping a network or conducting research on social media platforms.
2. Weaponization. The attacker chooses a remote access exploit based on the target and packages it into a deliverable payload. An example may be an executable packaged within a PDF file to exploit an Adobe Reader vulnerability.
3. Delivery. Payload is transmitted to the target, commonly through an email attachment or website download.
4. Exploitation. After the payload is delivered, the attacker's code is executed. This may be actively triggered by the user or auto-executed by the operating system.
5. Installation. The remote access payload is installed onto the target machine to allow the attacker to maintain persistence in the environment.
6. Command and Control(C2). Compromised hosts beacon outbound to a controlling server with the ultimate objective of giving the attacker remote control of activities inside the target environment.
7. Actions on Objectives. After completing the above six steps, the attacker can begin to execute on their ultimate objectives. This may be data exfiltration or compromise of further systems in the network.

Each stage has a set of indicators that identify attempted or active intrusion. Some examples are:

- An attacker performing reconnaissance on a target network is identified by multiple IP addresses on the network experiencing attempted connections across large numbers of ports from the same originating IP address.
- An email containing the same hyperlink sent to hundreds of individuals inside the organization may be phishing attempts.
- A machine initiating small network connections to a certain host on a set frequency may be evidence of beaconing to a C2 server.

Monitoring network traffic for evidence of behaviors that correspond to various stages of the cyber kill chain help identify indicators of new infections that have not been identified by signature-based security tools.

C. Identifying lateral movement within a network

One of these behaviors that can help identify indicators of intruders at the "Actions on Objectives" step in the cyber kill-chain is the connection of otherwise disparate groups of nodes that communicate with each other.

Application users tend to be grouped together in communities by the nature of the server or data housed on that server. Users are segregated to specific servers based on a need to access the data housed on that server. Security principles including least-privilege access and separation of duties underlie this segregation. This segregation is visible in network traffic as groups of clients that are the clients that connect to certain servers. Network traffic should show a number of distinct, separate graphs for a given set of applications or services used within a network. These networks would not be connected. Also, it is more likely that a high-value or mission-critical system will be accessible only to trusted internal clients. High-value or mission-critical servers would therefore not be directly accessible by clients external to the network.

An intruder's pattern for compromising an internal high-value system is (a) infect an Internet-facing host that likely has far fewer security restraints (such as a workstation or web server) and (b) use the infected host to move laterally across the network toward a higher-value target. The intruder's pattern of behavior would likely violate the expected behaviors of the hosts on the network that it has subverted and is using surreptitiously. This violation of expected behaviors is visible from the network, and will be seen as a node connecting otherwise disparate networks.

It can be difficult to identify hosts violating these expected behaviors without knowing the roles of servers and services used on the network and the clients expected to connect to

those servers. A method for detecting clients violating these expectations and connecting otherwise disparate servers can use routing-based graph analytics, such as the all-pair shortest path algorithm and betweenness centrality.

D. Betweenness Centrality

Betweenness centrality is a measure that ranks the number of paths between any two vertices in a graph, S and T, that pass through a third vertex in a graph, V. This evaluation of centrality calculates the shortest paths between all pairs of vertices within the graph, traditionally using the Floyd-Warshall algorithm. Betweenness centrality commonly has a computational complexity of O(|V|3). On sparse graphs, the number of edges form a significant element of the computational complexity[6]. Betweenness centrality assumes graphs are non-directional.

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$$

Figure 1: Betweenness centrality calculation

Figure 1 describes the betweenness centrality algorithm. In this equation, V is the vertex under consideration, S and T are vertices in the network, σ_{st} is the number of shortest paths between S and T, and $\sigma_{st}(v)$ is the number of shortest paths that pass through V.

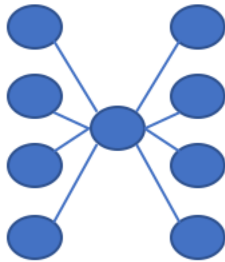

Figure 2: a hub-and-spoke graph

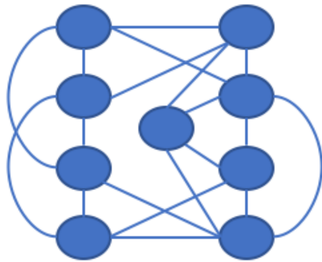

Figure 3: a relaxed clique, where every node is connected to 4 other nodes

Figure 2 shows a 9-vertex graph where the middle node is connected to every other node, and every other node is only connected to the middle node. The middle node's centrality score will be the highest in the graph. If the middle node is removed, the graph would cease to exist; no two nodes would be connected to each other. Figure 2 shows a behavior pattern seen in computer networks as a client and server model. The middle vertex is a server, and edge nodes are clients, using an application like a database or Secure Shell.

Figure 3 shows a 9-vertex graph where every node is connected to 4 other nodes. In this graph, every node would have the same centrality score. If any one node was removed, the graph would remain a graph; ever remaining node would still be connected other nodes in the graph. Figure 3 shows a behavior pattern seen in computer networks as a peer-to-peer model. Every node in the network acts as both a client and a server. These behaviors are seen in peer-to-peer routing protocols such as a Distributed Hash Table[9].

Figure 2 is more representative of common behaviors seen on computer networks than Figure 3. Human behavior tends to form scale-free graphs, and these behaviors have been confirmed in computer network graphs. This means that betweenness centrality scores from computer network graphs will be more varied than those from graphs similar to those in Figure 3.

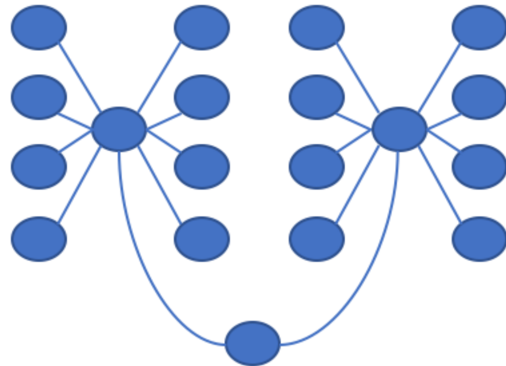

Figure 4: a candidate graph

Figure 4 shows two hub and spoke graphs connected by a vertex at the bottom of the figure. This bottom vertex would have the highest centrality of any vertex in the graph; higher than the hubs also present in the graph. This bottom node is representative of a client connecting to two servers that are otherwise disconnected.

This client could be connecting to these two servers for a variety of reasons. The client is shared between two people who occupy different roles in the organization. The client is used by a single person who switches job roles during the period of observation of network traffic from which the graph was built. The client is scanning for servers or applications on a network. The client is being used by an intruder to log into a server not ordinarily interacted with by the client's expected user. Two of these candidate behaviors are expected and benign on a network; the other two are suspicious and potentially malicious. Human analysis is required to determine which behavior is present within a graph when betweenness centrality identifies central nodes such as that shown in Figure 4.

III. ENVIRONMENT

This section describes the operational environment that is the basis for this study. This description includes the network monitoring technology used to monitor the network, the network under analysis, a description of Deloitte's Cray Urika-GX Big Data Appliance, and a discussion of the graph engines in use by the Apache GraphX and Cray Graph Engine.

A. BRO

Bro[1] is an open-source passive network security monitor for packet capture, metadata extraction, sessionization, and human friendly output files. Bro's open source nature allows security professionals to customize it to meet their own needs, including writing custom analyzers. Bro contains 70+ network protocol analyzers out of the box, including analyzers for web, email, and DNS traffic. We deploy Bro 2.4 on commercially-available R-Scope PACE BRO sensors developed by Reservoir Labs, which are capable of ingesting network packets up to 20 Gbps and producing a variety of metadata types that describe network connection, metadata, and application details from the OSI network stack layers 3-7. Bro generates these records for all packets that it observes; Reservoir Lab's R-Scope PACE sensors are powerful enough to observe all packets up to 20 Gbps and generate a variety of log types.

This study utilizes the connection log type (referred to hereafter as *conn.log*) to extract and represent our network graphs. The conn.log analyzer contains basic information on the session, such as originating and responding hosts, ports, bytes, and packets exchanged.[2] Other common Bro log file types are *http.log* (e.g. target domain, URI, and user agent string), *ssl.log* (e.g. certificate and encryption scheme), and *dns.log* (e.g. TTL and requested domain).

B. Network under discussion

The network generating the traffic analyzed in this paper is a very large network with over 500,000 endpoints generating traffic from dozens of physical locations through four Internet gateways. BRO network sensors at each of these four gateways monitor all outbound traffic leaving the network and all inbound traffic that has passed through the firewall rules and generate a record of all traffic successfully entering, successfully leaving, or attempting to leave the network This network contains a number of simultaneous behaviors including Windows, MacOS, Linux, and mobile operating systems and desktops, laptops, tablets, other mobile devices, servers, embedded devices, and special purpose hardware.

Figure 5, below, show the number of collected records each month.

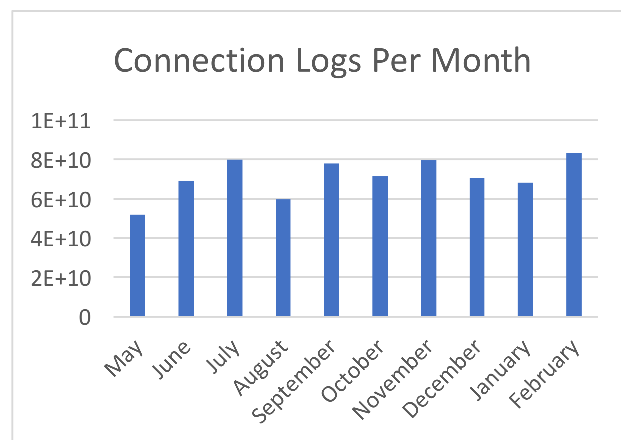

Figure 5: connection logs collected each month

This network is large enough and is comprised of a wide variety of endpoint types running a variety of operating systems and applications to generate representative traffic and analytic results that can be applied to other networks.

C. Cray Urika-GX

Our data analysis is performed on a Cray Urika-GX Big Data Appliance. This platform supports large-scale memory-intensive workloads across a cluster of 25 compute nodes each with 32 cores of Intel processors, 256 GB of RAM, and a shared file system. These compute nodes are connected by the Cray Aries network interconnect.

The Urika-GX platform supports common open source, big data technologies such as Apache Hadoop, Apache Spark, Apache Mesos, Lustre, and Python. The Hadoop File System (HDFS) distributes shares files across Urika-GX's blades to make data available to Spark. The platform includes Cray Graphics Engine for graph analytics. The underlying operating system is Linux CentOS.

D. Apache Spark and GraphX

Apache Spark is an open source, distributed data processing framework that supports interactive queries, streaming data, machine learning, and graph processing. Spark extends MapReduce by generalizing in-memory operations, transformations, and aggregations in a directed acyclic graph representation. Spark's primary programming environment is Scala with strong data science support for Python and R. This study focuses exclusively on Spark's graph processing library, GraphX.

GraphX utilizes the inherent scalability of Spark to analyze very large graphs across a compute cluster. It integrates seamlessly into an analytic workflow, from ETL to graph theoretic algorithms to analysis. A fundamental limitation to GraphX is memory available to store a graph for processing and analysis. Kabiljo et al found performance and

[1] https://www.bro.org

[2] Originating and responding hosts correspond generally to source and destination hosts.

scalability issues with GraphX on extremely large graphs, as well [4].

Our Urika-GX installation uses Spark 1.5. Since Spark is built upon the Hadoop File System (HDFS), we treat each blade as a Spark worker-node with distributed access to our data set. Up to 6.4 TB of RAM is available for parallel processing. GraphX includes a few graph functions in its distribution: PageRank, Triangle Counting, and Connected Components.

E. Cray Graph Engine (CGE) and SPARQL

CGE enables performing advanced analytics on the largest and most complex graph problems, and features highly optimized support for inference, deep graph analysis, and pattern-based queries. CGE is a highly-optimized software application for searching and querying very large, graph-oriented databases based on the industry-standard RDF graph data format and the SPARQL graph query language. CGE was designed from the ground up to run on Cray high performance platforms, such as the Urika-GX.

CGE is comprised of a front-end query engine and a back-end computational engine written in C++. The front-end engine parses and RDF description of a graph, loads into in in-memory representation, and evaluates SPARQL queries against the memory-resident database.

CGE ships with a library of built-in graph functions: BadRank, Betweenness Centrality, PageRank, Label Propagation, S-T Connectivity, S-T Set Connectivity, Triangle Counting, Triangle Finding, and Vertex Triangle Counting [10].

IV. METHODOLOGY

This section describes the experimental approach and method, the graph preparation for both CGE and GraphX, and a discussion of the composition of the generated network

A. Approach

As discussed in II.C and II.D, this paper is focused on conducting an experiment computing betweenness centrality on a number of graphs built from traffic generated by the network described in III.B. The experiment changes a number of variables to determine the performance characteristics of CGE and Apache GraphX while executing betweenness centrality. These variables include the size of graph under analysis and the number of Urika-GX compute nodes used to calculate betweenness centrality over that graph.

B. Method

As discussed in III.D and III.E, CGE ships with an implementation of betweenness centrality and Apache GraphX does not ship with betweenness centrality. Therefore, to conduct the experiment detailed in IV.A, we needed to implement betweenness centrality on Apache GraphX. This implementation was not straightforward, and a majority of our time was spent tailoring a version that functioned in the Spark paradigm. Our experiences are shared by others. Implementing betweenness centrality within the Apache GraphX framework to execute on Apache Spark clusters has been a documented challenge [8].

C. Data Preparation

Bro conn.log entries are records of network behavior that must be processed into a graph before betweenness centrality can be computed over it. CGE and Apache GraphX require different input formats and therefore require different data preparation workflows.

Figure 5 shows the CGE RDF graph data preparation workflow that was implemented in Scala scripts. At the top, a number of Bro conn.log entries are represented. Uid1 represents the BRO uid for the record, ts1 represents the timestamp for the record, orig represents the originating IP address for the record, and resp represents the responding IP address for the record. The Bro conn.log records contain more field which are omitted for clarity.

These conn.log records are first converted to RDF, which is represented in the next portion of Figure 5. The RDF representation shows how the three notional Bro conn.log records would appear in an RDF representation.

CGE uses SPARQL CONSTRUCT statements to induce subgraphs over which native CGE algorithms are executed. The last two portions of Figure 5 show how that CONSTRUCT query would convert the original RDF representation into a more-sparse representation, one where the details of the distinct connections between orig and resp are summarized into a single record. This conversion reduces the theoretical computational complexity of the betweennness centrality implementation executing across the summarized graph. Additionally, betweenness centrality is designed to execute against undirected graphs, and SPARQL CONSTRUCT statements induce directed graphs. Initial experimental executions on CGE produced meaningless results. Discussions with Cray indicated that the appropriate workaround would be to insert all edges into the graph twice; once in the original direction, and once in the reverse direction. CGE results from graphs using this workaround produced meaningful results, however it would also impact the theoretical computational complexity by doubling the number of paths considered during algorithm execution.

Apache GraphX requires a different data preparation workflow which was also implemented in Scala scripts. The betweenness centrality implementation that was used in this experiment requires to be defined by a node data frame and an edge data frame. The first contains the information pertaining to the nodes with associated look-up keys. The second contains two look-up keys, one for each connected node and edge data. The data preparation workflow takes Bro conn.log entries, which are contained in a different data frame and converts it into these two required data frames.

The data preparation workflows contain a number of conversion steps detailed above which require time to execute. This time is not representative of the algorithm execution, and that time is not included in this experiment.

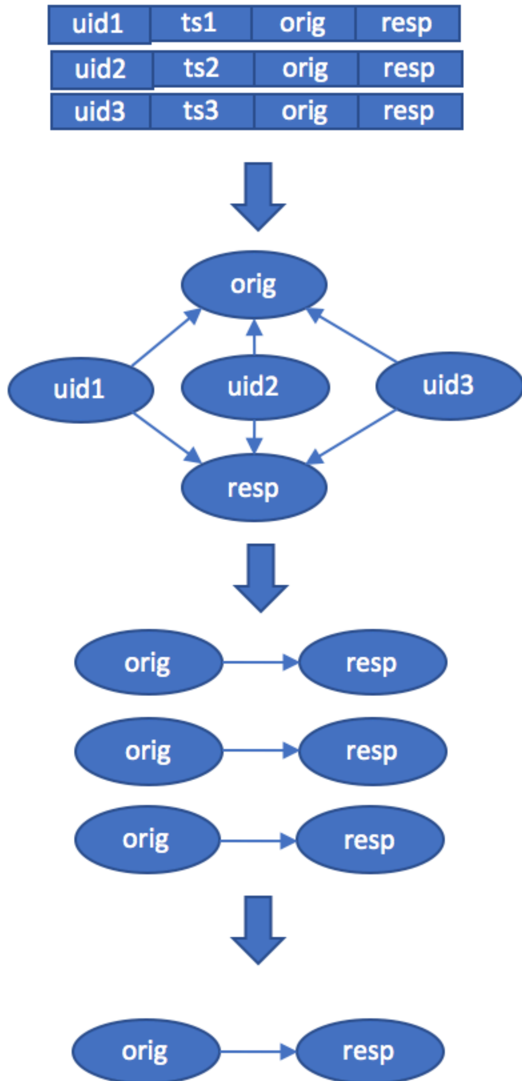

Figure 5: CGE data preparation process

D. Graph composition

The graphs used in this experiment are comprised of nodes which represent IP addresses and edges which represent TCP/IP traffic between the IP addresses. These graphs were generated from a number of days of traffic from the network discussed in III.B.

The network traffic that was used to build these graphs were restricted to successful connections on TCP ports 20-23, 123, 445, and 3389. These restrictions were used to focus the induced graphs on behaviors of interest to security professionals. Successful TCP connections are those that have passed through the three-way TCP handshake[7]. These connections are then ready to pass content that is accessible to the operating system and hosted applications listening on that port. These successful connections represent approximately 55% of the total connections. This restriction serves to reduce the noise present in the induced graphs.

Restricting network traffic to TCP ports 20-23, 123, 445, 3306, and 3389 serves to focus network traffic to applications that are known to bear interactive traffic and are commonly used for interacting with servers over the network. These ports represent a small portion of the traffic present on the network; the vast majority of the network traffic present on the network use TCP ports 80 and 443, which commonly carry web traffic and is not interactive in nature. This restriction also serves to reduce the noise present in the induced graphs by removing expected client-server web traffic.

Using these filtering criteria identifies a set of network traffic that can be used to generate a graph. This traffic was then subject to two variations to build graphs of different sizes to illustrate performance differences between CGE and Apache GraphX. These variations are the number of days of network traffic and the number of records from a set of days.

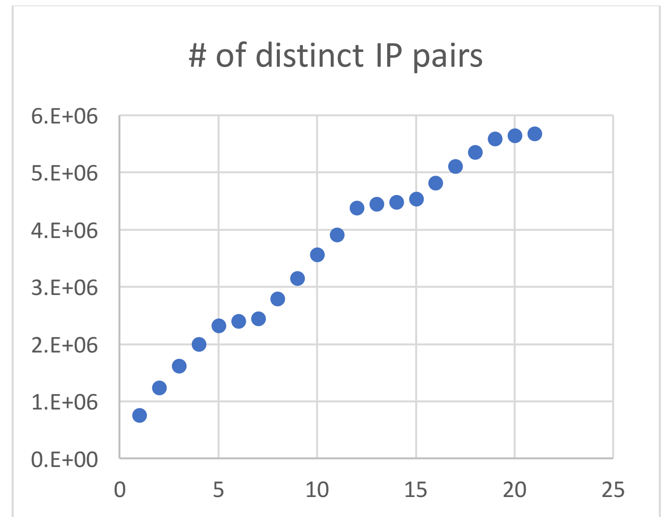

Figure 6: total edge count per day

Figure 6 shows the total number of unique graph edges per day for 21 days from February 2017. The unique edge count is expected to grow as the clients and servers on the network are active and observed by the BRO sensors. Once the clients connect to their servers, usually over a time period of several weeks, the total number of unique graph edges should level off. As Figure 6 does not fully level off, it suggests that the client and server interactions have not been fully-sampled in the 21 days of data used to construct graphs for these experiments. The flattening of slop in days 15-19 compared to days 1-5 suggests that sampling may be happening. The flat periods present at days 5, 12, and 19 correspond with weekends during which the network has fewer active users.

The network traffic from February 2017 was chosen for this paper due to the validated scanning activity present in the network during that time range. This data set contained representative graph structure to exercise CGE and Apache GraphX and show performance difference without occupying Apache Urika-GX compute nodes needed by security professionals.

TABLE I. GRAPH SIZES

Vertices	Edges
5,419	11,726
15,359	52,042
42,687	125,564
66,955	369,720
115,276	1,281,918

The fourth restriction is reducing the number of input records from a given date range. Table I outlines the graphs generated from 100,000 and 10,000,000 transactional records from a twenty one day range in February 2017. These records were selected from the set of available transactions in time order, operating functionally as a FIFO queue. Table I details vertices and edges rather than records as graph size rather than transaction size drives the computational complexity of the algorithm. The transactions included multiple connections between the same systems on the same ports, which is why the edge count is lower than the transaction count.

These filtering steps produced graphs are much smaller than expected. 60 billion connection log entries from 21 days in February transformed into 1,281,918 edges, as seen on Table I.

V. RESULTS

This section discusses GraphX and CGE execution times and their implications.

A. Execution

We tested the both graph engines on multiple graphs detailed in Table I. For each of the resulting graphs, we ran the respective functions on 1, 8, and 16 compute nodes using all available cores on each compute node. Results can be found in Table II and Table III.

TABLE II. CRAY GRAPH ENGINE RESULTS

Time for Compute			Vertices	Edges
1 Node	8 Nodes	16 Nodes		
2.61 s	29.38 s	29.27 s	5,419	11,726
77.7 s	653 s	718 s	15,359	52,042
353.89s	1643.29 s	1891.25 s	42,687	125,564
938 s	4730 s	6600 s	66,955	369,720

Time for Compute			Vertices	Edges
1 Node	8 Nodes	16 Nodes		
2.61 s	29.38 s	29.27 s	5,419	11,726
3205 s	15068 s	none	115,276	1,281,918

TABLE III. GRAPHX RESULTS

Time for Compute			Vertices	Edges
1 Node	4 Nodes	8 Nodes		
53.96 s	50.09 s	71.80 s	5,419	11,726
N/A	N/A	N/A	42,687	125,564

Table II details CGE execution times. A number of features present themselves in the execution times. The 1 node execution times are reasonable at 2.61 seconds for the first, smallest graph and then grow larger in an unexpected way, requiring nearly an hour to process. These run times suggest that the underlying graph topology is not strictly scale-free or small world and is wider than expected. Scaling these graphs to multiple nodes was not required for additional memory; even the largest graphs fit into the 256 GB of RAM present on a single node. Scaling to multiple nodes gave the algorithm more cores to use when executing the betweenness centrality algorithm. The observed runtimes did not reflect that expectation. Adding additional compute nodes actually resulted in longer algorithm execution times. This suggests there are inefficiencies present in the network communication between nodes that would have been exacerbated by any deviation from the scale-free or small-world model. Discussions of these execution times with Cray engineers indicated that they were aware of additional efficiencies available in the betweenness centrality implementation, and development is ongoing.

Table III details GraphX execution times. There were fewer execution times to report as GraphX did not complete in under 24 hours for any graph larger than the smallest graph used in this experiment. Adding additional nodes to the computation also resulted in inconsistent results. Given the network communication-heavy nature of GraphX being based on Spark, inefficiencies in network communication likely explain the inconsistent results seen when adding additional nodes. The inability of the GraphX implementation to run to completion on only the smallest graph in the experiment indicates that it is likely not ready for production use.

We examined the betweenness centrality results for scanners identified through other means and located some present within the more central nodes identified by the algorithm. The most central nodes did not appear to be scanners. Discussions with security professionals as the behaviors shown by the most central nodes are ongoing.

VI. Conclusion

Our tests indicated, in the case of using betweenness centrality as a graph function, that Cray Graph Engine was not only faster, but easier to implement than Apache Spark's GraphX. Cray's implementation of SPARQL had a version of betweenness centrality to use out-of-the-box while a custom version of distributed betweeness centrality had to be leveraged to achieve similar functionality. Additionally, GraphX does not include a query language, like SPARQL, requiring Scala programming, increasing the technical aptitude required to use the engine. CGE's SPARQL also came with significantly more graph functions than GraphX. Additionally, for GraphX, when dealing with larger graphs, the need to copy information about the graph to each compute node causes a constraint problem which we attributed to GraphX failing on our larger graph test cases.

VII. Acknowlegments

This document contains general information only and Deloitte Advisory is not, by means of this document, rendering accounting, business, financial, investment, legal, tax, or other professional advice or services. This document is not a substitute for such professional advice or services, nor should it be used as a basis for any decision or action that may affect your business. Before making any decision or taking any action that may affect your business, you should consult a qualified professional advisor.
Deloitte Advisory shall not be responsible for any loss sustained by any person who relies on this document.

As used in this document, "Deloitte Advisory" means Deloitte & Touche LLP, which provides audit and enterprise risk services; Deloitte Financial Advisory Services LLP, which provides forensic, dispute, and other consulting services; and its affiliate, Deloitte Transactions and Business Analytics LLP, which provides a wide range of advisory and analytics services. Deloitte Transactions and Business Analytics LLP is not a certified public accounting firm. These entities are separate subsidiaries of Deloitte LLP. Please see www.deloitte.com/us/about for a detailed description of the legal structure of Deloitte LLP and its subsidiaries. Certain services may not be available to attest clients under the rules and regulations of public accounting.

References

[1] E. Dull. "Cyberthreat analytics using graph analysis." CUG 2015

[2] U. Brandes, "A faster algorithm for betweenness centrality," J. Mathematical Sociology, vol. 25, no. 2, pp. 163-177, 2001.

[3] J. Berry and G. Mackey "The MultiThreaded Graph Library," CASS-MT 2009. http://cass-mt.pnnl.gov/docs/sc09_mtgl_presentation.pdf

[4] M. Kabiljo, D. Logothetis, S. Edunov, and A. Ching, "A comparison of state-of-the-art graph processing systems," unpublished.

[5] E. Hutchins, J. Cloppert, and R. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," unpublished.

[6] D. Marcous, "Distributed K-Betweenness (Spark),", unpublished.

[7] J Postal. RFC 793 – Transmission Control Protocol. 1981.

[8] Apache Spark mailing list http://apache-spark-user-list.1001560.n3.nabble.com/All-pairs-shortest-paths-td3297.html

[9] M. Castro, P. Druschel, Y. Charlie Hu, A. Rowstron, "Exploiting Network Proximity in Distributed Hash Tables," Published in International Workshop on Future Directions in Distributed Computing (FuDiCo), 2002. https://www.microsoft.com/en-us/research/publication/exploiting-network-proximity-in-distributed-hash-tables/

[10] Cray® Graph Engine (CGE) User Guide (S-3010-1000).