A Report on Malicious Program Detection in Small Embedded System

Rajorshi Biswas
Department of Computer and Information Sciences
Temple University, Philadelphia, PA, USA
rajorshi@temple.edu

***Abstract*—The increase of usage of small computing units increases vulnerability and playground of attackers. In large scale industries, parts of the machineries are controlled by microcontrollers like Atmega16 or Arduino. The microcontrollers are connected with a wired or wireless network and sometimes a malfunctioning unit can cause failure of other units. In this project, we study different probable malicious programs that can run on Arduino and their detection procedure. We implement a monitoring program that provides resource usage information of all programs in Arduino. The usage information is analysed using multiple machine learning approaches including linear regression, support vector machine, adaBoost, decision tree, neural network, and k-nearest neighbor. We collect usage information over fifteen sample programs and some malicious programs with different settings for long periods of time. Our results shows that the activity of maliscious program can be detected with 99.6% accuracy.**

***Index Terms*—machine learning, malicious program detection, arduino, microcontroller, neural network**

I. Introduction

Nowadays, the usage of small computing units in large scale industries is increasing. This small embedded system are becoming vulnerability and playground of attacker. Parts of the machineries in large scale industries are controlled by microcontrollers like Atmega16, Raspberry PI, or Arduino. Most of them are connected with a common wireless or wired network. Figure 1 shows s industrial communication system that monitors compressor stations [1]. This kind of network is subjected to multiple types of attack including DDoS and Memory/CPU misuse attack. For example, a program running in microcontroller may continuously allocate memory until free heap space is filled up. Therefore, other programs cannot allocate memory and their normal activities are hampered. Another type of malicious programs create multiple tasks with high priority. Therefore, the CPU spends a lot of time executing garbage programs. Our goal in this project is to detect malicious activity of programs in small embedded system. The embedded systems are composed of small computing and memory units. So, its not possible to run heavy machine learning in the microcontroller. We can run light program that can collect some usage information and activity log and sent to the detection server. The detection server runs the machine learning methods to detect the malicious activity.

In this project, we build a prototype of the system. We implement a malicious program detection system. We divide the project into three components:

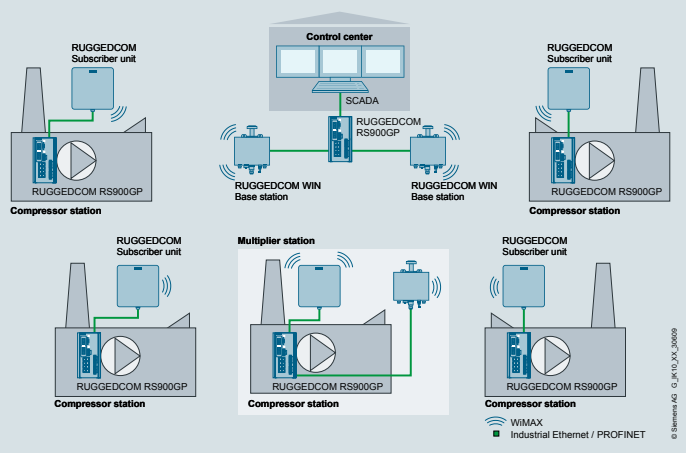


Fig. 1. Monitoring compressor stations with small embedded system.

1) **Embedded Module:** In this module, we implement multiple malicious and an activity logger program. The activity logger program sends the number of running tasks and free heap memory through serial communication cable.
2) **Communication Module:** In this module, we establish connection between microcontroller and computer using serial communication. We implement a receiver program that receives the activity log from the microcontroller and save to file.
3) **Detection Module:** In this module, we implement different machine learning algorithms to detect malicious activity.

To our best knowledge there is no such kind of public dataset available. Therefore, we generate our own dataset. We use the FreeRTOS operating system for multitasking support. We run different sample programs which comes with FreeRTOS [2]. We build some malicious programs. We run our malicious program and activity logger to generate activity log. We run the malicious and sample programs and collect the free heap memory and number of running tasks and save them in a file. We labeled the usage information as malicious if any of the malicious programs are running with the sample programs. If none of the malicious programs is running then we label the usage information as non-malicious. We compare performances of multiple machine learning algorithms including linear regression, support vector machine, adaBoost, decision tree, neural network, and k-nearest neighbor.

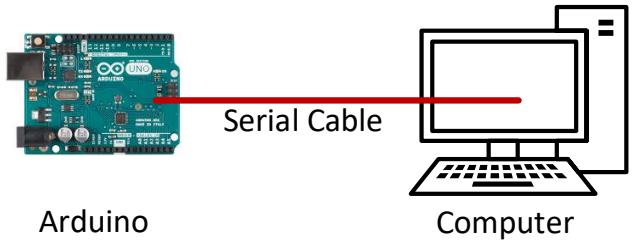


Fig. 2. system architecture.

II. Approaches

In this section we describe our approaches for building the system and malicious program models.

A. System Architecture

Figure 2 shows the complete system model. In our system, we have three modules: embedded module, communication module, and detection module.

In the embedded module, we needed a monitoring program which periodically gets the number of running task and free heap memory. We could get these information when the program starts but dynamically getting these information was not working initially. We explored the source code of FreeRTOS to see the root cause of the problem. We found that the FreeRTOS does not allow dynamic memory allocation (malloc()) from a task. Which is good in a sense that nobody can write a malicious program that allocates memory until it is full. In other sense, it is bad that our program cannot get these usage information dynamically. We overcome the problem by using another version of heap implementation. We use the Atmel Studio as SDK. It is difficult to debug in Ateml Studio because there is no serial output library. Therefore, we needed to add a LCD console to the Arduino. We added the LCD and run some sample code that can output to the LCD. We use the keypad to take input and create/delete task and output to the LCD display. Our embedded module is shown in Figure 3. We run some sample program including LED blink, analog to digital conversion, serial communication, keypad scanning, and LCD output.

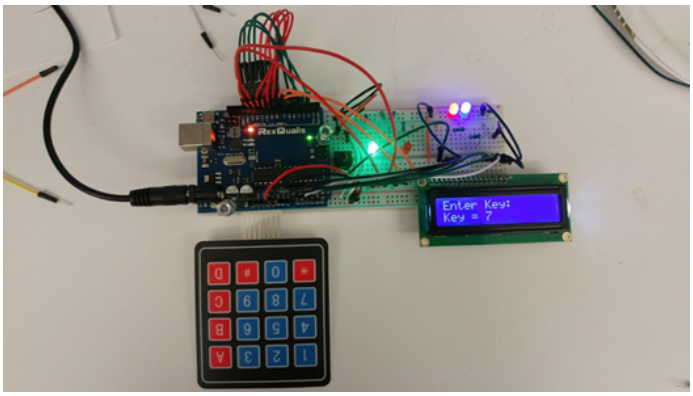


Fig. 3. Embedded module.

Algorithm 1 Malicious program type 1

```
1: Procedure: MAL-PROG-T1(void)
2:     while true do
3:         x ← random number.
4:         p ← allocate x byte memory.
5:         delay for random milliseconds.
6:     end while
7: end Procedure:
```

In the communication module, we connect our computer with a serial communication cable. We needed another program to receive the data which come through the serial port. Therefore, we build a C/C++ program to receive the data. The number of tasks and free heap memory are both integer numbers and we send them as byte stream separated by white space from arduino. The program receives the data and saves to a file.

In the detection module, we use python to run some machine learning algorithms on the collected dataset. We split the stream into multiple overlapping chunks. The chunks are used as features. Then we run linear regression, support vector machine, adaBoost, decision tree, neural network, and k-nearest neighbor algorithms on the dataset and compare the accuracy.

B. Malicious Program Model

We define three types of malicious programs. Type 1 malicious programs allocate memory for nothing and do not free them. Type 2 malicious programs creates many tasks so that the cpu of arduino becomes very busy. The type 3 malicious programs are mix of type 1 and 2. They intelligently creates tasks and allocates memory.

Algorithm 1 shows the complete method of type 1 malicious program. This programs keep allocating random memory but never free up. Therefore, when this kind of programs run in arduino we will always see some non-increasing numbers of free heap memory. This is easily detectable. Algorithm 2 shows the complete method of type 2 malicious program. This type of programs keep creating unnecessary tasks but never terminate them. Therefore, when this kind of programs run in arduino we will always see some non-decreasing numbers of running task. This is also easily detectable.

Algorithm 3 shows the complete method of type 3 malicious program. This type of programs sometimes create creating unnecessary tasks or allocate memory. They free or kill the task with a probability. Therefore, when this kind of program runs in arduino we will see both decreasing and increasing number of running task or free heap memory. If we consider the overall progress we will definitely see the amount of free heap is decreasing and the number of tasks is increasing. This kind of malicious programs are hard to detect.

III. Data Collection and Prepossessing

We compile sample program with our monitor program and run for 30 min on an average. We collect the free heap memory and number of tasks for 7394 and 3742 times for sample non-malicious and malicious programs. Because of the unexpected

Algorithm 2 Malicious program type 2

Procedure: MAL-PROG-T2(p)
  while true **do**
    $x \leftarrow$ random number.
    if $x < p$ **then**
      create new task.
    end if
    delay for random milliseconds.
  end while
end Procedure:

Algorithm 3 Malicious program type 3

Procedure: MAL-PROG-T3(p, p')
  while true **do**
    $x \leftarrow$ random number.
    if $x < p$ **then**
      create new task p.
      ENQUEUE(P)
    else
      $p \leftarrow$ allocate some memory.
      ENQUEUE(P)
    end if
    $x' \leftarrow$ random number.
    if $x' < p'$ **then**
      $p \leftarrow$ DEQUEUE()
      kill or free p
    end if
    delay for random milliseconds.
  end while
end Procedure:

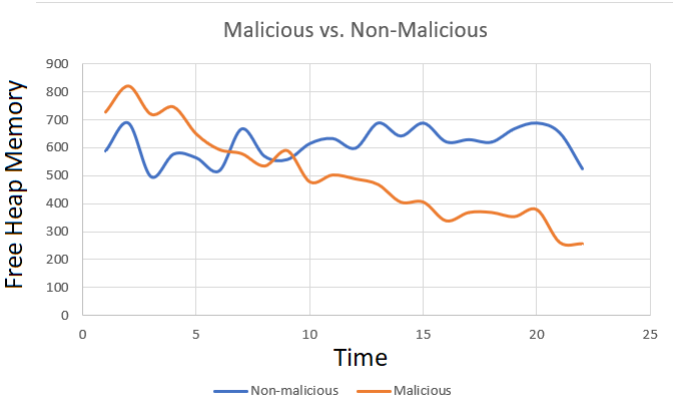
crash and transmission error there are some errors in data. We had a java program that filters out the garbage data from the saved files. A subset of the dataset is plot in Figures 4(a) and 4(b). Figure 4(a) shows comparison between non-malicious and malicious programs in terms of free heap memory. We can clearly see that the overall free memory is decreasing by time. On the other hand, the amount of free memory in non-malicious program is fluctuates but no overall decrease is observed.

In Figure 4(b) we see the behavioral difference in terms of number of running tasks. The number of tasks in non-malicious program remains similar and low. On the other hand, the number of tasks in malicious program increased over time and remains high.

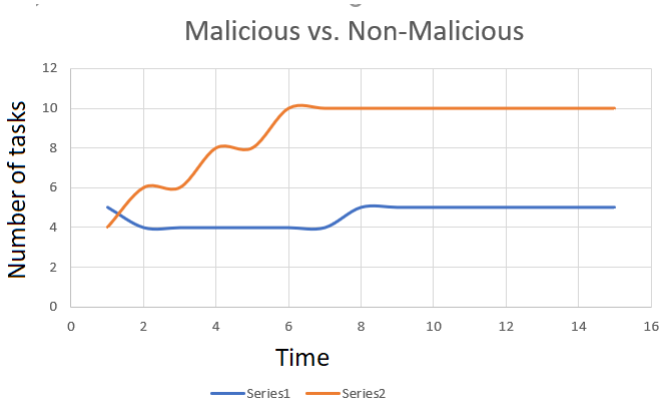
After collecting data, we divide the stream into fixed size overlapping chunks. We arrange the series of free heap memory numbers at the beginning and the number of tasks at the end. For example, if we select the chunk size 10, then the first sample contains 20 attributes including first 10 free heap memory and first 10 number of tasks. We slide the window of chunk by 1 byte and take another chunk. Therefore, our dataset contains two classes: malicious and non-malicious. We divide the dataset into test (40%) and train (60%) dataset randomly. Details are shown in the Table I.

TABLE I
DATASET DESCRIPTION

Number of samples in non-malicious class	7384
Number of samples in malicious class	3732
Number of non-malicious programs	15
Number of malicious programs	5
Train-test ratio	$60\% - 40\%$



(a) Difference in free heap memory.



(b) Difference in number of running tasks.

Fig. 4. Behavior of malicious programs.

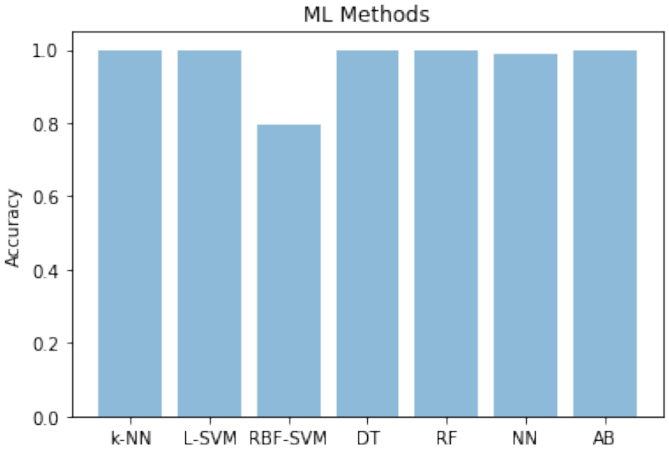
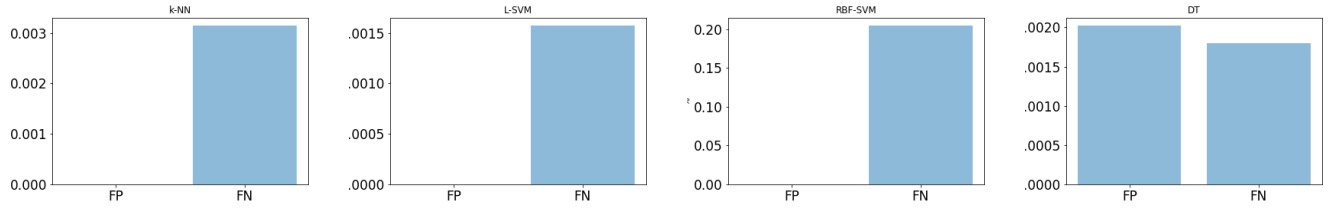


Fig. 5. Accuracy of all ML methods.

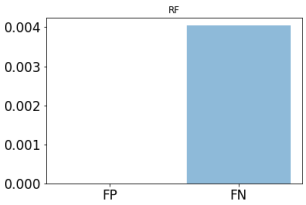


(a) K-nearest neighbor.

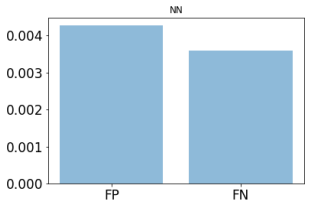
(b) Linear SVM.

(c) RBF-SVM.

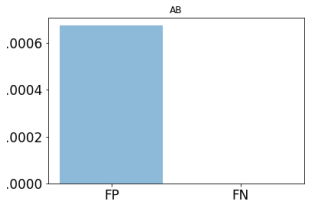
(d) Decision tree.



(e) Random forest.

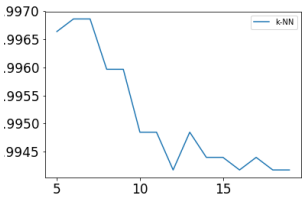


(f) Neural network.

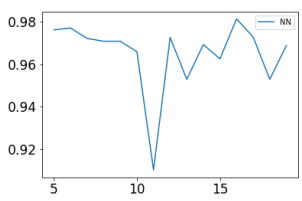


(g) AdaBoost.

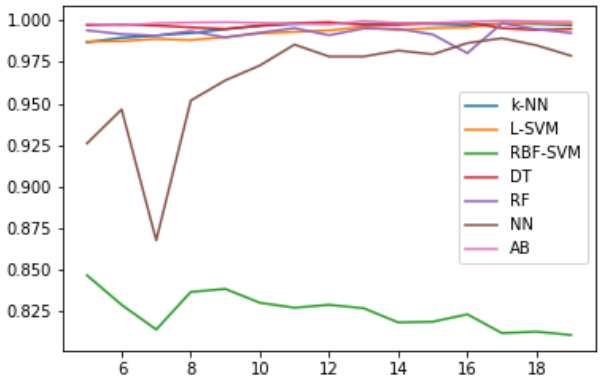
Fig. 6. False positive and false negative of different approaches.



(a) k-NN effect of k.



(b) NN effect of hidden layers.



(c) Effect of chunk size.

IV. Experimental Results

We applied k-nearest neighbors (k-NN), linear SVM (L-SVM), RBF SVM (RBF-SVM), Decision Tree(DT), Random Forest (RF), Neural Net (NN), and AdaBoost (AB) to the dataset. The obtained accuracy is plotted in Figure 5. We can see that the k-NN method has the highest accuracy 99.6%. We used the $k = 7$ and uniform weight settings. We can see that the false positive of k-NN is 0% and false negative is 0.04%. False positive and false negative are plotted in Figure 6.

We change setting of the k-NN and NN to get improved result. We varied the value of k in k-NN and the accuracy decreased by k. Figure 7(a) shows the accuracy by k in k-NN. We also varied the number of hidden layers in NN. We can see that the accuracy is decreasing from upto 11 hidden layers. After that the accuracy started to increase and the peak achieved at 18 hidden layers. At 11 hidden layers we see the accuracy is very low which may be an occurrence of over-fitting. We kept the chunk size 10 for these experiments.

Next, we varied the chunk size and observe the improvement in accuracy. Accuracy of different approaches by the chunk size is plotted in Figure 7(c). We can see that accuracy of most of the approaches are increasing by the chunk size. This is normal because larger chunk size means long period of observation. NN, RBF-SVM, and RF did not perform consistently for different chunk size. Therefore, we can conclude that the NN and RBF-SVM are not suitable classifiers for this kind of applications.

V. Conclusion

In this project, we developed a prototype of large scale industrial machinery controller and defined some types of malicious program. We built a monitoring program for data collection from embedded system (Arduino) and analyzed the collected data for malicious program detection. We applied multiple machine learning algorithms to classify the running program to malicious or non-malicious. We achieved 99.6% accuracy in detecting malicious program with 0% false positive. We also found that the neural network which is widely used for complex classification does not work well. Instead, we observed highest accuracy on k-nearest neighbor approach with specific settings. Codes and dataset are available in [3].

References

[1] https://w3.siemens.com/markets/global/en/oil-gas/pages/industrial-communications.aspx.
[2] FreeRTOS, https://www.freertos.org.
[3] https://github.com/raj0rshi/ml_project.