Adversarial examples for malware detection

Csia Klaudia Kitti,
HA5YCV

Supervisors:
Dr Buttyán Levente
Nagy Roland

1 ABSTRACT

Machine learning has undergone tremendous development in recent years. Thanks to the development of machine learning, it has begun to completely transform today's technologies from traditional algorithms to modern deep learning architectures, putting them on a new footing. Its application spans a wide range of areas, including the cybersecurity department. However, like any new cybersecurity development or innovation, there is a risk that this information will be used for malicious attacks and create vulnerabilities. In our current situation, the attacker can modify the data during training or test time, thus allowing the classification of the models to do incorrectly, i.e., intentional perturbations generated in the test samples. It can spiral into an endless loop of creating an innovation, a patch and then causing a response attack. Develop something in response. This survey provides a comprehensive introduction to hostile attacks against malware detection systems. The article will focus on a specific attack on machine learning techniques to generate malicious files and change the structure of target files. This survey models the threat posed by the adversary and then describes these algorithms and the processes for developing them. Problems encountered in the research will also be mentioned, and what solutions have been suggested for them. The survey concludes with the identification of open future research directions.

Index conditions – *Enemy attack, Enemy modelling, Malware detection, Machine learning, Security, CFG, Graph analysis, Internet of Things*

2 INTRODUCTION

2.1 BACKGROUND

The Internet of Things (IoT) is a technology that emerged years ago that encompasses types of devices that can manage several distinct types of devices on a single network. Such a system can consist of multiple sensors, audio assistants, automatic controls and more. Unfortunately, many vulnerabilities have been found in these devices in recent years, and unwanted attacks have been carried out. It is essential to understand the IoT, which addresses these security issues, abstractions, and classifications through analysis. The promising direction is to use a graph-theoretic approach to this type of security, especially in the analysis of IoT malware. Feature vectors are representative static features that can be extracted from a graph, such as the Control Flow Graph (CFG), from which these types of classifiers work. Because CFG graph-based features can represent IoT software, these features can also be used to create an automatic detection protection system that can determine whether particular software is malicious or not. Machine learning algorithms and deep learning networks are actively used in detection/classification processes to distinguish malware from benign ones and, in effect, integrated by incorporating machine learning theory into the detectors of this type of software, which begins to work with graphs of incoming patterns. This type of machine learning can learn the representative features of a graph and then, based on these, tries to categorise which class it can be classified into based on the graph of the following, already unknown software (these will be the test vectors later). These are widely used today and are primarily found in industry, healthcare, cybersecurity and even finance.

The project was based on the idea of a current study and based on that, and the first goal was to prove that what they stated was feasible in practice. An examination of these followed this, and comparable results were obtained. It was also important because the initial study lacked information and worked with very few examples generated manually. The goal was to implement them again and then improve them, that is, to automate everything in it, thus encouraging work on the more extensive data set and examining exactly how the connection of two graphs works, which was also covered quite incompletely in it.

2.2 MOTIVATION

Unfortunately, such machine/deep learning-based software is prone to this vulnerability. The problem occurs when an opponent tries to manipulate these graphs, that is, to create adversarial examples (AE) that can be used to achieve misclassification, thereby passing its malware to the defence system. Although this is an active area of research, truly little research is being done to understand its impact on deep learning, especially those using CFG features. In practice, this type of attack will be expected in the future so that the opposing side can directly reclassify malware, which can have dire consequences for such sensitive applications, confirming the importance of the issue to be explored in more detail.

2.3 YOUR CONTRIBUTIONS

The project was based on the idea of a current study and based on that; the aim was first to prove that the work in it was feasible in practice. An examination of these followed it, and comparable results were obtained. It was also important because the initial study lacked information and worked with very few examples generated manually. The goal was to implement them again and then improve them, that is, to automate everything in it, thus encouraging work on the more extensive data set and examining exactly how the connection of two graphs works, which was also covered quite incompletely.

2.4 BACKGROUND

The first article based on the project was a 2019 study written by staff/students at the University of Florida for the 2019 IEEE International Conference. "*Adversarial Learning Attacks on Graph-Based IoT Malware Detection Systems*." It focuses primarily on attack methods, and then on a practical level, they created the graph embedding and augmentation attack mentioned above. Other types of attacks are also being considered. The basic idea proved to be particularly interesting. However, unfortunately, there was not enough information available based on the article, nor was there enough evidence that it would work. They created the classifier/detector themselves and manually put together six pieces of malware and benign, which were sent to the detector they built. From this, unfortunately, the identical replica proved to be almost impossible, so only the basic idea was carried forward. The second primary source was a dissertation completed last year by a graduate student at the Budapest University of Technology entitled "*Deep Learning-Based Malware detection using CFG*". Although the basic idea was drawn from the first article, after the first article revealed little information on how to replicate this type of attack accurately, the idea for usable tools came from this dissertation. The first article mentions that *Radar2* was used to create the graphs, but based on the dissertation description, the program called angr (Python library) mentioned there, in which the student performed the tests, proved to be much more promising. He examined the CFG generating functions provided by the program, how much it is worth working with, which has the advantage, and what results and statistics he managed to produce with the extracted

graphs. Based on the two sources, we were able to build the current stand-alone lab, so the time they invested saved much time in our current situation.

3 TECHNOLOGY BACKGROUND

3.1 BACKGROUND

First, let us examine the concepts already mentioned but not explained: **malware analysis**. Malware analysis includes the analysis of malware. The study itself can be a dynamic examination of the spread of malicious code, an assessment of the activities performed, or even a static review of the code itself. It is also important because the results of these analyses are usually used to build comparative databases of similar detectors so that the detector can classify as "good" or "bad." Still, it can also roughly determine what it is facing by type.

Control Flow Graph (CFG) is a graphical representation of the control processor calculations during the execution of programs or applications. Control process graphs are most used in static analysis and compiler applications because they can accurately represent the flow within a program unit. The CFG can show all the routes it can take during the program's execution. CFG is also a directed graph in our present case. Its edges count as control flow paths, and its nodes represent the basic logic blocks.

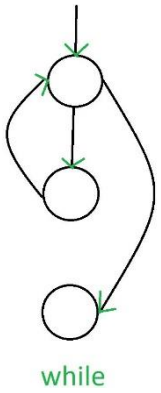

Figure 1.: *While cycle CFG*

The Python library called **angr** mentioned above was used in conjunction with another library called **networkx** for the CFG-focused part of the study. It has many pre-written graph computational functions, such as shortest path computation, betweenness centrality, closeness centrality, and many more used to generate feature vectors that will be used later. It is because angr has a function called CFGFast that can create a graph from binary. On the other hand, graphs of this type can now be managed by networkx, so the calculations were done using this library.

Graph metrics can also be considered as several representations of graph properties. For any binary (because it is customary to generate them from a binary), if a graph can be successfully generated, it can be used as a mathematical graph. Such operations were also performed using the networkx library, and the machine learning algorithm was later taught.

Random Forest is a supervised machine learning algorithm based on collaborative learning. Collaborative learning is a type of learning that combines diverse types of algorithms or the same algorithm multiple times to create a more efficient prediction model. The random forest algorithm combines several algorithms of the same type, i.e., numerous decision trees, and results in a *forest of trees*, hence the name "*Random Forest*". The random forest algorithm can be used for both regression and classification tasks.

4 SPECIFICATION AND DESIGN

4.1 BACKGROUND

The starting idea was the reference research and the dissertation from which this project started. The department already provided the malware and benign binaries database, consisting of code written four thousand for ARM processors and another four thousand for MIPS processors. This study deals only with

binaries written by the ARM processor, so only two thousand counts. Python3 libraries were used to convert the binaries to CFG, from which feature vectors were calculated by adding additional Python3 libraries. The files of the feature vectors created have been sorted into separate folders with the original name of the binary. The figures for the completed feature vectors were then imported into a sizeable standard CSV file, of course, the benign, the malware, and the concatenated graphs. The idea of concatenation also came from the reference study. Still, its dubious implementation was rethought, so we had to produce the idea that we create graphs not by starting points but by adding an extra node and then connecting the starting points with it.

Calculations took much time, so not all the two thousand files had been used by the end of the study. The codes generated roughly 800 benign, more than 1,100 malware, and almost 1,100 concatenated graphs, some successfully and some unsuccessfully. Fortunately, the number of unsuccessful never exceeded 10%, which means that 80 were unsuccessful in generating 800 benign, but the other 720, for example, have already become usable. The figures in these data sets are well above the values already in the above reference, so more reliable results could be obtained. Random Forest machine learning was used for a more secure classification, where the two prepared CSVs were submitted to it. Then their concatenated data sets later formed the training and testing base. CSVs of the concatenated graph's vectors were fed into the code after training to be evaluated by the algorithm already taught and classified accordingly.

5 IMPLEMENTATION

5.1 BACKGROUND

The adversarial example was generated using the Python3-compatible angr mentioned above. Of course, networkx, pandas, NumPy, and matplotlib were also used to create these graphs and their vectors. The graphical representation of graphs was not essential, but it helped a lot in understanding the processes. The generation of the CFG was done with the help of angr, after which the graph itself was completed. It could be checked if, for instance, it was generated in a .dot file or displayed graphically. When it was generated in a .dot file, it was nicer to see that the nodes would be the logical block that the program had to jump. Moreover, at the edges is the interaction itself, which shows how we can jump from one node to another. It can be tracked by including this as assembly code by generating it in a .dot file.

```
1 strict digraph  {
2 <CFGNode 0x100e1[22]>;
3 <CFGNode 0x100f7[6]>;
4 <CFGNode 0x100fd[4]>;
5 <CFGNode 0x10115[16]>;
6 <CFGNode 0x10101[12]>;
7 <CFGNode 0x10153[10]>;
8 <CFGNode 0x1015d[4]>;
9 <CFGNode 0x101f3[12]>;
10 <CFGNode 0x10161[8]>;
11 <CFGNode 0x10213[6]>;
12 <CFGNode 0x101ff[2]>;
13 <CFGNode 0x10125[12]>;
14 <CFGNode 0x10169[6]>;
15 <CFGNode 0x101f9[6]>;
16 <CFGNode 0x10219[10]>;
```

Figure 2: *Dot file - graph nodes*

```
49 <CFGNode 0x100e1[22]> -> <CFGNode 0x100f7[6]>  [ins_addr=65779, jumpkind=Ijk_FakeRet, stmt_idx="-2"];
50 <CFGNode 0x100f7[6]> -> <CFGNode 0x100fd[4]>  [ins_addr=65785, jumpkind=Ijk_FakeRet, stmt_idx="-2"];
51 <CFGNode 0x100fd[4]> -> <CFGNode 0x10115[16]>  [ins_addr=65791, jumpkind=Ijk_Boring, stmt_idx=15];
52 <CFGNode 0x100fd[4]> -> <CFGNode 0x10101[12]>  [ins_addr=65791, jumpkind=Ijk_Boring, stmt_idx="-2"];
53 <CFGNode 0x10115[16]> -> <CFGNode 0x10153[10]>  [ins_addr=65827, jumpkind=Ijk_Boring, stmt_idx="-2"];
54 <CFGNode 0x10101[12]> -> <CFGNode 0x1010d[8]>  [ins_addr=65801, jumpkind=Ijk_FakeRet, stmt_idx="-2"];
55 <CFGNode 0x10153[10]> -> <CFGNode 0x1015d[4]>  [ins_addr=65881, jumpkind=Ijk_FakeRet, stmt_idx="-2"];
56 <CFGNode 0x1015d[4]> -> <CFGNode 0x101f3[12]>  [ins_addr=65887, jumpkind=Ijk_Boring, stmt_idx=15];
57 <CFGNode 0x1015d[4]> -> <CFGNode 0x10161[8]>  [ins_addr=65887, jumpkind=Ijk_Boring, stmt_idx="-2"];
58 <CFGNode 0x101f3[12]> -> <CFGNode 0x10213[6]>  [ins_addr=66045, jumpkind=Ijk_Boring, stmt_idx=34];
59 <CFGNode 0x101f3[12]> -> <CFGNode 0x101ff[2]>  [ins_addr=66045, jumpkind=Ijk_Boring, stmt_idx="-2"];
60 <CFGNode 0x10161[8]> -> <CFGNode 0x10125[12]>  [ins_addr=65895, jumpkind=Ijk_Boring, stmt_idx=18];
61 <CFGNode 0x10161[8]> -> <CFGNode 0x10169[6]>  [ins_addr=65895, jumpkind=Ijk_Boring, stmt_idx="-2"];
```

Figure 3: *Dot file - graph edges*

Networkx, pandas, and NumPy took part in the mathematical calculations, creating a .txt file displayed line by line with the feature vectors. If it was necessary to display the graph graphically, matplotlib was also included at the beginning of the code and was called directly during code generation. Then, with the help of another code, built-in Python functions, all the .txt feature vector files created so far will be included in a CSV file, with vectors of a graph line by line, and they got a number format converter for Random Forest because it cannot process long numbers. Of course, this could have been solved with Random Forest running with *StandardScaler*, but it was easier to verify that all these scaling were successful by implementing them at the time of generation. Corrupted files from which generation failed could be included in the CSV as an empty queue, which was then deleted for Random Forest to work correctly.

The most spectacular part of the whole research is the operation of Random Forest, which looks to get the two CSVs in the first round, one containing the generated benign vectors and the other containing the malware vectors. After that, they both get an extra column called "Hit," which is the classification for us in this case. The benign file gets a column with zeros in it. In other words, a zero at the end of each row indicates that this data is harmless in terms of categorisation, and the same is true for a malware file but a column with ones in it. Furthermore, now that each has received a tiny "flag bit", the two databases are concatenated using the *Concat* function. In this case, the code copies the values from the other database after the values from one database. It is visible after the subsequent database dump. What had to be noticed during concatenation was that the size of the two databases had to be the same. Otherwise, the program would be gone. (This can be overwritten to some degree, but it was not such a severe problem that it needs to be addressed separately.)

	Edges	Nodes	Density	Components	...	Median_dc	Mean_dc	Sd_dc	Hit
0	7546	3587	0.000	269	...	0.000	0.001	0.002	1
1	9637	4304	0.000	213	...	0.000	0.001	0.001	1
2	9270	3627	0.000	139	...	0.001	0.001	0.002	1
3	11871	5765	0.000	1119	...	0.000	0.000	0.001	1
4	58	80	0.009	80	...	0.025	0.018	0.011	1

[5 rows x 25 columns]

Figure 4.*: Random Forest concatenated databases*

After that, our completed, concatenated database will be divided into two parts: there will be a part with which we will train the algorithm, this will be part of the training, which makes up 80% of the database, and the remaining 20% will be evaluated to see how accurate the accuracy, squarely, tests itself. These values are broken down to show the current accuracy for each run. It is also an essential step because

when training, 80% of the data in the database is always randomly selected to see different accuracy per run.

```
   Edges  Nodes  Density  Components  ...  Median_dc  Mean_dc  Sd_dc  Hit
0   7546   3587    0.000         269  ...      0.000    0.001  0.002    1
1   9637   4304    0.000         213  ...      0.000    0.001  0.001    1
2   9270   3627    0.000         139  ...      0.001    0.001  0.002    1
3  11871   5765    0.000        1119  ...      0.000    0.000  0.001    1
4     58     80    0.009          80  ...      0.025    0.018  0.011    1

[5 rows x 25 columns]
Accuracy score:  0.9855072463768116
Confusion matrix:
[[145   0]
 [  4 127]]
Classification report:
              precision    recall  f1-score   support

           0       0.97      1.00      0.99       145
           1       1.00      0.97      0.98       131

    accuracy                           0.99       276
   macro avg       0.99      0.98      0.99       276
weighted avg       0.99      0.99      0.99       276
```

Figure 5: *Random Forest accuracy*

```
0x10101
DiGraph with 14 nodes and 16 edges
0x8074
Graph with 1 nodes and 0 edges
```

Figure 6: *Checking entry point*

The massive headache caused the proper concatenation of the graphs during the semester. During the later work, it would be possible to easily separate them from each other and make it run again with the next goal. The referred study was accomplished by connecting the starting and ending points of the two graphs, and thus the concatenated graph was created. It has raised several questions, as there can be many endpoints running a program, so it is difficult to determine what a fixed endpoint can be for a program, as it can be different. The idea came from introducing a new node that will be the starting point of the graph and will be bound to it by 1-1 edge of the two different graphs. In terms of implementation, it looks like three graphs will be generated during the program. The first will be the malware graph, the second will be the benign graph, and the third will be a single node stored as a directed graph for successful concatenation. Then we copy all the edges and points of the other two graphs into this single node graph. Finally, the connection of the three graphs follows. Bluntly, we draw a line from the graph consisting of a single node to the starting node of one graph and then step on the same at the other. The examination of the starting address to see if the graph will be bound to an exemplary node was tested using the *.entry* function in the Python directory *pwnlib.elf.elf*. It is also clear from the example file that the initial address of the first appended graph follows the graph of a single node. After testing this on a few more graphs, it has become inevitable that even if the program is automated, it will still connect the graphs properly and connect them to the starting point. These steps have been taken so that later when this graph needs to be disassembled to run, it will have fewer problems with rebuilding.

```
1 strict graph  {
2 7;
3 <CFGNode 0x10101[18]>;
4 <CFGNode 0x1011f[4]>;
5 <CFGNode 0x10113[10]>;
6 <CFGNode 0x10129[14]>;
7 <CFGNode 0x10137[4]>;
8 <CFGNode 0x101db[10]>;
9 <CFGNode 0x1013b[8]>;
10 <CFGNode 0x10143[6]>;
11 <CFGNode 0x1011d[2]>;
12 <CFGNode 0x1010f[4]>;
13 <CFGNode 0x10149[8]>;
```

Figure 7: *Merged graph .dot file*

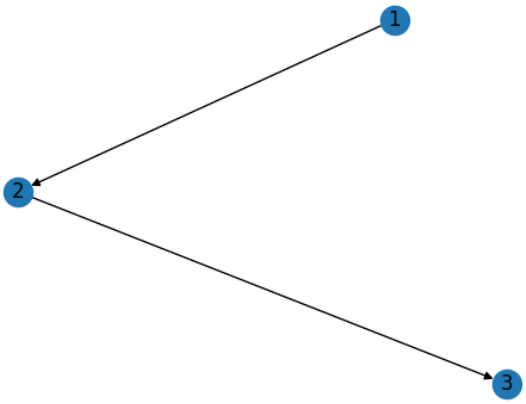

Figure 8: Test Directed Graph: G Graph

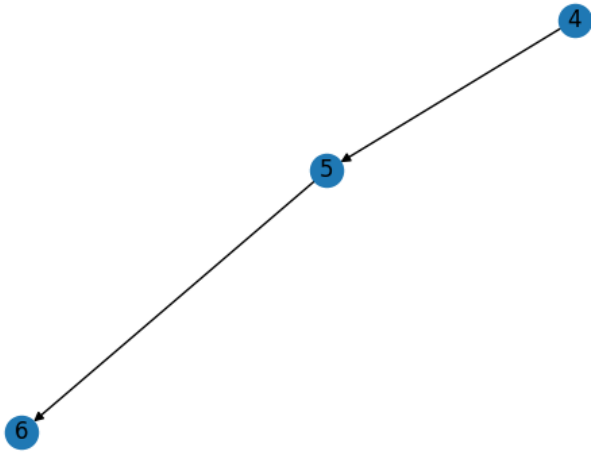

Figure 9: Test Directed Graph: H Graph

Figure 10: Test Directed Graph: F Graph

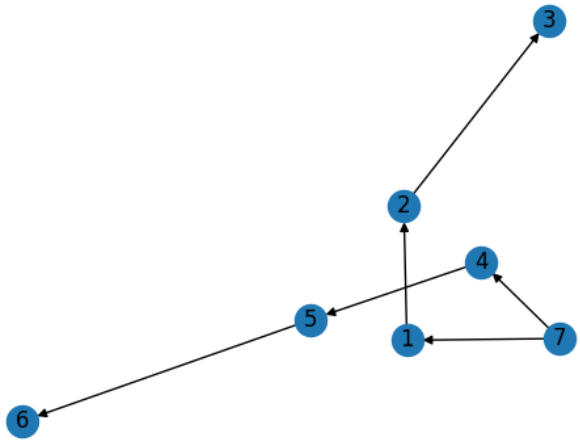

Figure 11: Test Directed Graph: Merged graph - F

We run the already known feature vector generation code on this concatenated graph in the last step. Finally, we hand it over to the already done Random Forest algorithm to classify whether the submitted file is benign or malware.

6 Results and evaluation

6.1 Background

Results came as a pleasant surprise, as the assumption that if a concatenated CFG is passed through the classifier, the file can be successfully classified aside. On the other hand, according to the above reference study, *"... while the GEA approach can misclassify all malware as benign"*, the misclassification was unsuccessful in some cases. (Although this study does not yet cover this statistic with evidence) it is a conjecture that if the size of the benign file is relatively smaller than the size of the malware file, the

misclassification will not be successful. So, the claim that any benign and malware-catenating always results in misclassification is not always accurate because the graphs' size matters in both files. Testing looked like a roughly small to medium-sized malware had been selected and concatenated with more than a thousand benign. Thus, this result was obtained by testing the CSV of the concatenated graph files in Random Forest:

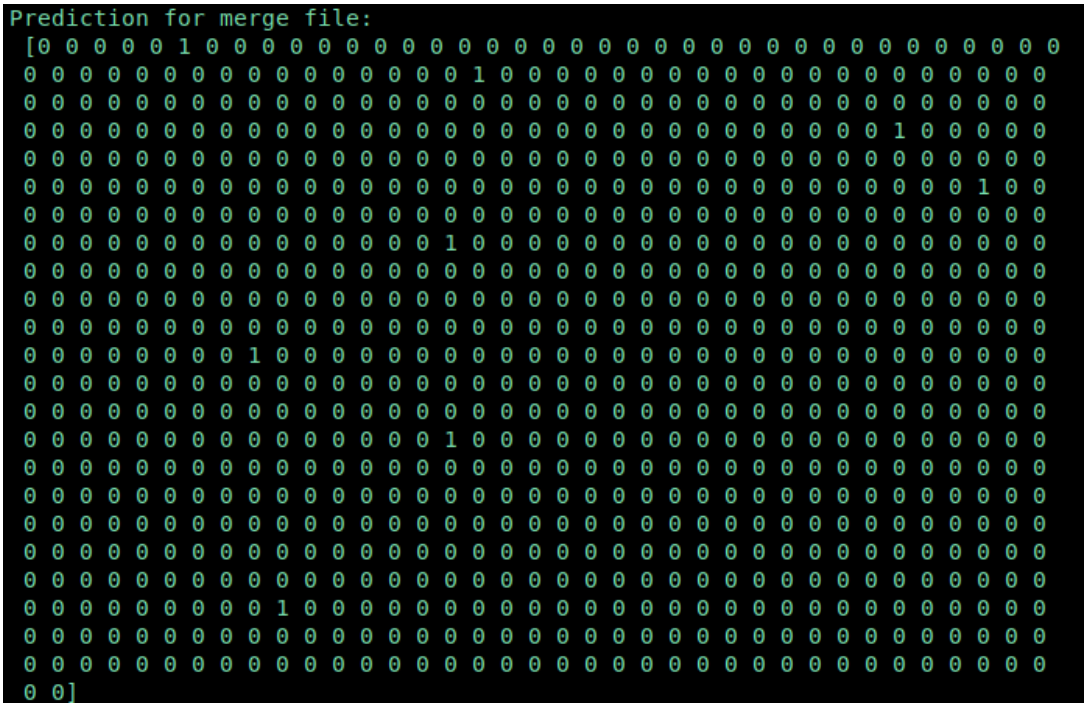

***Figure 12**: Random Forest classification results*

[1]

The result was similar for all runs; although it is considerably few, one is always present, as there are tiny benign files in the database.

7 CONCLUSION AND FUTURE WORK

7.1 BACKGROUND

This study aimed to examine an existing study proposition and then put the idea into practice. Within the framework of this practice, great emphasis was placed on the fact that all current runs of the program should be as fast and optimal as possible. In any case, the least likely human interaction should be required to run them. In other words, the codes should be automated as much as possible. It also allows the program to work continuously, even on larger data sets, gaining as many examples as possible. The topic itself is primarily an in-depth analysis of malware binaries through the creation of abstract structures using CFGs, which are analysed in several ways, such as the number of nodes and edges, and graph algorithmic constructs such as average shortest path, proximity, density and a good few more aspects. It was followed

[1] As a reminder: 0 = benign, 1 = malware.

by training a classification system from the feature vectors generated in the previous step (from both benign and malware samples) and then generating adversarial examples by combining malware and a benign graph based on a specific idea. The generated model was finally sent to the already completed classifier to check if it had succeeded in misclassifying it. Although this approach itself has achieved an erroneous misclassification rate, it is not yet guaranteed that the constructed AE will be able to retain its functionality after this. One source of problems that have not yet been studied is that more minor, detached graphs are created when these graphs are generated. Although they remain in the concatenated graph after concatenation, it is questionable whether they will cause a problem when disassembled.

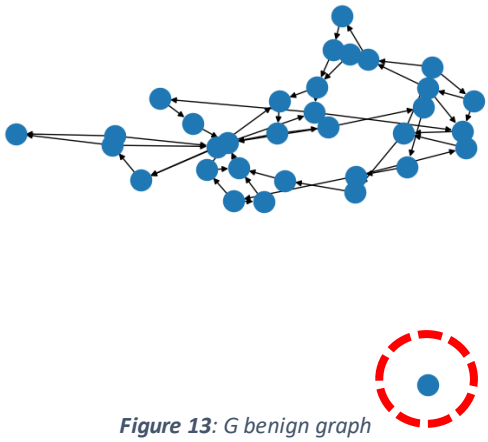

***Figure 13**: G benign graph*

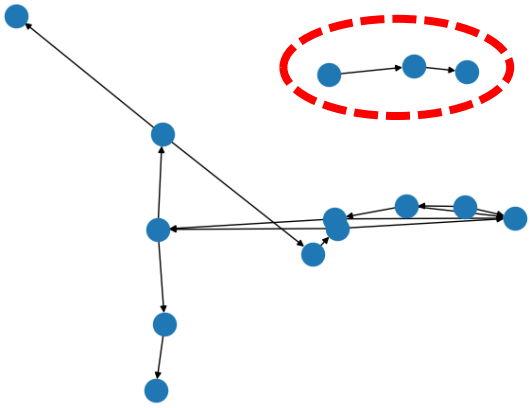

***Figure 14**: H malware graph*

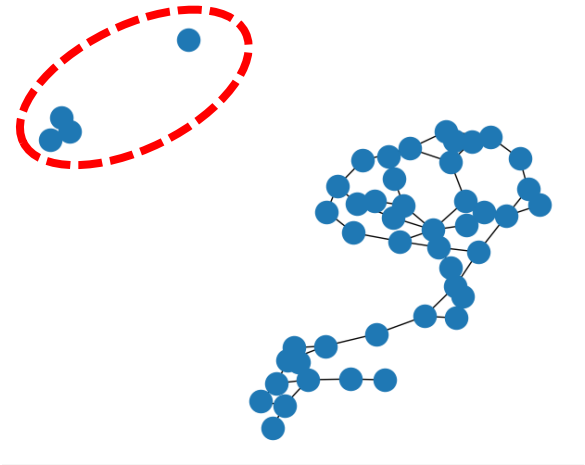

***Figure 15**: F merged graph*

In the following rounds, the emphasis will be on making the entire program design even more optimal and reaching an even more substantial number of examples. One more focus will be on working on as massive a data set as possible. Finally, the most important thing is to prove, after a successful misclassification, that the malicious part can still be taken out of the combined graph and then re-transformed into a form that makes it executable, and later that we need more thoughtful IoT malware detection programs that can already eliminate this problem.