© 2018 Kyo Hyun Kim

APPLICATION OF DEEP LEARNING FOR PREDICTING SCHEDULES IN REAL-TIME SYSTEMS

BY

KYO HYUN KIM

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Sibin Mohan

ABSTRACT

Hard real-time systems are often used in safety critical systems: a task missing a deadline can be catastrophic for the system and endanger human lives. To guarantee that it meets every deadline, hard real-time systems are designed to have deterministic behavior. However, such determinism is prone to timing inference attacks. Using an analytical approach, an inference attack can be launched with a priori knowledge about the task-set. However, the advancements in deep learning opens new methods that can be used to carry out such attacks. We believe that the current state of machine learning algorithms is powerful enough to launch the attack without the complete a priori knowledge.

Therefore, we propose a novel architecture that will accurately predict future occurrences of target tasks in systems using real-time scheduling algorithms. We intend to use minimal information, for instance by observing only the sequences of busy intervals and rest intervals. The architecture will: infer size of the task-set, map tasks to each time steps of busy intervals and predict future task execution.

To my family, for their love and support.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Sibin Mohan and Assistant Professor Sanmi Koyejo for their guidance. I fully appreciate my advisor's faith in me and his guidance in the systems perspective of this research. I am thankful for Prof. Sanmi Koyejo for his advice in the machine learning perspective of this research.

I would also like to express my thanks to Chien-Ying Chen. He has provided multiple resources for the project which I really appreciate.

TABLE OF CONTENTS

LIST OF TABLES . . . vii

LIST OF FIGURES . . . viii

LIST OF ABBREVIATIONS . . . x

CHAPTER 1 INTRODUCTION . . . 1

CHAPTER 2 RELATED WORK . . . 3

CHAPTER 3 BACKGROUND . . . 5
- 3.1 Feed Forward Neural Nets . . . 5
- 3.2 Recurrent Neural Nets . . . 5
- 3.3 Long Short-Term Memory . . . 6

CHAPTER 4 ARCHITECTURE . . . 8
- 4.1 Adversarial Model . . . 9
- 4.2 Task-set size inference . . . 9
- 4.3 Task to busy interval mapping . . . 10
- 4.4 Future task prediction . . . 11

CHAPTER 5 EXPERIMENTAL SETUP . . . 12
- 5.1 Problem Formulation . . . 12
- 5.2 Model . . . 13
- 5.3 Features . . . 14
- 5.4 Utilization . . . 14
- 5.5 Task-set Size . . . 14

CHAPTER 6 ANALYSIS OF RANDOM TASK-SET GENERATION 15

CHAPTER 7 TASK PREDICTION RESULTS . . . 22
- 7.1 Effect of Total Utilization and its distribution on Performance 23
- 7.2 Effect of Feature Engineering on Performance . . . 23
- 7.3 Effect of task-set Size on Performance . . . 24
- 7.4 Other Trends . . . 24

CHAPTER 8 LIMITATIONS AND FUTURE WORK 27
8.1 Performance Under Randomized Traces 27
8.2 Finishing the First Two Stages 27

CHAPTER 9 CONCLUSION 28

REFERENCES 29

LIST OF TABLES

4.1 Notation definition 8

5.1 Formulation Parameter 13
5.2 Weighted Cross Entropy Symbols 13
5.3 Training Parameters 13

6.1 Default Generator Parameters 15

7.1 Result in F1 Score 22
7.2 Sample task-set Count 23

LIST OF FIGURES

4.1 Our proposed architecture. The arrows represent dependencies. The blue shaded squares represent collections of randomly generated data based on the information collected from the dependency arrow. 9

6.1 Histogram of task-set distribution of utilization. The x-axis is the utilization and the y-axis the number of tasks in the given bin. The x-axis is split into 100 bins. 17

6.2 Histogram of task-set distribution of utilization for high, medium, low utilization task-sets respectively from top to bottom. The x-axis is the utilization and the y-axis the number of tasks in the given bin. The x-axis is split into 50 bins. 18

6.3 Histogram of per-task distribution of utilization for high, medium, low utilization task-sets respectively from top to bottom. The x-axis is the utilization and the y-axis the number of tasks in the given bin. The x-axis is split into 50 bins. 19

6.4 Histogram of per-task distribution of execution for medium utilization task-sets. The x-axis is the execution time and the y-axis the number of tasks in the given bin. The x-axis is split into 50 bins. 20

6.5 Histogram of per-task distribution of period for medium utilization task-sets. The x-axis is the period and the y-axis the number of tasks in the given bin. The x-axis is split into 50 bins. 21

7.1 The plot shows per-task performance of "not heavy" medium utilization task-set. The red dots are the high priority tasks. Green dots are medium priority tasks, and blue dots are low priority tasks.The x-axis is utilization, the y-axis is performance metric. 25

7.2 The plot shows per-task performance of low heavy low utilization task-set. The red dots are the high priority tasks. Green dots are medium priority tasks, and blue dots are low priority tasks. The x-axis is utilization, the y-axis is performance metric. 26

LIST OF ABBREVIATIONS

EDF	Hard Real-Time Systems
HRTS	Hard Real-Time Systems
IoT	Internet of Things
LSTM	Long Short-Term Memory
NLP	Natural Language Processing
RMS	Rate-Monotonic Scheduling
RNN	Recurrent Neural Network
RTS	Real-Time Systems
SCADA	Supervisory Control And Data Acquisition
SRTS	Soft Real-Time Systems

CHAPTER 1

INTRODUCTION

Real-time systems (RTS) are systems that, apart from functional correctness, also have timing guarantees. There are two types of RTS requirement: soft and hard. In soft real-time systems (SRTS), tasks are allowed to miss deadlines but must put best efforts to meet the deadline. The penalty for missing the deadline in SRTS is not dire: it only causes an inconvenience at most. However, for hard real-time systems (HRTS), the consequence of missing the deadline is severe. Reason for such is that HRTS are used in safety critical devices (e.g. pace-makers, car anti-lock break systems, and control systems in a nuclear power plant). To prove that a given HRTS has hard deadline guarantees, the scheduling algorithm is designed to be predictable; two of the thoroughly analyzed HRTS scheduling algorithms are rate-monotonic scheduling (RMS) [1] and earliest deadline first (EDF). However, the property can be a double-edged blade.

The predictability of task executions can be used as a security feature where any deviation from the predicted norm can be considered anomalous. For instance, there exists anomaly detection models that uses distribution of system-call [2] and memory access patterns [3]. Because the scheduling is deterministic, there can be small and finite sets of such models that can define the behavior of the whole system.

On the other hand, it can leak information. The RMS algorithm was shown to have side-channels in multi-level secure systems due to resource sharing [4] which led to proposed modifications to RMS [5]. Furthermore, Chen et al [6] showed that under RMS, an adversary can reconstruct the exact scheduling behavior by knowing some properties of the task-set.

Side-channel attacks are a huge concern because RTS traditionally had security through obscurity. Therefore, the biggest obstacle was knowing the hardware and gaining any access to the system. However RTS that primarily relied on such mechanisms have recently shown to be vulnerable due to the

rise of common-off-the-shelf components, Internet of Things (IoT) and remote control of RTS via the Internet (e.g. SCADA) or other channels. Therefore the attack surfaces for RTS are larger than ever and the adversary being aware of the context of task execution in the system can worsen the attack.

Other attacks on the RTS are also practical in modern times. In 2015, there was a vulnerability where an adversary managed to wirelessly hijack a Jeep Cherokee [7]. Similarly, certain models of school bus and semi-tractors were also shown to be prone to hijacking but required physical access [8]. Such vulnerabilities compromise the safety of the driver and the passengers. Even though Chrysler, parent company of Jeep, patched the vulnerability, an adversary can still hijack with physical access [9] therefore the automotive RTS security still is a serious concern.

The Stuxnet worm showed that knowing the context of task execution and knowing the occurrence of certain events were crucial in carrying out a successful attack [10]. The worm first gained foothold into its target system via USB storage and remained dormant until it detected that a SCADA controller was trying to control certain equipment. When the event occurred, the worm would perform man-in-the-middle attack by feeding malicious control values to the target equipment replaying the legitimate controls back for the monitor to see. Such knowledge meant that the worm had very little footprint in the system.

With the recent advancement of deep learning I intend to investigate its impact in this field. Therefore, the contribution of this thesis is two-fold. First, we *propose an architecture that will outline the overall process to predict the future task executions given sequences of idle and busy intervals*. Second, we *analyze the performance of deep neural network models to investigate the factors that can affect it*.

The thesis is outlined as follows. In Chapter 2, we discuss relevant work to provide the context of this paper and factors that distinguish this paper. In Chapter 3, we overview the building blocks of deep network to understand the proposed third stage model. In Chapter 4, we describe the overall process of the architecture and formalize the the problem and the objective which each model in the stage tries to solve. In Chapter 5, we describe the setup and configuration for the experiment. In Chapter 6, we discuss the result and analyze it. In Chapter 7, we discuss the limitations and the potential future work and conclude in Chapter 8.

CHAPTER 2

RELATED WORK

Chen et al extended their work ScheduleLeak [6] to show the practicality of a side channel attack on RTS by demonstrating simulated attack on a UAV with FreeRTOS [11]. The busy intervals are observed through the preemptions of an observer task (the compromised task with the lowest priority). The attack demonstration showed that the time cost of measuring the busy interval is small enough to be practical and the cache footprint it leaves behind is small enough to use cache as a side channel. This is assuming that the adversary has a priori knowledge about the task-set.

One method to mitigate the problem is to randomize the task execution to obfuscate the adversary observation. Yoon et al [12] proposed a method of randomizing the task execution trace wile guaranteeing the hard real-time deadlines for RMS algorithm. That method exploits the idle intervals and tries to distribute the task execution as evenly as possible to maximize the observation entropy.

In this work, we try to establish a baseline performance (for launching the side channel attacks) without the a priori knowledge using deep learning.

There exists work that uses deep learning in RTS. Nomani et al [13] demonstrated applying feed-forward neural net to predict the the behavior of application context switching to reduce the side-channel attack in multi-core processor using hardware counters. The work depends on the existence of information leakage due to shared functional units. The adversary application's performance degradation is measured to infer the information from the victim application.

Our work differs for two reasons. First, we are using ML for offense. Therefore, the attack model is smaller and there is more coarse information to work with (e.g. it does not have access to hardware counters to make predictions). Second, our work focuses on single core real-time systems (which is the most common type of such systems). The side-channel due to shared functional

units cannot exist in such systems.

CHAPTER 3

BACKGROUND

In this section, we first formally define layers that compose a deep neural network and their properties.

3.1 Feed Forward Neural Nets

Standard feed forward neural nets forms prediction as follows:

$$y = \sigma^{(l)}(A^{(l)} \dots A^{(2)} \sigma^{(1)}(A^{(1)}x + b^{(1)}) + b^{(2)} \dots + b^{(l)})$$

Where $A^{(i)}$ is a matrix of size $\mathbb{R}^{nodes(i+1)} \times \mathbb{R}^{nodes(i)}$ that represents vectors of weights that fully connects an output layer i to layer $i+1$ and $b^{(i)}$ is the bias. The $nodes(i)$ is the number of neurons at layer i where the input is layer 1 and the output is layer l. Each $\sigma^{(i)}$ is an activation function (e.g. sigmoid, tanh, linear) for layer i. The last activation function is usually softmax for classification models and linear for regression models. The parameters are tuned using gradient descent.

3.2 Recurrent Neural Nets

The recurrent neural net (RNN) is known to capture the temporal properties of time series observations [14]. It differs from feed forward since the previous output gets fed back to the model as current input.

3.2.1 Vanilla RNN

Vanilla RNN is formalized as follows:

$$output_t = tanh(\langle W, \begin{pmatrix} input_t \\ output_{t-1} \end{pmatrix} \rangle)$$

The parameters, $W \in \mathbb{R}^{nodes(output)} \times \mathbb{R}^{nodes(input)+nodes(output)}$, are tuned to capture the dependency between the previous time slot and the current time slot. However, the vanilla RNN is prone to vanishing gradient [15] and fails to capture the long term dependencies well. It is often infeasible to backpropagate to the beginning of the dataset every time due to long training time. Therefore, truncated backpropagation through time [16] is used in practice that unfolds the RNN finite number of steps in time before applying backpropagation.

3.3 Long Short-Term Memory

LSTM [17] is an improvement over RNN. LSTM keeps track of long term dependencies in its internal cell state and is not prone to vanishing gradient like the vanilla RNN.

$$\begin{pmatrix} f \\ i \\ o \\ g \end{pmatrix} = \begin{pmatrix} sigm \\ sigm \\ sigm \\ tanh \end{pmatrix} W \begin{pmatrix} input_t \\ output_{t-1} \end{pmatrix}$$

Each element of the resulting matrix is a gate: f is the forget gate, i is the input gate, o is the output gate, and g is the candidate cell gate. The forget gate regulates the amount of previous cell state gets passed to the current cell state. The input and candidate cell gate determines how much of the current input affects the current cell state. And the output gate determines how much of the input gets directly incorporated in the output. Formally,

$$c_t = f \odot c_{t-1} + i \odot g$$

$$output_t = o \odot tanh(c_t)$$

Where c_t is the LSTM cell state at time t and $\odot$ is an element wise multiplication.

Similar to LSTM, GRU has internal cell state and keeps track of the previous output. The cell state allows the GRU and LSTM to capture long term dependencies. Unlike LSTM, instead of *forget gate* and *input gate* being separate, GRU combines the two into a single *update gate*. The performance between LSTM and GRU are similar [18] but GRU has lesser parameters to tune due to the simplification therefore faster training time.

3.3.1 Bidirectional

Making RNNs bidirectional allows the neural nets to capture both the forward time properties and backward time properties [19]. The bidirectionality works by having two separate RNNs where one reads the input in ascending time and the other reads input in descending time. The output of the both RNNs are then concatenated for the next layer to use.

CHAPTER 4

ARCHITECTURE

Our proposed architecture is a pipeline composed of three stages: task-set size inference, task to busy interval mapping and future task prediction. The intuition behind this construction is to infer general variables (i.e. size of task-set) first then infer the specific variables (i.e. future task execution sequences) based on the previous inference. And to infer the general variables, we use the synthetically generated task-sets. The architecture is shown on Figure 4.1. Notice that the first two stages of the architecture depends on a random task-set generator. The distributions of the outputs from such a task-set generator is discussed in chapter 6. In this section, we will formalize the problem that each stage tries to solve. Throughout this chapter, we will use the notation defined in Table 4.1.

Table 4.1: Notation definition

Symbol	Definition
$\mathbb{Z}_+$	Non-negative integer
$X_i^{(j)}$	j^{th} input data for stage i
$Y_i^{(j)}$	j^{th} output data for stage i
M_i	Stage i model
n_i	Size of dataset at stage i
$\hat{X}$	Observed target input
$\hat{Y}$	Model target prediction of stage i
θ	Generator parameters
$\mathcal{G}$	Generator distribution
$\mathcal{F}$	Target distribution

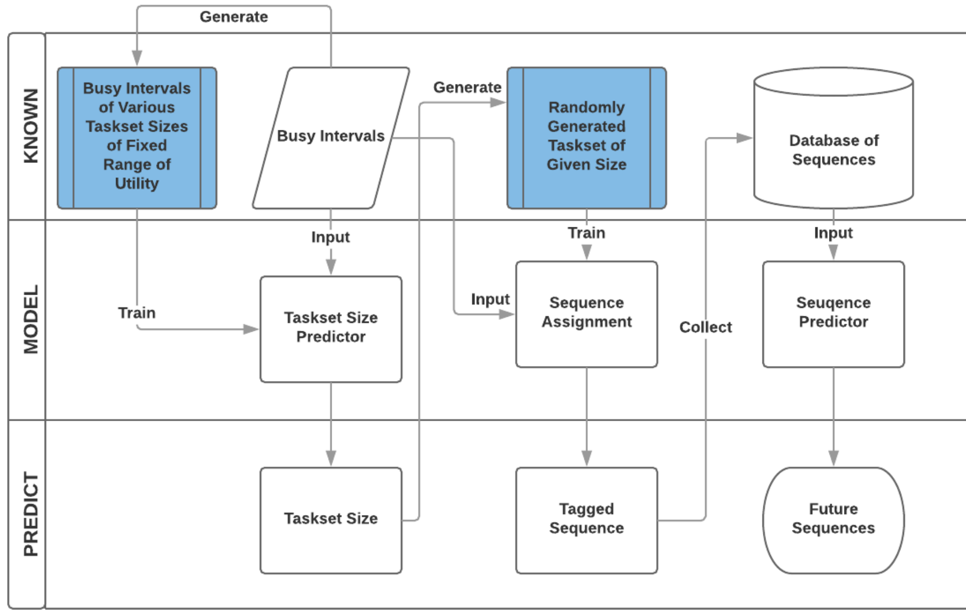

Figure 4.1: Our proposed architecture. The arrows represent dependencies. The blue shaded squares represent collections of randomly generated data based on the information collected from the dependency arrow.

4.1 Adversarial Model

The adversarial model for this architecture is the same as the one proposed by Chen et al [6] and [11], but with three differences. First, a priori knowledge about the task is not assumed; therefore the property of the task-set must be inferred. Second, the architecture assumes that it is able to observe busy and idle intervals without an observer task. For instance, the architecture could be provided execution traces as input. Third, the attack is considered successful if it manage to predict all tasks with high precision and recall.

4.2 Task-set size inference

The objective of the first stage is to generate large samples of random task-sets for the model to learn the distribution of busy intervals for different task-set sizes. This approach follows an intuition and makes an assumption that when the utilization is given and the task-set size is bounded, there exists observable unique structure for each task-set size.

Formally, the training dataset is composed of $(X_1^{(1)}, Y_1^{(1)}), \ldots, (X_1^{(n_1)}, Y_1^{(n_1)}) \sim \mathcal{G}(\theta)$ where $\mathcal{G}$ is the output distribution of the task-set generator. Each $(X_1^{(i)}, Y_1^{(i)}) \in \mathbb{Z}^T \times \mathbb{Z}_+$ are alternating sequences of busy and idle intervals of length T and its corresponding task-set size sampled i.i.d.. A positive value in $X_1^{(i)}$ represents a busy interval duration while a negative represents an idle interval duration. Parameter θ is the bound for the total utilization, task-set size, task period and offset. The task of the model is to predict the task-set size $\hat{Y}_1$ given an observed sample $\hat{X} \sim \mathcal{F}$ where $\mathcal{F}$ represents the true target joint distribution of $(\hat{X}, \hat{Y})$.

Any extra information can be used to skew $\mathcal{G}$ towards $\mathcal{F}$ to enhance the accuracy. For instance, knowing the execution time and period of a single task significantly reduces the sample space of the task-set generator. Therefore, assuming that the target task is of higher priority and system is running RM, the generator only needs to consider the numbers of tasks that has higher priority than the known task.

4.3 Task to busy interval mapping

Given that size of a task-set is known, this stage of the pipeline maps each task to each ticks of the busy intervals. The generator only needs to generate task-sets of a particular size. The intuition is that the model should recognize certain portions of the busy intervals to be more periodic than the other intervals and recognize that the most periodic portions are due to highest priority tasks.

The model, M_2, is trained on $(X_2^{(1)}, Y_2^{(1)}), \ldots, (X_2^{(n_2)}, Y_2^{(n_2)}) \sim \mathcal{G}(s, \theta)$ where $s = \hat{Y}_1$. The $X_2^{(i)}$ in $(X_2^{(i)}, Y_2^{(i)}) \in \{0, 1\}^T \times \mathbb{Z}_+^T$ is a binary vector that represents idle (represented by 0) and active (represented by 1) ticks of length $T \in \mathbb{Z}_+$. The Y_i represents the mapped vector where the value at a given tick represents the priority of the task executing. And s is the known task-set size: higher the value of s, higher the priority. θ represents the remaining parameters of the generator.

4.4 Future task prediction

The mapped busy intervals and idle intervals are then stored and used to train the predictor model. The objective of the model is to predict which task is executing c ticks in the future given a sequence of tasks for t consecutive ticks.

The model is trained on $(X_3^{(t-1:j)}, Y_3^{(j)})$ for some value of j where $X_3 = \hat{Y}_2$ and $Y_3^{(j)} = X_3^{(j+c)}$. The problem is similar to character prediction in natural language processing (NLP). However unlike NLP, the prediction offset into the future is large. Therefore it is expected that models that work well in the NLP scenario to not perform as well in this setting.

CHAPTER 5

EXPERIMENTAL SETUP

In this section, we prose a model for the third stage of the architecture and investigate which variables can affect the model's performance. The experiment was performed on Intel Core i7 6700k, 16GB memory, and Geforce GTX 1080Ti using Keras [20] with Tensorflow [21] as back-end. The objective of the experiment is to test to see if the architecture is viable for determine the methods and variables that could affect the performance of the deep learning model. Because there exists viable models for problems similar to the first two stages of the architecture, we will evaluate the plausibility by proposing a model for the last stage of the pipe line that can perform well. We only consider the task-sets with size three unless otherwise specified. We chose the size three to simplify the priority variables that can affect the performance.

5.1 Problem Formulation

For the experiment, we formulated the problem as a sequential prediction problem. In the sequential approach, the model is given t past consecutive ticks and the objective is to predict the task that is executing c ticks into the future. Therefore, the output vector is an one-hot-vector of size s, the last activation function is softmax (normalization using exponential function), and the loss function is weighted categorical cross entropy. The size s is equal to one added to the size of the task-set. The parameters specified in Table 5.1.

The weighted version of cross entropy was used to prevent the model from biasing towards the prior. It is defined as,

$$wcc(y, \hat{y}) = \sum_{i} \frac{1}{prior(y_i)} \mathcal{L}(y, \hat{y})$$

Table 5.1: Formulation Parameter

Symbol	Value
c	10000 ticks
t	30 ticks
s	4

The variables are defined in Table 5.2

Table 5.2: Weighted Cross Entropy Symbols

Symbol	Definition
$\mathcal{L}$	Cross entropy before the sum
y	True label one-hot vector
$\hat{y}$	Predicted label vector
$prior(y_i)$	prior distribution of label i

5.2 Model

In all of our experiments, we used two layered stacked LSTM with the cell state size of 64. We found that the stacked LSTM provided the best performance out of various models we have tried. The parameters used for the training is specified in Table 5.3. The gradient descent optimizer, Aadam [22], is commonly used in practice. The parameters were chosen on the basis of providing the good prediction performance with low training time.

Table 5.3: Training Parameters

Parameter	Value
Batch size	400
Epochs	50
Optimizer	Adam

5.3 Features

To investigate how feature engineering can affect the performance, we have tested two models of same neural architecture but different input vector: one with sequence only input the other with sequence + engineered features. Our engineered feature is the value of ticks since the last execution of corresponding task or the value of ticks since the start of the execution if the task is in the middle of execution.

5.4 Utilization

In this experiment, we investigate how the utilization can affect the model performance. Therefore we will compare three groups of task utilization and four types of distribution within each group. Each group is defined as follows:

1. High utilization: Group of task-sets with total utilization in $[92.5\%, 97.5\%]$.
2. Medium utilization: Group of task-sets with total utilization in $[47.5\%, 52.5\%]$.
3. Low utilization: Group of task-sets with total utilization in $[2.5\%, 7.5\%]$.

Each type of distribution is defined as follows:

1. High heavy: The high priority task takes at least 50% of the total utilization.
2. Medium heavy: The medium priority task takes at least 50% of the total utilization.
3. Low heavy: The low priority task takes at least 50% of the total utilization.
4. Not heavy: The no single task takes at least 50% of the total utilization.

5.5 Task-set Size

To investigate the effect of size of a task-set, we compared two task-sets with different size but similar utilization and utilization distribution. In particular, we compare the performance between task-set of size three and task-set of size 11.

CHAPTER 6

ANALYSIS OF RANDOM TASK-SET GENERATION

The distribution of the task-set generator (used to generate synthetic task-sets for our inputs) can impact the real world performance of our algorithms since the first two stages of the architecture depends on it. In this section, we will discuss the output distribution of the task-set generator. The user must set bounds on the range of period and offset of generated task. The parameters used in the experiment is specified in Table 6.1 where $tick$ is the size of the bin used to discretized time, p is period, and o is the offset.

Table 6.1: Default Generator Parameters

Parameter	Value
$tick$	$100\mu s$
p_{max}	$100ms$
p_{min}	$10ms$
o_{max}	$3s$
o_{min}	0

The task-set generator that we used has the pseudo-code (as specified in Algorithm 1) that was created and used by Chen et al [6] [11]. Essentially, the algorithm first generates a set of equal utilization tasks with the specified size in the parameter. Next the algorithm mixes the utilization between tasks by randomly selecting two tasks and moving one percent of the total utilization from one task to the other. Then the the algorithm randomly selects periods and offsets from the specified range and calculates the execution time.

```
    parameter: p_{min,max}, o_{min,max}, u_{min,max}, size
 1  total_utilization ← uniform_random(u_{max},u_{min});
 2  Γ ← equal_utilization_task_set(size);
 3  for counter ← 0 to 100 do
 4  |   task1, task2 ← pick two random tasks in Γ;
 5  |   task1.utilization ← task1.utilization + 0.01 total_utilization;
 6  |   task2.utilization ← task2.utilization − 0.01 total_utilization;
 7  end
 8  for task ∈ Γ do
 9  |   task.period ← uniform_random(p_{max}, p_{min});
10  |   task.execution ← task.period · task.utilization;
11  |   task.offset ← uniform_random(o_{max}, o_{min});
12  end
13  if not_schedulable(Γ) then
14  |   goto 1;
15  end
16  return Γ;
```

Algorithm 1: task-set Generation Algorithm

This algorithm generates task-sets with the distribution shown in Figure 6.1 with the parameters $u_{min} = 0$ and $u_{max} = 1$ and was sampled 100000 times. The test was conducted through worst case response time analysis. A Hyperbolic bound [23] can also be used for the schedulability test that provides better bounds than the traditional utilization bound test [1]. However, these methods can result in sharp cut-off in the in the histogram distribution. The purpose of plotting is to visualize the overall distribution of the task-set.

Ideally, the distribution of the task-set should be uniform. Generally, we want to avoid the case where the global shape of the distribution looks Gaussian or some screwed distribution across the parameterized intervals.

Although the distribution is not ideal, we see that the distribution becomes flat after 0.1 utilization and fluctuates some time until it reaches 0.8 utilization. After around 0.83 utilization, the distribution falls off since there are lesser idle times that makes schedulability of randomly chosen sets of tasks hard.

To visualize the the task-set utilization distribution for high, medium, and low utilization task-sets where the parameter $p_{min,max}$ was set accordingly, we

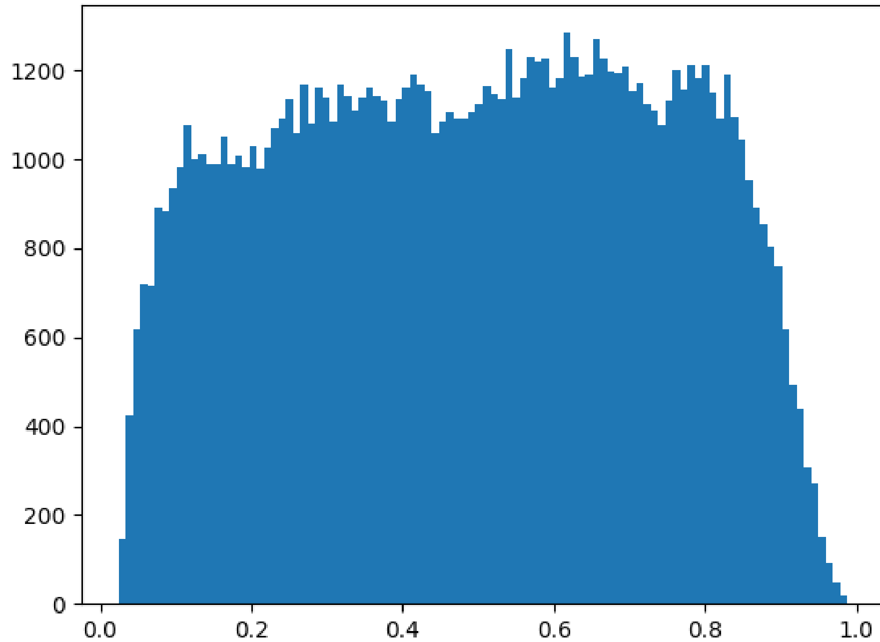

Figure 6.1: Histogram of task-set distribution of utilization. The x-axis is the utilization and the y-axis the number of tasks in the given bin. The x-axis is split into 100 bins.

sampled 5000 task-sets from each group. Figure 6.2 shows the distribution for three groups of utilization. Notice that the task-sets seem to follow the distribution from Figure 6.1. The only difference is that the distribution has many tall but short spikes which is not a problem. This is probably due to not enough sampling.

The distribution of individual tasks sets are shown in Figure 6.3. The shape of the distribution seem to be Gaussian with mean centered around $\frac{total_utilization}{number of tasks}$ regardless of the priority. An interesting observation is that the utilization variance is higher as total utilization increases. This could imply that inferring high utilization task-sets require more sampling to fully capture the distribution.

Another interesting observation is the per-task execution distribution in Figure 6.4 and period distribution in Figure 6.5. As we can see, the tall but short spikes are visible in the period distribution similar to the overall task-set utilization distribution in Figure 6.2 but the resulting execution distribution

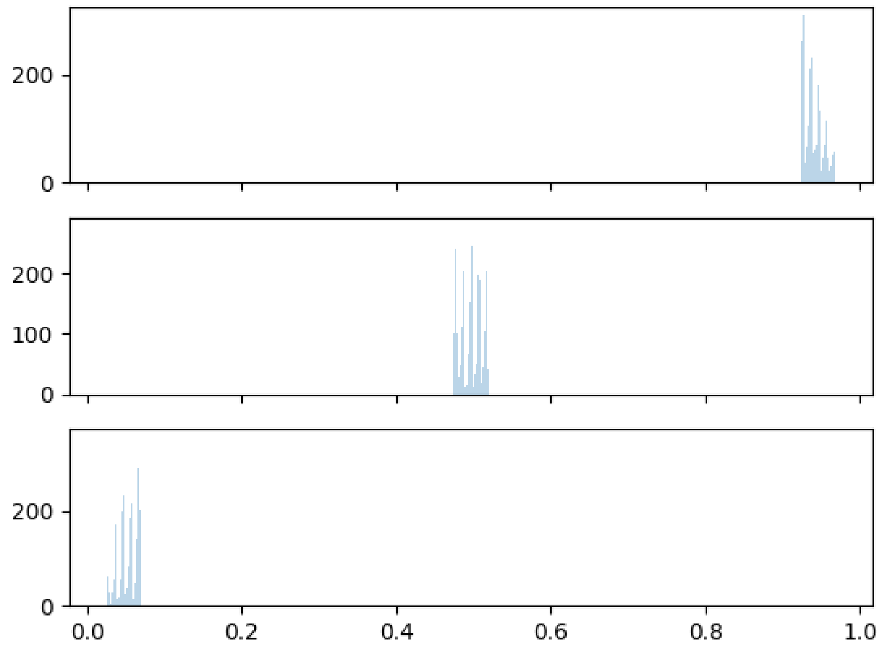

Figure 6.2: Histogram of task-set distribution of utilization for high, medium, low utilization task-sets respectively from top to bottom. The x-axis is the utilization and the y-axis the number of tasks in the given bin. The x-axis is split into 50 bins.

is fairly smooth. This indicates that the tall but short spikes are artifacts of uniform random sampling function in the implementation since the task-set utilization and each task period depend on it.

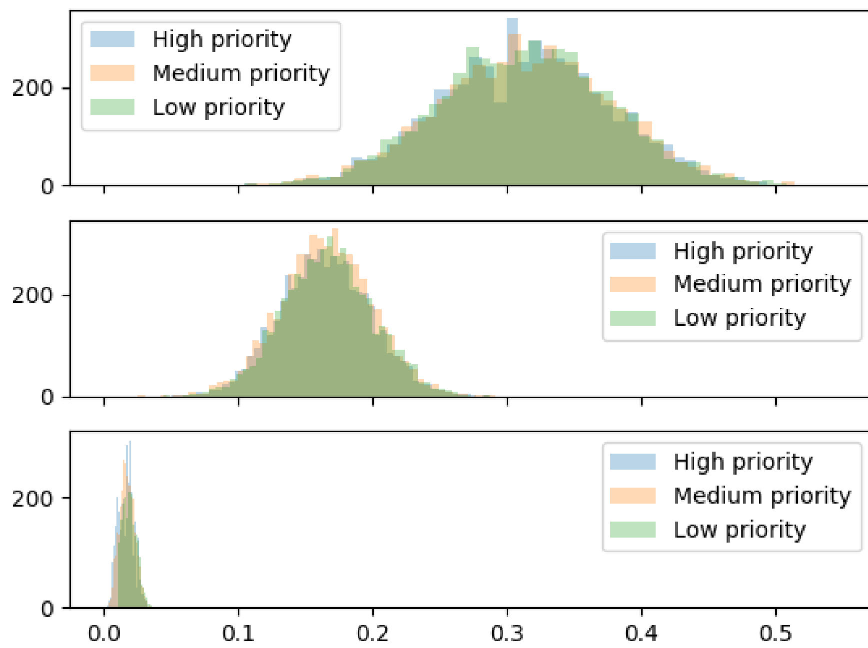

Figure 6.3: Histogram of per-task distribution of utilization for high, medium, low utilization task-sets respectively from top to bottom. The x-axis is the utilization and the y-axis the number of tasks in the given bin. The x-axis is split into 50 bins.

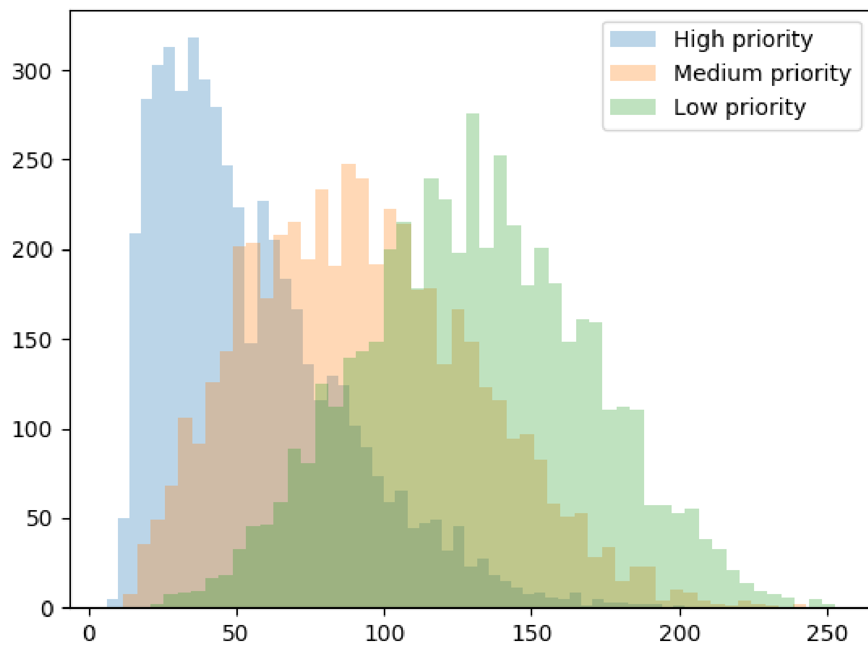

Figure 6.4: Histogram of per-task distribution of execution for medium utilization task-sets. The x-axis is the execution time and the y-axis the number of tasks in the given bin. The x-axis is split into 50 bins.

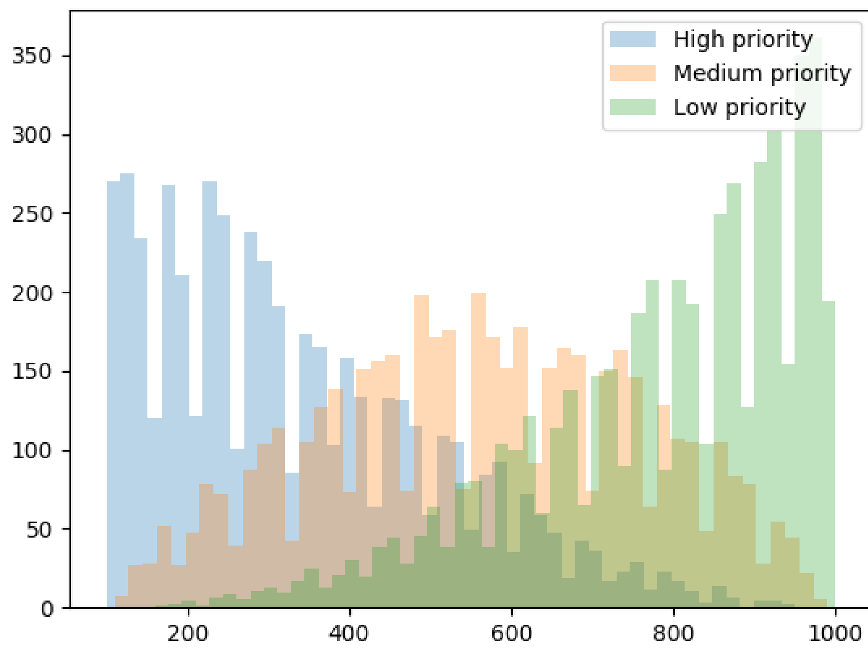

Figure 6.5: Histogram of per-task distribution of period for medium utilization task-sets. The x-axis is the period and the y-axis the number of tasks in the given bin. The x-axis is split into 50 bins.

CHAPTER 7

TASK PREDICTION RESULTS

The summary of the results for investigating the effect of utilization and feature engineering is shown in Table 7.1. The number of samples used for each slot in Table 7.1 is shown in Table 7.2. Training a single model takes about 15 minutes.

Table 7.1: Result in F1 Score

Sequence + Timer				
Total Utilization	Distribution	High Priority	Medium Priority	Low Priority
high	High Heavy	0.9970	0.8914	0.83267
	Mid Heavy	0.9916	0.9769	0.8734
	Low Heavy	0.9890	0.9394	0.9270
	Not Heavy	0.9969	0.9592	0.9035
Med	High Heavy	0.9931	0.9276	0.8437
	Mid Heavy	0.9804	0.9836	0.8565
	Low Heavy	0.9860	0.9484	0.9596
	Not Heavy	0.9878	0.9500	0.9181
Low	High Heavy	0.8793	0.5398	0.6429
	Mid Heavy	0.5787	0.8114	0.5066
	Low Heavy	0.53819	0.5891	0.7309
	Not Heavy	0.7323	0.7183	0.6548
Sequences Only				
Med	Not Heavy	0.4700	0.3975	0.4429

Table 7.2 shows the total number of task-sets that was used for the experiment.

Table 7.2: Sample task-set Count

Sequence + Timer		
Total Utilization	Distribution	Sample Count
high	High Heavy	22
	Mid Heavy	17
	Low Heavy	12
	Not Heavy	25
Med	High Heavy	30
	Mid Heavy	28
	Low Heavy	33
	Not Heavy	25
Low	High Heavy	11
	Mid Heavy	20
	Low Heavy	72
	Not Heavy	25
Sequences Only		
Med	Not Heavy	25
Total		345

7.1 Effect of Total Utilization and its distribution on Performance

Based on Table 7.1, the F1 score of each task is significantly lower when the total utilization is low compared to medium and high total utilization. This result implies that the model performs better when it observes more of the event it is trying to predict. The implication is further supported by observing the shift in task-set utilization distribution while keeping the total utilization constant especially when the task-set total utilization is low. However, when the total utilization is high enough, the shift in the distribution does not seem to affect task F1 score.

7.2 Effect of Feature Engineering on Performance

To enhance the performance of the model, we explicitly embedded the "time since last execution" for each task into the feature. We will refer to this engineered feature as "timer" feature. As a result, the feature vector becomes twice as long for each time step where the first half of the vector is the one-hot vector representation of the task and the second half is the time since

each task last executed.

To compare the effect of the feature engineering, the sequence only input model was compared to sequence + timer input model. As shown in the Table 7.1, the timer feature boosted the F1 score of medium and low priority tasks significantly. This boost is likely due to offloading history tracking away from the LSTM.

In this scenario, we believe when LSTM observes the same task for a long time, it may not know the significance of the current observation and decides to forget the current observation. As a result, when the LSTM observes a new task, it fails to capture the dependency between the new observation with the old. Based on our quick testing, mitigating this effect by increasing the number of layers does not seem to improve the performance (it either over fits or under fits).

7.3 Effect of task-set Size on Performance

The model's performance degrades as task-set size increases. With the same previously described configuration, we managed to obtain around 90% accuracy on a task-set of size 11 when only considering predictions with 99% confidence and the model was confident around 47% of the time. Therefore, a more expressive model and longer trace observations are required to deal with high task-set size. We found that with small task-sets, it is better to use simple models since more expressive models lead to over-fitting (i.e. high training accuracy but low testing accuracy).

7.4 Other Trends

We observed that the model tends to predict high priority tasks accurately (as shown on Fig 7.1). The high priority tasks are always have near perfect F1 score regardless of its task utilization. This leads us to believe that it is likely due to short period of the higher priority tasks and because high priority tasks cannot be preempted by another task. However as shown on Fig 7.2, when the total utilization is small and the high priority tasks has small utilization compared to other tasks, then task utilization becomes the

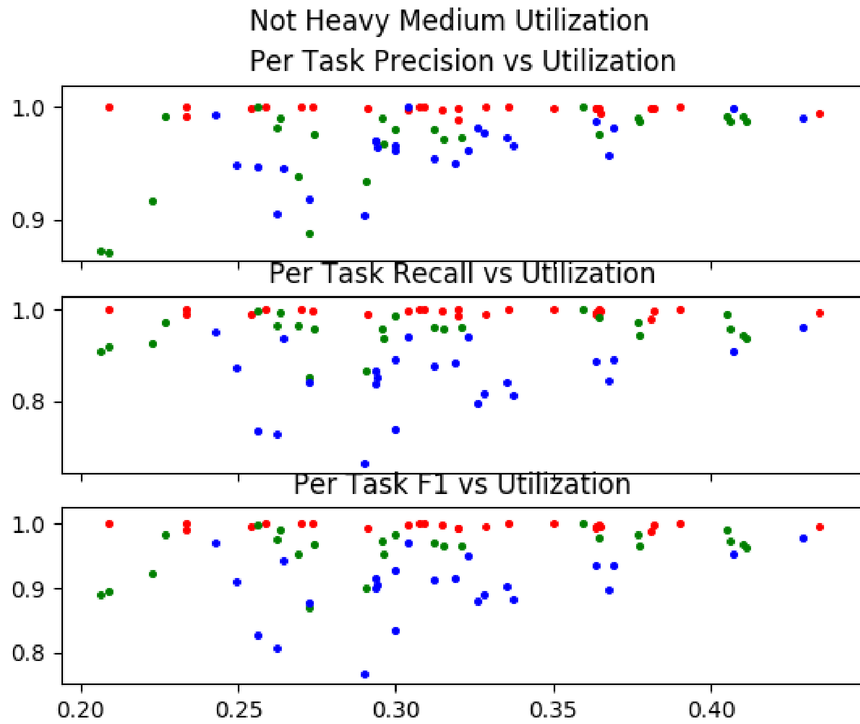

Figure 7.1: The plot shows per-task performance of "not heavy" medium utilization task-set. The red dots are the high priority tasks. Green dots are medium priority tasks, and blue dots are low priority tasks.The x-axis is utilization, the y-axis is performance metric.

determining factor for predicting tasks.

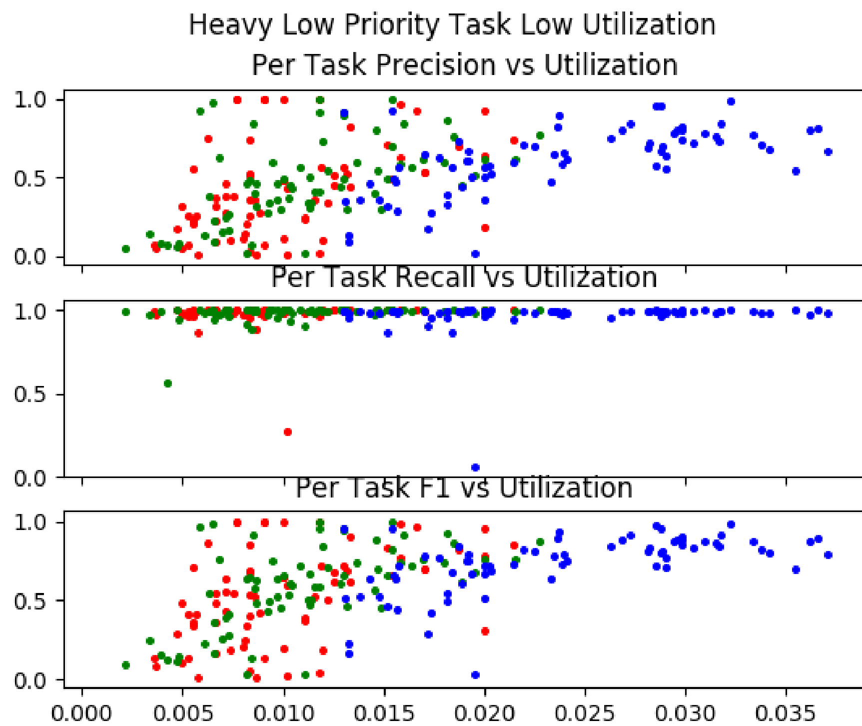

Figure 7.2: The plot shows per-task performance of low heavy low utilization task-set. The red dots are the high priority tasks. Green dots are medium priority tasks, and blue dots are low priority tasks. The x-axis is utilization, the y-axis is performance metric.

CHAPTER 8

LIMITATIONS AND FUTURE WORK

8.1 Performance Under Randomized Traces

Currently, the architecture only considers when the target system uses the rate-monotonic scheduling algorithm. And under the current architecture, task shuffling [12] could make the task-set size inference more difficult due to the introduction of random busy intervals. In such cases, another covert channel may be needed. The performance evaluation under these setting will be considered for future works.

8.2 Finishing the First Two Stages

For the first stage of the pipeline, We currently have a prototype model using bidirectional LSTM . Our preliminary experiment on the model shows that the model has about 80% accuracy when trying to distinguish the interval traces of five different task-set sizes. We believe that we can achieve higher performance.

Although we do not have an implementation for the second stage model, we realized that the problem of mapping tasks is similar to colorizing the grayscale pictures (an active area of research in computer vision). Mapping busy interval to tasks can be considered as "colorizing" busy interval. The standard practice in computer vision is to use convolutional neural nets and improvements are made such as [24], [25], and [26]. We will explore these kind of models for the second stage of the pipeline for our future work.

CHAPTER 9

CONCLUSION

Real-time systems are designed to have predictable behavior to provide safety guarantees. However, an adversary can exploit this property to launch timing inference side-channel attacks or even cause instabilities. We propose an architecture to predict future execution of tasks from only observing sequences of busy and idle intervals. The architecture is composed of task-set size inference, task to busy interval mapping, and task prediction. We show that the output distribution of the task-set generator (that the architecture depends upon) is uniform enough with respect to utilization in task-set scope and is Gaussian with respect to task utilization for per-task cases. The preliminary results show that there are corner cases where the task prediction model does not perform well. However, we show that the model performs well even when the task-set is of size 11. We plan to complete the implementation of the rest of the architecture and make it robust against randomized task execution traces.

REFERENCES

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[2] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha, "Learning execution contexts from system call distribution for anomaly detection in smart embedded system," in *Internet-of-Things Design and Implementation (IoTDI), 2017 IEEE/ACM Second International Conference on*. IEEE, 2017, pp. 191–196.

[3] M.-K. Yoon, S. Mohan, J. Choi, and L. Sha, "Memory heat map: anomaly detection in real-time embedded systems using memory behavior," in *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 2015, p. 35.

[4] J. Son et al., "Covert timing channel analysis of rate monotonic real-time scheduling algorithm in mls systems," in *Information Assurance Workshop, 2006 IEEE*. IEEE, 2006, pp. 361–368.

[5] S. Mohan, M. K. Yoon, R. Pellizzoni, and R. Bobba, "Real-time systems security through scheduler constraints," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE, 2014, pp. 129–140.

[6] C.-Y. Chen, A. Ghassami, S. Nagy, M.-K. Yoon, S. Mohan, N. Kiyavash, R. B. Bobba, and R. Pellizzoni, "Schedule-based side-channel attack in fixed-priority real-time systems," Tech. Rep., 2015.

[7] A. Greenberg, "Hackers remotely kill a jeep on the highway-with me in it," Jun 2017. [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[8] M. Wolf and R. Lambert, "Hacking trucks-cybersecurity risks and effective cybersecurity protection for heavy duty vehicles," *Automotive-Safety & Security 2017-Sicherheit und Zuverlässigkeit für automobile Informationstechnik*, 2017.

[9] J.-H. Lee, "Black hat: Knowledge resource for cybersecurity [society news]," *IEEE Consumer Electronics Magazine*, vol. 6, no. 1, pp. 16–19, 2017.

[10] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[11] C.-Y. Chen, A. Ghassami, S. Mohan, N. Kiyavash, R. B. Bobba, R. Pellizzoni, and M.-K. Yoon, "A reconnaissance attack mechanism for fixed-priority real-time systems," *arXiv preprint arXiv:1705.02561*, 2017.

[12] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, "Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE.* IEEE, 2016, pp. 1–12.

[13] J. Nomani and J. Szefer, "Predicting program phases and defending against side-channel attacks using hardware performance counters," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy.* ACM, 2015, p. 9.

[14] J. T. Connor, R. D. Martin, and L. E. Atlas, "Recurrent neural networks and robust time series prediction," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 240–254, 1994.

[15] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[16] M. Boden, "A guide to recurrent neural networks and backpropagation," *the Dallas project*, 2002.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[18] A. Karpathy, J. Johnson, and L. Fei-Fei, "Visualizing and understanding recurrent networks," *arXiv preprint arXiv:1506.02078*, 2015.

[19] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm networks," in *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, vol. 4. IEEE, 2005, pp. 2047–2052.

[20] F. Chollet et al., "Keras," https://github.com/keras-team/keras, 2015.

[21] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin et al., "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[22] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[23] E. Bini, G. C. Buttazzo, and G. M. Buttazzo, “Rate monotonic analysis: the hyperbolic bound,” *IEEE Transactions on Computers*, vol. 52, no. 7, pp. 933–942, 2003.

[24] Z. Cheng, Q. Yang, and B. Sheng, “Deep colorization,” in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 415–423.

[25] R. Zhang, P. Isola, and A. A. Efros, “Colorful image colorization,” in *European Conference on Computer Vision.* Springer, 2016, pp. 649–666.

[26] D. Varga and T. Szirányi, “Fully automatic image colorization based on convolutional neural network,” in *Pattern Recognition (ICPR), 2016 23rd International Conference on.* IEEE, 2016, pp. 3691–3696.