SCANSPHERE - AN ADVANCED NETWORK MAPPING AND ANALYSIS TOOL

Dhivahari Vivek, Katerina Walter, Turgerel Amgalanbaatar, Hosna Zulali, and Kulprawee Prayoonsuk

A project report submitted to UCD's College of Engineering, Design, and Computing (CEDC) in partial fulfillment of the requirements for CSCI 4738-39: Senior Design.

ABSTRACT.

Network scanners and vulnerability analysis tools are invaluable in discovering fingerprinting data about network hosts. These fingerprints consist of a diverse and large variety of attributes, ranging from host OS and open or closed ports, to running services and their patch levels or vulnerabilities. The similarity of these fingerprints can provide very important insights for network security administrators for tackling threat propagation, or conducting forensic analysis. Thus, we developed ***ScanSphere***, a network mapper tool that uses off-the-shelf scanning tools to fingerprint hosts in a network or across various networks, and then clusters these hosts based on the similarity of their whole or partial fingerprints. In other words, our tool generates a high-level map of the network where hosts are clustered according to their fingerprint similarity. In order to tackle the curse of dimensionality, our approach uses network embedding techniques. This tool provides network administrators with a holistic view of the network and assists them in real-time security decisions regarding threat management and analytics.

PROJECT APPROACH.

ScanSphere's front-end and back-end features are written entirely in Python 3. The tool's design process and development was executed in five stages, namely (1) Data Collection, (2) Graph Construction, (3) Graph Embedding Methodology (GEM), (4) Clustering, and (5) Graphic User Interface (GUI) implementation.

DATA COLLECTION

As we briefly describe in our abstract, *ScanSphere* uses off-the-shelf scanning tools to collect data on network hosts and their attributes. For a variety of reasons ranging from cost and efficiency to convenience and compatibility, this research prototype primarily uses **Nmap**, a free and open-source network scanner that is used to discover

hosts and services on a computer network by sending packets and analyzing the responses. Many of Nmap's features - including host discovery/service and operating system detection - influenced the team's decision to choose Nmap as this prototype's method of network scanning and data collection. The use of Python's *python-nmap* library was crucial in Nmap's seamless integration into our code. This library provides both the scanning tool itself and methods that we could use to manipulate and parse the tool's scan results, subsequently allowing us to extract the following information about a network host:

- Host name and host state
- Open/listening ports
- Operating system information, including:
    - Type
    - Vendor
    - Family
    - Generation

Using this library's built-in scanning options, *ScanSphere* has the option to (1) scan a single IP address, (2) scan a range of IP addresses (using CIDR notation), and (3) scan several specific IP addresses. After the preferred scan is complete, every host and its corresponding attributes are saved in a Dictionary data type in preparation for data transfer to a secure database server as part of the project's next phase.

GRAPH CONSTRUCTION

The network data which, at this point, is saved as elements of the Dictionary data type, is then transferred to a secure MongoDB server with four "collections" of documents (or database entries), namely ***nmapData***, ***nodes***, ***attributes***, and ***edges***. Additionally, a fifth collection we called counter with exactly one such document stores our iterable numerical ID for each database document (it was easier for our tool to parse through the database with numerical identifiers instead of MongoDB's assigned slew of random characters). Each entry from the Dictionary data type corresponds to a document in our *nmapData* collection which stores raw network scan results by host.

Through a very intricate process that uses the aforementioned numerical identifiers as pointers to relevant documents, our program scanned each document in *nmapData* to

create host nodes (or nodes with type "H") and attribute nodes (or nodes with type "A"), and these nodes are added as entries to the *nodes* collection. Once this was complete, the program finally paired each host node to its corresponding attribute nodes - each document in the *edges* collection represents one such pair or edge in our graph.

- ***nmapData***: The raw data returned from the nmap. Each host is represented with a single mongodb 'document' containing keys for identification, and a list of attributes found by nmap.
- ***nodes***: Two types of documents are stored in the nodes collection. Host nodes (or nodes labeled with type "H") are made to represent each device located in the scan. This node contains reference keys to the appropriate raw nmap data. Attribute nodes are made to represent scanned information in key/value pairs; eg: if a host was found with "os family: windows," an attribute node in the attribute collection is created with "key: os family" "value: windows". This is then referenced by the type "A" node in the nodes collection.
- ***attributes***: For ease of reference, the attribute node data is saved in a separate collection, with the appropriate reference keys to other mongodb documents.
- ***edges***: Each edge document takes the foreign keys of an 'H' node and an 'A' node, to relate them with a predetermined weight. This weight is decided by the importance of the attribute key/value. This edge collection then contains all necessary information to build our graph using the networkX library.

GRAPH EMBEDDING

In order to process large quantities of data effectively, we use Graph Embedding which is the process of mapping complex networks to vector spaces while preserving the properties of the network itself. Vector spaces are considered more valuable than raw data as the implementation is often simpler and faster than their graphic counterparts. Our tool uses two such graph embedding techniques - *Structural Deep Network Embedding* (*SDNE*) and *node2vec*.

The SDNE approach performs its construction while maintaining first and second order proximity to optimize a semi-supervised model that captures nonlinear network structures, which is unique to other embedding methods. First-order proximity is used with supervised components to preserve the local network structure. However, the

component for the second-order proximity is unsupervised and captures the entire network's structure. By optimizing them both, SDNE can preserve the global and local network structure.

Node2vec parameterizes random walk to understand low-dimensional representations of nodes by preserving network structures and neighborhoods of the graph's nodes. A second order random walk is implemented by sampling network neighbors of nodes, therefore preserving the local neighborhood. This creates a flexible algorithm that can include multiple definitions of network neighborhoods through the simulation of biased random walks to explore a variety of differing neighborhoods.

These techniques were implemented through the *Graph Embedding Methods* (*GEM*) Library, a Python package, that provides a framework for graph embedding techniques. This includes the evaluation of embedding with visualization, node classification, link prediction, and graph reconstruction.

K-MEANS CLUSTERING

After we embed our graphs, we now have this large amount of network data that we want to group based on previously specified similarity so that we can actually interpret this grouped data. K-Means clustering yields several advantages. For one, the algorithm is relatively easy to implement - especially with Python's *sklearn* library. K-Means can also scale well to large datasets. This is useful considering the scale and most probably large size of the computer networks *ScanSphere* is likely to scan. Perhaps most notably, K-Means clustering easily adapts to new examples. In other words, with each scan yielding widely different network structures, this tool can adapt its clusters to accurately represent new similarities found within the network.

Many cite the need to manually specify the number clusters as a disadvantage of K-Means clustering. Luckily, we can use the Elbow Method to determine the optimal number of clusters, known as *k*. After plotting the elbow method, The tool asks the user to interpret the plotted values by finding the so-called "elbow" (or the point at which the graph drastically changes) as our optimal *k* value. The user can then enter this *k* value, and the graph that is given to the user will reflect the number of clusters desired. The user can also try different values of k and see these reflected changes in new graphs.

The accompanying presentation video uses two real network examples retrieved through *ScanSphere*'s trial runs to visually explain this process.

However, what does this clustered data mean? How do we make this data useful? In every clustered graph, each node shares a one-to-one correspondence to a scanned host. This means that we can access the raw data and connect these data points to actual host devices scanned. Depending on the particular graph results, the user can then predict any trends within their network. For instance, if one such cluster (which we can call cluster A) comprises a certain part of the network, the devices found within this cluster must surely be related. For network security purposes, this could imply shared vulnerabilities. If a device in cluster A was compromised, the client might be encouraged to look at other devices in cluster A for similar vulnerabilities and enact a proper mitigation plan if it is appropriate.

Graphic User Interface (GUI)

The GUI was also developed in Python and primarily used the **tkinter**, the standard Python interface to the Tk GUI toolkit. The module itself was user-friendly, and so produced a user-friendly, interactive portal through which the client can interact with the program.