Abstract--Cognitive Radio (CR) is an intelligent radio technology that can learn from and adapt to its environment. Its flexibility enables it to opportunistically access spectrum white space over a large spectrum range and hence enables the development of Dynamic Spectrum Access (DSA) networks. Its huge potential in significantly boosting spectrum utilization makes it likely to be widespread in the near future. Since Cognitive Radio is a new and growing technology, measures need to be taken to protect and secure each system. As CR becomes more popular, the demand for malware detection will increase. The purpose of this project is to lay the four ground for designing an effective CR malware detection system based on dependency and correlation among CR component events. This solution is based on the observation that a normal-operating CR exhibits strong causal dependencies and correlations among its operational events. A CR that is infected by malware, however, is unlikely to exhibit similar relation. Our solution will monitor the CR operation events at multiple levels, including radio hardware and software, operating system, user application, and network levels. It will compare the event traces with the event correlation and dependency profile of normal CR operation and detect malware through the discovered discrepancy. We will achieve this goal using the Perl Scripting Language, GNU Radio, and the Linux System using Ubuntu.

I. INTRODUCTION

Cognitive Radio is a term that is becoming well-known throughout the engineering field, which is continuing to grow and evolve with time. It can be defined as “an intelligent wireless communication system that is aware of its surrounding environment and uses the methodology of understanding-by-building to learn from the environment and adapt its internal states to statistical variations in the incoming RF stimuli by making corresponding changes in certain operating parameters (e.g., transmit-power, carrier-frequency, and modulation strategy) in real-time” [1]. However, new technological advances, such as the ones being made with cognitive radio, bring new challenges. There is a need for security and protection for this type of advancement in technology. The motivation for this project is to create malware detection software for cognitive radios using dependence analysis. Specifically, a user can create a dependency graph from function calls of an application to determine if those calls are equivalent to the function calls of the dependency graph. If function calls do not match up, the user is alerted that malware is in the system.

II. BACKGROUND

Before we observe the issues of malware detection, certain background information must be understood. Cognitive radio is known to be “highly reliable communication whenever and wherever needed,” and “efficient utilization of the radio spectrum” [1]. Given that cognitive radio is becoming a widely-known use of
communication, great measures need to be taken to ensure the security of applications involving cognitive radio. Malware is known as malicious software, or “computer programs designed to infiltrate and damage computers without the user’s consent” [2]. There are numerous types of applications that are designed to detect malware and protect software from it. Our main focus is to search through files and detect if there is any present malware. One of our few applications that assisted us in this process is GNU radio, “which is free open source software that provides signal processing blocks to implement SDR systems” [3]. Inside of GNU radio lies an operating system called Ubuntu, which is very similar to the Linux system. GNU radio and Ubuntu allow a user to trace files, which is “a report of a program’s position in the in the code execution stack” [4].

III. METHODOLOGY

Several approaches were taken to begin the process of malware detection. A Python trace in Ubuntu was used to trace and read function calls from an application called dial_tone.py. Only one application was used because if we execute one successfully, others can be created in the same way. The application used was located in the GNU Radio applications that can be accessed in Ubuntu. When this file is run, a dial tone can be heard after a certain amount of time. Once the function calls were read, we were instructed to place them in a dependency graph, which was made using an application called Edraw Max. We noticed that there was unnecessary code located in the dial_tone.py file, therefore we deleted any code that would not change the effectiveness of the trace and reran the trace. Figure 1 shows an example of what our dependency graph will look like. Once the graph is complete, it can be used for comparison with other manual or automatic graphs.

IV. RESULTS

As a result to the problem, Figure 2 is the completed dependency graph of the dial_tone.py application. Looking at the figure, it is very elaborate and could easily be manipulated. If another graph was given to the user for the application, the user could manually check to see if there was malware. Malware would be detected if one or more bubbles in the dependency graph had been different. This is how the dependency graph is useful in detecting malware and providing security.

V. TECHNICAL PROCESS

Before creating the dependency graph of the dial_tone.py application, multiple conditions were met. These conditions included becoming familiar with Ubuntu commands, learning how to shorten code, find function calls, and trace the file of our application. Each condition played an important role in making sure the dependency
A graph would be correct so it would be able to be used for malware detection.

Being able to smoothly operate within Ubuntu allowed for accuracy and efficiency. Commands that needed to be learned were “cd”, “ls”, “vi”, “gedit”, “ctrl - c”, navigation keys, and also lines of code that played our dial tone. The “cd” command allows the user to change directories, for instance our application has to be accessed in the GNU Radio and Python directories. The “ls” command offers a list of applications or directories that are inside each directory. Both “vi” and “gedit” are text editors that were used to read the function calls and also edit code. The command “ctrl - c” would kill the application that is running and return to the next line in the terminal screen. Simple navigation keys that were helpful were “k” as up, “h” as left, “j” as down, and “l” as right. Finally, the most important line of code was the one that allowed access to our application. By typing in “python -m trace -T dial_tone.py” our application could be heard and then using a text editor, its function calls could also be viewed.

Once commands in Ubuntu were learned, realizing how to trace and access the function calls of the dial_tone.py application were the next steps in creating the dependency graph. A trace file shows the commands and code that the application is accessing and performing. This trace was accessed by typing the code “python -m trace -t dial_tone.py,” and then again typing, “test.output.” To read the function calls of the application the code, “python -m trace -t -T dial_tone.py,” and then “function.output.” From this command the function calls were used to make the dependency graph.

By viewing the code (Figure 3) for the dial_tone.py application it was noticed that some lines could be edited to shorten the amount of function calls. By typing into the command “vi dial_tone.py,” the code for the application can be viewed. Here the code can be edited by changing the sample rate to “4800,” the “parser” lines of the code were deleted, and the audio sink y axis was changed to an empty string (“”). The code was shortened (Figure 4), because when creating a dependency graph manually, it takes time, and with a shortened code, there are fewer function calls. With fewer function calls, the graph will still serve the same purpose but it will be easier to manually create.
The process of making the dependency graph for dial_tone.py was timely but fairly straightforward. After downloading Edraw Max, a flow chart creator, the function calls were read and circles were created depending on which class or function was being called for each section of code. For instance, in Figure 5, the class is shown above the module and function calls that are being produced. Looking at Figure 6, the class, module, and function calls were divided into shapes and connected creating the dependency graph.

To understand a little amount of what a section of a function call does look again at Figures 5. Here the audio class is being called and the audio_swig module being accessed, which is calling functions. Two of these functions being called include audio sink and source. These functions are in the audio module, which are connected to a computer’s sound card which will allow the dial tone to be heard. The dependency graph is made up of multiple functions calls that all play a role in making the computer create a dial tone.

In Figure 7, there is an example of what a malware would look like in the dependency graph form. The graph on the right represents the correct code, and on the right, a graph with a malware. The malware is highlighted in red to show that this function call being called is not the correct call. The user would have to go through each graph to pick out if there is malware in the code. If there is malware, the user can compare to the correct code and fix or even take out the malware being detected.

VI. CONCLUSION

A dependency graph was created from the “dial_tone.py” application in GNU Radio that was accessed through Ubuntu. Numerous commands were learned to complete this graph and a great deal was learned about the process. This dependency graph will be applied to check for any malware or code that might be different from this exact application. Other graphs can now be made and provide security in similar fashion for other applications. In the future, an automatic dependency graph would be scripted.
in a programming language, like Perl. This will allow for a faster process and have the graph in a code form that can easily be applied to other applications, and then in turn, cognitive radios. This will provide a dependable security for cognitive radios that users can trust and monitor.

VII. FUTURE WORKS

As mentioned in the conclusion the next step would be to make an automatic data structure in Perl. Algorithms and data structures should be considered because they are similar to the manual dependency graph that was created in this project. An algorithm is a sequence of computational steps that transform the input into the output [5]. The analysis of algorithms often requires us to draw upon a body of tools, in this case a graph [5]. The input would be function calls and the output would be a data structure that can be used to detect malware.

Perl also allows the user to read a text file line by line. Therefore, searching for “funcname”, for example, can be extracted through lines of code. These new lines can be placed in a new file and then converted to a data structure. Data structures can include tree diagrams, like binary search trees for example. These trees are what our dependency graph is but in a programming language form. Then when used together, algorithms can be used to print out tree diagrams. These data structures can then be made for cognitive radios and applied across a larger spectrum. As cognitive radios continue to become increasingly popular this need to dependable security will grow.

VIII. ACKNOWLEDGEMENTS

We would like to acknowledge Dr. Carl Dietrich, our REU site director, Dr. Yaling Yang, our mentor, and the National Science Foundation (NSF).

IX. REFERENCES


