Designing a Robust Controller for Obfuscating a Computer's Power

Raghavendra Pradyumna Pothukuchi[1], **Sweta Yamini Pothukuchi**[1], **Petros G. Voulgaris**[2], **and Josep Torrellas**[1]

[1]**University of Illinois at Urbana-Champaign**
[2]**University of Nevada, Reno**

ABSTRACT

Maya [1] is a new security defense that uses formal control to obfuscate the power side channels of a computer. This technical report describes the procedure to design the formal controller in Maya necessary for reshaping a computer's power. The controller is designed with robust control theory principles and the design process is semi-automated with software like Matlab. This technical report includes the Matlab code required to design the controller.

Keywords: Robust controller, system identification, mu synthesis, power side channels, security, robust control

1 INTRODUCTION

Designing a robust controller has three steps: building a model of the system to be controlled, specifying the controller's design parameters (also known as hyper parameters), and validating the obtained controller for deployment. In the next sections, we describe how to perform these three steps using Matlab.

An overview of robust controllers and how they are applied to computer systems can be understood from [1–3]. A control-theoretic insight into these designs can be found in [4,5]. An earlier technical report from Pothukuchi and Torrellas [6] has additional background material on designing LQG (Linear Quadratic Gaussian) controllers [7] for computer architects.

Some concepts from [6] such as transfer functions are relevant for designing robust controllers, which are the focus of this manual. Therefore, we recommend the readers to review works [1,2,5,6] before using this manual.

Regarding terminology, in [1], we used architectural abstractions to convey the principles of a robust controller, while we must use the control-theoretic specifications to design the controller using Matlab. Nonetheless, we present the connection between the two whenever possible.

Finally, the software for the full Maya framework (excluding the Matlab code in this manual), is hosted at `https://github.com/mayadefense/maya` [8]. This software can be used to mask a computer's power and also to run system identification, which is necessary for the controller's design.

2 MODELING THE SYSTEM

Before we can design a controller for a system, we must obtain a mathematical description (i.e., a model) of how the system's outputs respond to the system's inputs or actuators that the controller will change. In the specific case of Maya, the system has one output (the computer's power) and three inputs (the computer's frequency, percentage of idle activity, and the level of a power balloon application) [1]. Since it is very difficult to obtain a model of a computer system with theoretical calculations, we use black-box system identification [6,9] to obtain this model.

System identification has two steps: (i) collecting input-output data and (ii) fitting standard model structures to the data and selecting the best model.

2.1 Data collection

We must run training applications and collect the values of power observed when changing the inputs. As described in [1], we can use standard applications from PARSEC [10] as our training applications. In Maya, we run each application thrice, recording the impact of one input in each run. In a run, all inputs except the one being studied are held constant at the midpoint of the range of values it can take, and the input under study is varied randomly.

With the Maya software [8], data can be collected by running it in the system identification mode. For example, to record the data when changing CPU frequency, we can use the `Launch.sh` file in the `Scripts` directory of the software as follows:

```
sudo -E --preserve-env=PATH env "LD_LIBRARY_PATH=$LD_LIBRARY_PATH" bash
 ↪  ./Launch.sh --rundir "../Dist/Release/" --options "--mode Sysid --idips
 ↪  CPUFreq" --logdir "<logdir>" --tag "<name>" --apps "<appname>"
```

We only need a few training applications to get our models. After the data is collected, it is imported into Matlab for pre-processing and developing a model between the inputs and power.

2.2 Obtaining a Model

First, we should normalize the experimental data before fitting a model. Normalizing improves model fitting by removing the effects of having different ranges of values that the inputs and outputs can take. To linearly scale down the values of each input and output to be within a -1 to +1 range, we apply the following transformation:

$$Normalized\ value = \frac{Raw\ value - Offset}{Scaling\ factor} \tag{1}$$

where, $Offset = \frac{Maximum\ value + Minimum\ value}{2}$ and $Scaling\ factor = \frac{Maximum\ value - Minimum\ value}{2}$.

After the data is normalized, system identification can be done in Matlab using the “SystemIdentification” app. We can open this app with the command `SystemIdentification`. We will see a window like the one shown in Figure 1. Then, we can import the input-output data series for the training applications. More details on importing data can be found in [6]. After all the data series are imported, they can be merged in the preprocessing option present in the window. The merged data object is placed in the “Working Data” slot of the window.

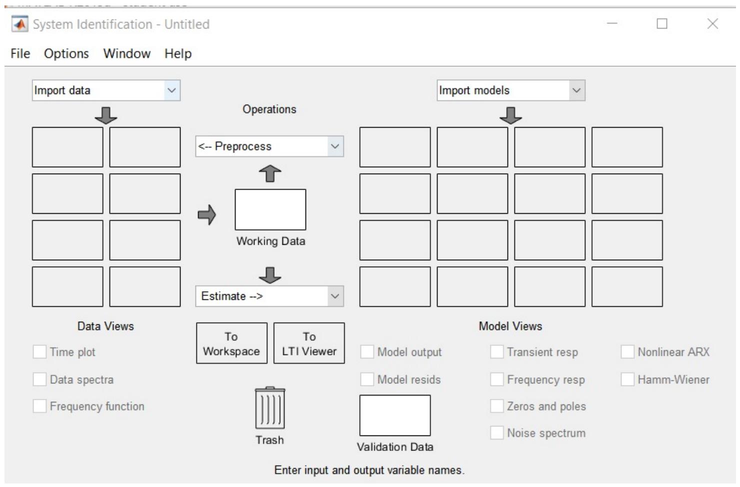
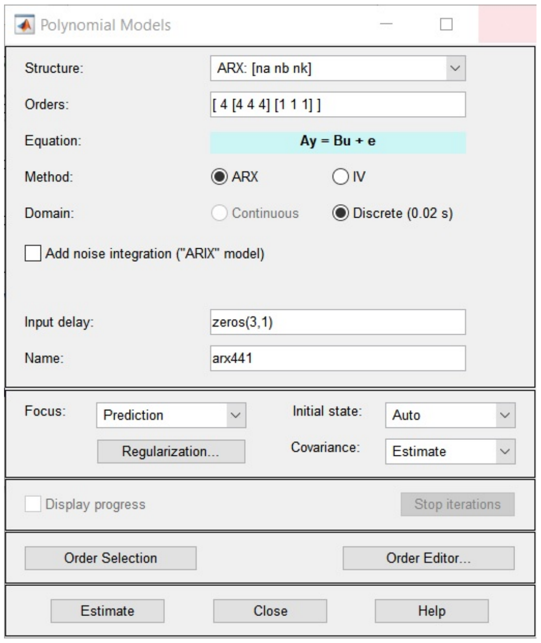


Figure 1. System identification application.



Structure: ARX: [na nb nk]
Orders: [4 [4 4 4] [1 1 1]]
Equation: Ay = Bu + e
Method: [✓ ARX] IV
Domain: Continuous [✓ Discrete (0.02 s)]
Add noise integration ("ARIX" model)
Input delay: zeros(3,1)
Name: arx441
Focus: Prediction
Initial state: Auto
Covariance: Estimate

Figure 2. Polynomial model estimation window.

To begin estimating a model, select the `Estimate` option and choose `Polynomial models...`. A window similar to Figure 2 will appear. Choose a Box-Jenkins model structure, `BJ: [nb nc nd nf nk]`. We will use the data we collected to identify a model of this structure. More information about this model can be found in [6].

In the Box-Jenkins model structure, `nb` has the number of current and past values of inputs that the model should use to calculate the present value of power. `nd` and `nf` correspond to the past values of output that the model uses. We choose that the history include current and previous values of the inputs and outputs i.e., specify `[[2 2 2], 2, 2, [2 2 2], [0 0 0]]` as the `Orders`. The model also identifies the impact of non-deterministic noise on the output values.

We set the *Focus* to *Stability* because our system is inherently stable—for bounded input values, we get bounded values of power consumption. We choose `Initial state:` to be `Zero`. We can add *Regularization* to avoid over-fitting the data. The value we used is 0.6. Once we specify the options, we select `Estimate`, and an identified model will be placed in the system identification window, as shown in Figure 3.

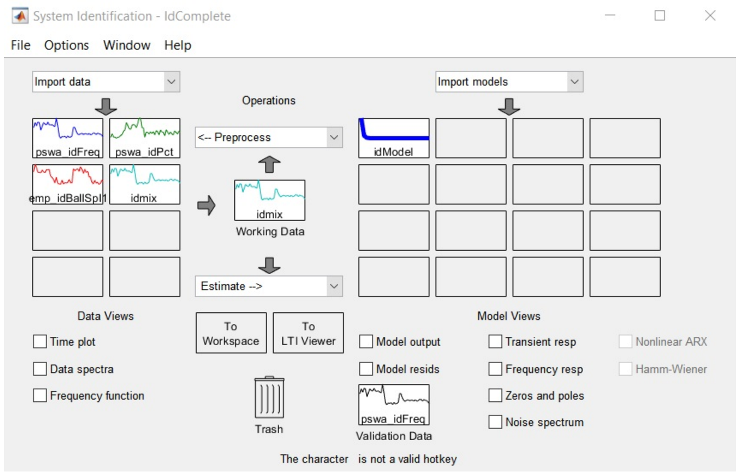


Figure 3. System identification app with the identified model.

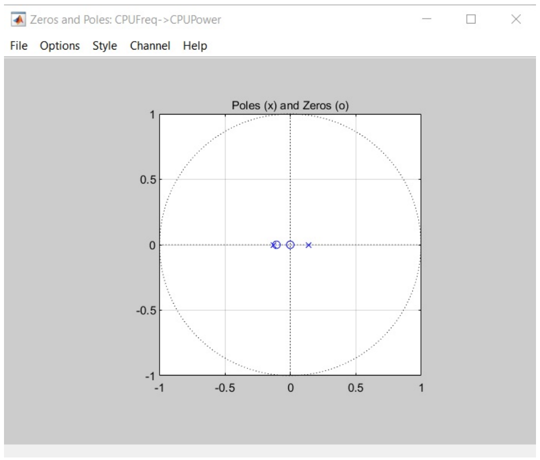


Figure 4. Pole Zero analysis.

We must first check if our identified model is stable. This can be done by checking that all the poles and zeros of the model are within the unit circle (see [6] for more information about these terms). To perform this check, select the `Zeros and poles` checkbox in the app. A window like the one shown in Figure 4 will appear. Select the `Style` option in the menu bar and choose `Unit circle`. Verify that the poles (marked by ‘x’) and zeros (marked by ‘o’) are all inside or on the unit circle. Select other channels from the `Channel` menu in the menu bar to see the pole zero map for the model’s response to all the inputs and noise. The noise channel is shown as `e@CPUPower->CPUPower` in the `Channel` menu. It is necessary to verify that the noise channel is also stable i.e., all the poles and zeros are in or on the unit circle. If the model is unstable, we can change the parameters of the Box-Jenkins model like reducing its dimensions, or use other model structures, or repeat data collection.

After we verified that the model is indeed stable, we can examine the behavior of the model. Click and drag the model on the `LTI Viewer` icon. When the LTI viewer opens, select `Edit` and then `Plot configurations` to show both the step response and Bode plots shown in Figure 5. The step response plot for each input shows how power would change if the input is suddenly changed from the minimum level to the maximum level at time 0—i.e., if there is a step change in the input. The Bode plot shows the change in power when the input is varied at different rates in the range of 0 Hz (i.e., constant input) to 25 Hz (i.e., maximum rate). The maximum rate is 25 Hz because of the Nyquist criterion that it should be half the sampling frequency, which is 50 Hz—corresponding to a sampling interval of 20 ms for Maya. We can right click on the plots to change properties like the units on Bode plots (e.g., changing the rate unit from rad/s to Hz). We can also select the `Characteristics` option on the right click menu to show confidence regions.

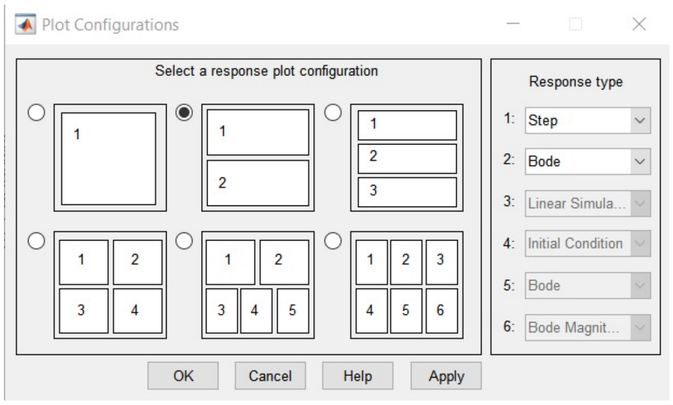


Figure 5. LTI viewer.

Figure 6 shows the impact on power (step response and Bode magnitude and phase plots) for the three inputs, CPUFreq, IdlePct and PBalloon, corresponding to the chip frequency, idle activity level, and the power balloon level, respectively. The shaded regions in Figure 6 correspond to the confidence of the model’s response. The bands are narrower when the model’s response has high confidence and wider if the model has lower confidence. When the confidence is lower, it means that the model was unable to accurately capture the response to an input. This would require specifying a larger uncertainty guardband for the robust controller, or repeating system identification with more data about the impact of that input. We can see that all the responses have high confidence in Figure 6.

From the step response in Figure 6, we can see that the model's response aligns with our intuition—power should go up when CPU frequency or the Balloon power are increased, and power should go down when the idle activity is increased. We can also see this in the Bode plots of Figure 6. The Bode magnitude plot shows the magnitude of response and the phase plot shows the sign of the response. The phase response of power to the idle activity input (IdlePct) is 180°, meaning that power decreases if idle activity increases.

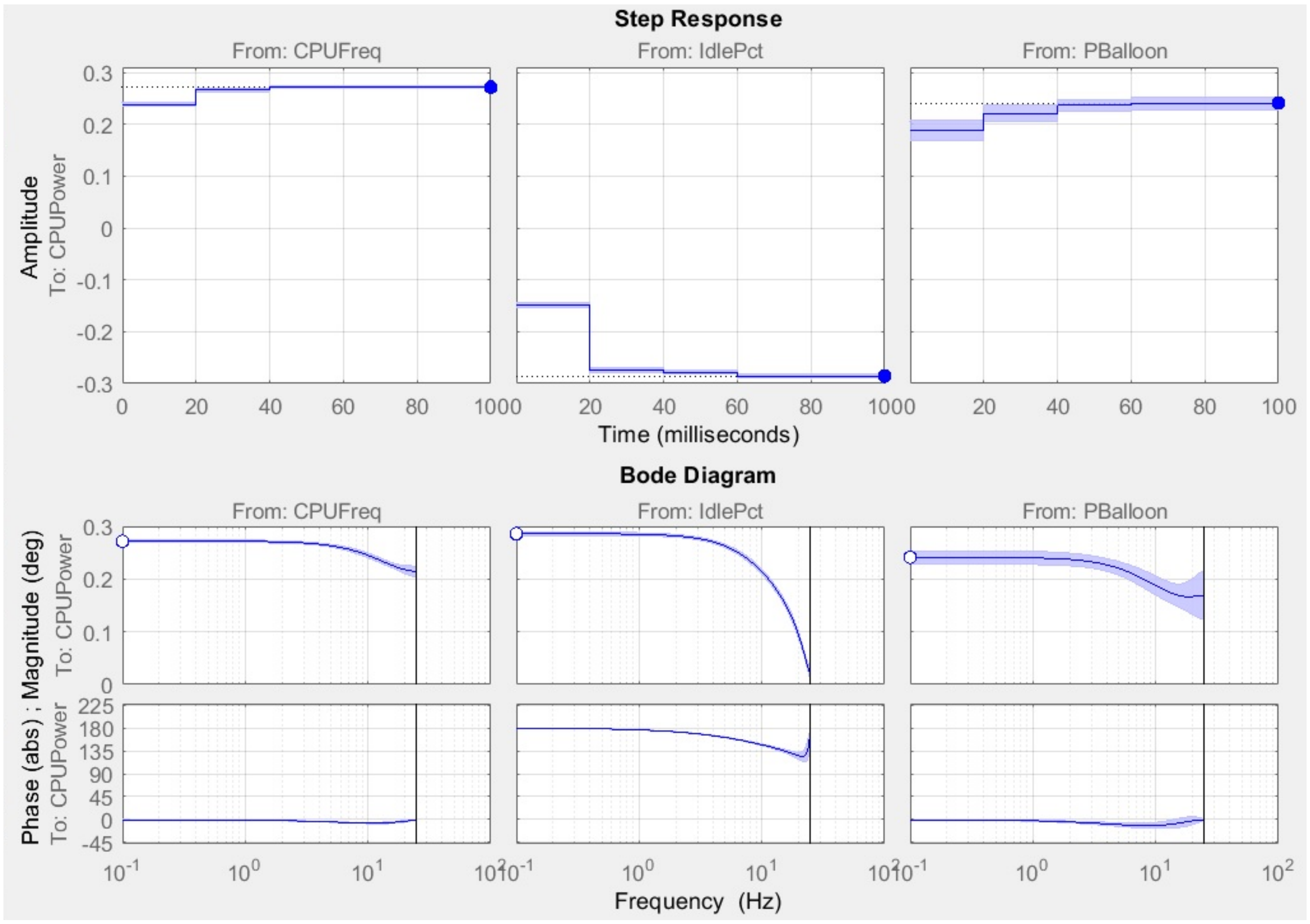


Figure 6. Analysis of the identified model.

From the Bode magnitude plots, we can see that the response of power from idle activity falls off steeply at higher rates. This means that the model predicts the impact of idle activity to be subdued when the input is changed rapidly i.e., the amount by which power is reduced is lower. Indeed, the reasons can be understood by examining the Linux kernel's implementation of this input. When idle activity is changed, the kernel spawns threads that evict the application threads and force the cores to idleness. The kernel tries to maintain the specified fraction of idle cycles measured over a range of time (micro to milliseconds) by periodically checking the fraction of idle and active cycles and re-configuring the idle threads. When the idle activity input is changed rapidly, a new idle activity level is issued to the kernel before the system could fully realize the previous idle activity level. Hence, the impact of idle activity on power is diminished when it is changed too fast—and it is good that our identified model captures this behavior.

We can use other plots in the LTI Viewer, such as Linear simulation to view the response of the models to particular waveforms applied at the inputs and check if the model is satisfactory. If we desire a better or a different model, we can try using other model structures and/or capturing better data.

Once we are confident to choose a model, we click and drag it onto the `To Workspace` icon. This will export the model to the Matlab workspace. In our case, we export the model named `idModel`.

2.3 Model Reduction

Sometimes, the dimensions of the model we identified could be further compressed without loss of fidelity. Choosing a smaller model means that the final controller will also be smaller and require fewer computations to run. To check if we can compress our model, we need to look at the contribution of the different “states” in our model using what is known as Hankel singular value decomposition (`hsvd`). This can be done with the following code:

```
identifiedModel = idModel;

% hankel svd
figure
hsvd(idss(identifiedModel));
```

This command brings up a plot like the one shown in Figure 7 that hows the contribution of each state in the model. The model has 8 states, but only 3 of them contribute the most to the model’s output.

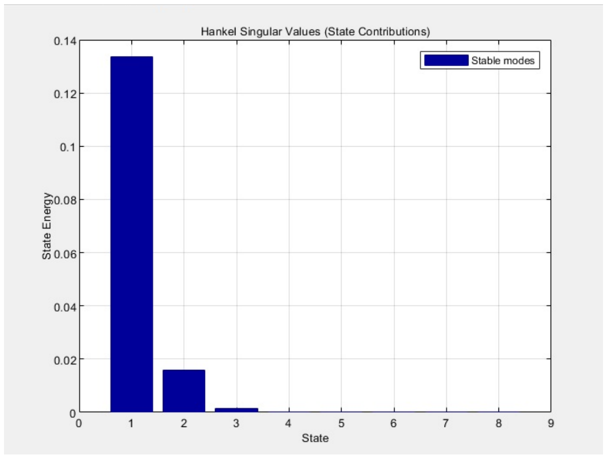


Figure 7. Hankel singular value decomposition analysis of the identified model.

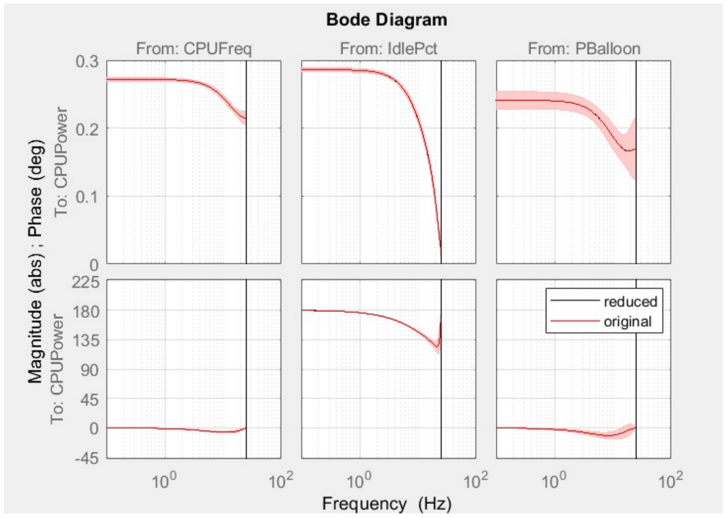


Figure 8. Comparing the reduced and original identified models.

We can conservatively trim the size of the model to 4. Recall that our identified model captures both the deterministic and non-deterministic (i.e., noise) aspects of the system. We obtain reduced versions of the full model and the deterministic part of the model. This can be done with the following code:

```
newModelOrder = 4;
%deterministic + noise
reducedIdentifiedModel = balred(idss(identifiedModel), newModelOrder);

%deterministic part of model
systemModel = balred(ss(identifiedModel), newModelOrder);
```

To verify that our reduced model is close enough to the original version, we can compare their response with the following code. This operation will produce a plot similar to Figure 8. We can see that the responses of the original and reduced models are virtually identical.

```
bode(reducedIdentifiedModel, 'black', identifiedModel, 'red');
legend('reduced', 'original')
```

We can now proceed to the controller design using our reduced model.

3 DESIGNING THE ROBUST CONTROLLER

Designing the robust controller is semi automated with Matlab. We first specify the controller's design parameters (also called hyper-parameters) and then provide these parameters to the Matlab robust control synthesis functions. We consider these two steps in the next two sections.

For convenience, we define Matlab variables to hold information about our model, including the number of inputs, outputs and states (or dimension) of the model, and the time interval between consecutive samples of the data collected for system identification (i.e., the sampling time). The inverse of the sampling time (i.e., the sampling rate) is the rate at which the controller, which we will design in the subsequent steps, will be invoked. We set all these variables from the properties of the model we identified (`systemModel`) as shown in the following code. The code also calculates the range of frequencies over which the controller and system will be analyzed in the subsequent steps.

```
numStates = length(systemModel.A);
numInputs = size(systemModel.B, 2) - numExternals;
numOutputs = size(systemModel.C, 1);

samplingTime = systemModel.Ts;

%Sampling frequency (in rad/s) = 2*pi/samplingTime
%Nyquist frequency is the maximum frequency for which we can analyze the system.
 ↪  It is 0.5 * sampling frequency = pi /samplingTime
nyquistFrequency = pi / samplingTime;
%The input rates (or frequencies) over which we run our analysis: 10^-3 to
 ↪  nyquistFrequency, with a 100 evenly spaced points in between.
omega = logspace(-3, log10(nyquistFrequency), 100);
```

3.1 Specifying the Design Parameters

For our controller, we need to specify: (i) the effect of discretized values of the inputs, (ii) the uncertainty guardband or modeling error that the controller should tolerate, (iii) the bounds on the deviations of the output from its target, and (iv) the weights for the inputs [1, 2]. After we specify these parameters, Matlab will be able to automatically synthesize the appropriate controller. These parameters are specified as transfer functions.

We model the input discretization effect as a combination of input multiplicative uncertainty and by weighing the inputs [11]. More information on other ways to model such effects can be found in [11]. The transfer functions are set through the following commands. We choose 15% as the impact of input discretization for all inputs, 40% as the uncertainty guardband, and 0.25 as the input weight for all inputs. The code also brings up the Bode plots of the transfer functions.

```
%% Input discretization
%Using 15% for all three inputs
WrIp1 = zpk(0.8624, 0.4493, 0.5, samplingTime);
WrIp2 = zpk(0.8624, 0.4493, 0.5, samplingTime);
WrIp3 = zpk(0.8624, 0.4493, 0.5, samplingTime);
WrIp = blkdiag(WrIp1, WrIp2, WrIp3);

subplot(2, 2, 1);
plt = bodeplot(WrIp(1, 1), WrIp(2, 2),WrIp(3, 3), omega);
setoptions(plt, 'PhaseVisible', 'off', 'MagUnits', 'abs', 'FreqUnits', 'Hz',
 ↪  'YLim',{[0, 1]}, 'grid', 'on');
title('Input multiplicative uncertainty for modeling discretization');

%% Output multiplicative uncertainty guardband
%Choosing 40%
WrOp = zpk(0.5093, 0.01832, 0.4, samplingTime);
WrOp = blkdiag(WrOp);

subplot(2, 2, 2);
```

```
plt = bodeplot(WrOp(1, 1), omega);
setoptions(plt, 'PhaseVisible', 'off', 'MagUnits', 'abs', 'FreqUnits', 'Hz',
 ↪  'YLim',{[0, 1.5]}, 'grid', 'on');
title('Output multiplicative uncertainty guardband');

%% Tracking performance weight or output deviation bounds
%Choosing a 10% bound
%We want deviation < Werr^-1
Werr = zpk(-0.7293, 0.1353, 4.5, samplingTime);
Werr = blkdiag(Werr);
Werrinv = Werr^-1;

subplot(2, 2, 3);
plt = bodeplot(Werrinv(1, 1), omega);
setoptions(plt, 'PhaseVisible', 'off', 'MagUnits', 'abs', 'FreqUnits', 'Hz',
 ↪  'YLim',{[0, 1.5]}, 'grid', 'on');
title('Tracking performance weight');

%% Input weight
%Choosing 0.25 for all inputs
%We want input changes < Wip^-1
Wip1 = tf([4], [1], samplingTime);
Wip2 = tf([4], [1], samplingTime);
Wip3 = tf([4], [1], samplingTime);
Wip = blkdiag(Wip1, Wip2, Wip3);
Wipinv = Wip^-1;

subplot(2, 2, 4);
plt = bodeplot(Wipinv(1, 1), Wipinv(2, 2), Wipinv(3, 3),omega);
setoptions(plt, 'PhaseVisible', 'off', 'MagUnits', 'abs', 'FreqUnits', 'Hz',
 ↪  'YLim',{[0, 1]}, 'grid', 'on');
title('Input weights');
```

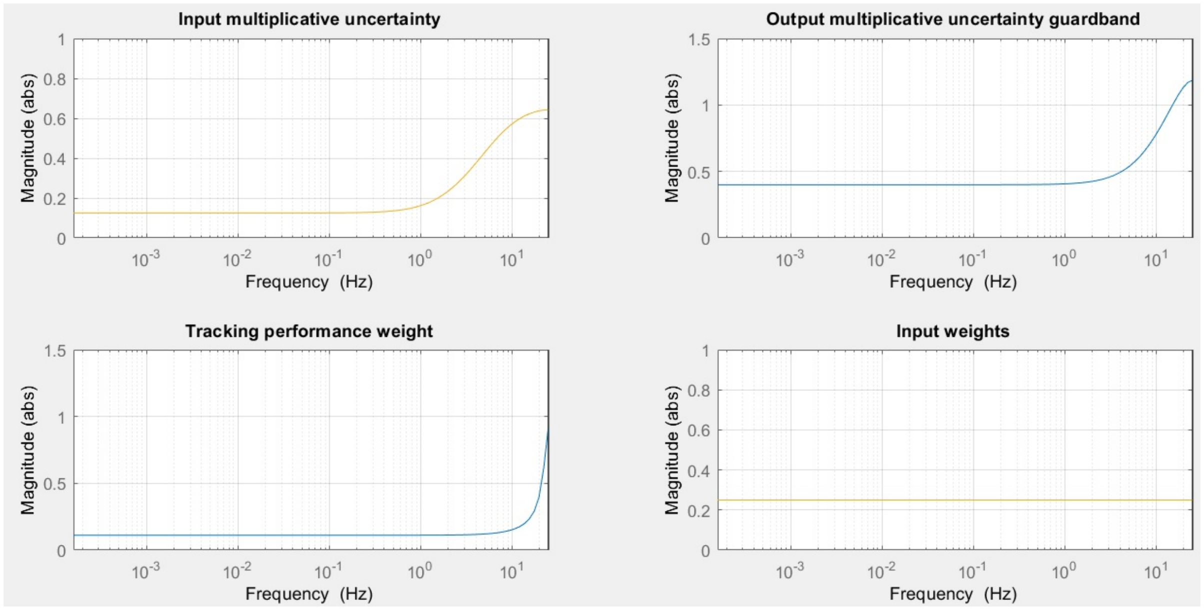


Figure 9. Control design parameters.

Next, we provide these parameters to the Matlab robust control synthesis functions using a technique known as structure specification.

3.2 Creating a System Structure and Controller Synthesis

First, we must specify the discretization effect of the inputs and the uncertainty guardbands as $W-\Delta$ pairs, where Δ is a block of unit size that represents unmodeled effects, and W is the worst case impact of those effects. For example, if Wr_{op} is our uncertainty guardband, then the $Wr_{op}-\Delta$ pair means that there are unmodeled effects (or modeling errors) in the system that affect the model's output, and the worst case impact of those effects is Wr_{op}.

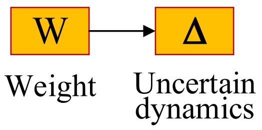


Figure 10. Specifying unmodeled effects as $W-\Delta$ pairs, where Δ represents unmodeled (or uncertain) phenomena, and W is a weight describing the worst case impact of the unmodeled effects.

Note that there is no constraint on what the unmodeled effects (Δ) are and how they impact the model's accuracy—e.g., the unmodeled effects could be linear or nonlinear, and time-varying or time-invariant. We are only specifying this *worst* possible impact of such effects as Wr_{op}. The robust controller we obtain will be stable and keep outputs near the targets even with such worst-case uncertainty. This is a powerful guarantee that enables our controller to work for several types of applications (and possibly other computers) beyond what we used for system identification. This is also the reason why robust controllers are particularly suited for computers because we do not have accurate dynamic models of computers.

We can use the following code to create the Δ blocks for the input discretization and uncertainty guardband.

```
%% Delta block for input multiplicative uncertainty
deltaIp1 = ultidyn('Delta_ip1', [1 1], 'Bound', 1.0);
deltaIp2 = ultidyn('Delta_ip2', [1 1], 'Bound', 1.0);
deltaIp3 = ultidyn('Delta_ip2', [1 1], 'Bound', 1.0);
deltaIp = blkdiag(deltaIp1, deltaIp2, deltaIp3);

%%Delta block for uncertainty guardband
deltaOp = ultidyn('Delta_op', [1 1], 'Bound', 1.0);
```

To understand what happens if we include the Δ blocks with our model, we can connect the $W-\Delta$ pairs with it and create what is known as an Uncertain Model, as shown in Figure 11. The input to this uncertain model is the controller's actuation for the three inputs (`ctl{3}`) and the output is the power value. The following code performs this connection.

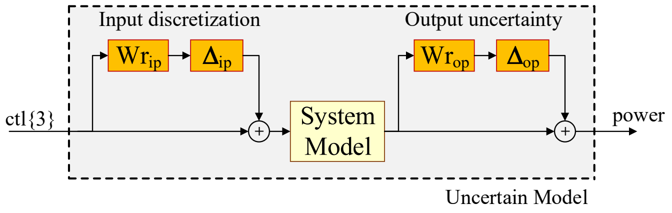


Figure 11. The uncertain model created by connecting our system model with the $W-\Delta$ pairs.

```
% Define uncertain model: uncertip (ip +delta)->sys->uncertop (op+delta)
ny = num2str(numOutputs);
nu = num2str(numInputs);
systemnames = 'systemModel WrIp WrOp deltaIp deltaOp';
inputvar = ['[ ctl{' nu '} ]'];
outputvar = '[ systemModel + deltaOp ]';
input_to_WrIp = '[ ctl ]';
```

```
input_to_deltaIp = '[ WrIp]';
input_to_systemModel = '[ ctl + deltaIp ]';
input_to_WrOp = '[ systemModel ]';
input_to_deltaOp = '[ WrOp ]';
sysoutname = 'uncertModel';
cleanupsysic = 'yes';
sysic;
```

The uncertain model is a more realistic description of the system because it says that the impact of the inputs could be different than what is given by the model. Indeed, we can see the step response of the uncertain model using the following code. Running this code will show a response similar to Figure 12. Figure 12 shows the response of the identified model as the “Nominal” response and the various possible responses covered by the uncertain model as “Perturbed”.

```
uncertSamples = usample(uncertModel, 10);
step(uncertModel.NominalValue, 'r-o', uncertSamples, 'c', 0.1);
legend('Nominal', 'Perturbed');
title('Step response of the uncertain model');
```

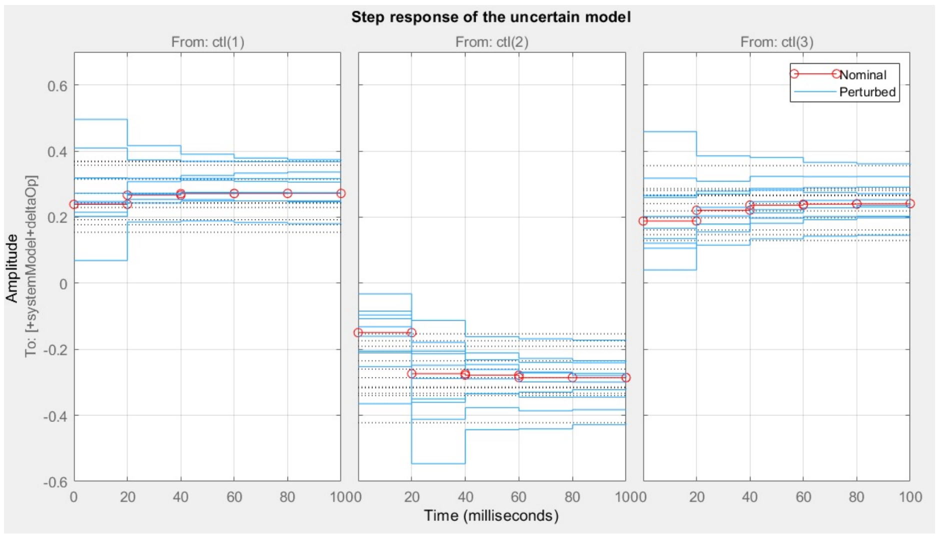


Figure 12. The step response of the uncertain model.

It can be seen that the uncertain model captures a larger class of responses, which could be higher or lower than that of the nominal model. This makes the uncertain model more realistic, and increases its coverage of actual applications. Thus, it is a better mathematical proxy for the actual system. The goal of robust control synthesis is to generate a single controller that can effectively manage systems with any of these responses.

We will now specify the full structure of the system with the $W-\Delta$ pairs, the output bounds, and input weights. This structure is shown in Figure 13. This structure specifies that the controller will read the deviations of power from the system, and that these deviations should be kept below the W_{err}^{-1} that we chose earlier for any given target value (`ref`). Furthermore, the input changes that the controller produces must be below W_{ip}^{-1} because the input values are quantized and finite.

The following code makes the connections to create the structure in Figure 13. In this code, the full description is named `uncertStruct` (line 7 in the code listing).

```
1 systemnames = 'uncertModel Werr Wip';
2 inputvar = ['[ ref{' ny '}; ctl{' nu '}]'];
3 outputvar = '[ Werr ;Wip; ref - uncertModel ]';
4 input_to_uncertModel = '[ ctl ]';
5 input_to_Werr = '[ ref - uncertModel ]';
6 input_to_Wip = '[ ctl ]';
```

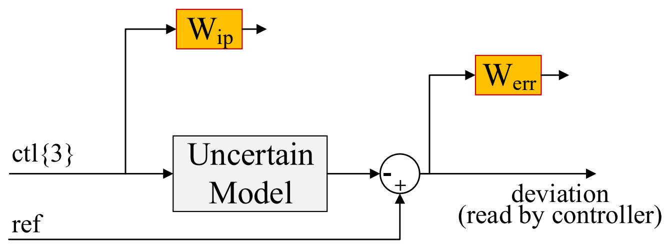


Figure 13. Structure with all the specifications.

```
sysoutname = 'uncertStruct';
cleanupsysic = 'yes';
sysic;
```

Finally, through the following code, we invoke the Matlab functions to design a robust controller (also called a Mu or SSV controller) that satisfies the requirements and constraints given in the structure `uncertStruct` created above.

```
numAutoIterations = 3;
dkopts = dksynOptions('FrequencyVector', omega, 'NumberofAutoIterations',
 ↪  numAutoIterations, 'DisplayWhileAutoIter', 'on');
[Kmu4, ~, gamma_mu, dkinfo] = dksyn(uncertStruct, numOutputs, numInputs, dkopts);
```

The code generates a controller stored in variable `Kmu4`. It also shows a plot such as the one shown in Figure 14 with the SSV (Structured Singular Value) or Mu value [2] of the system with the generated controller. If the SSV value is less than 1, it means that we obtained a controller that meets all the requirements [2]. If this value is more than 1 or if the synthesis fails, we have to relax our design parameters, such as the uncertainty guardbands, output deviation bounds, or input weights. We can see from Figure 14 that the controller we designed meets all the constraints we specified.

We use the following code to specifically check the closed loop system's robust stability (i.e., if the system including the controller is stable for all modeling errors within the guardband), and its robust performance (i.e., if the deviations of power are kept within the bounds for all given targets even with modeling errors). This code generates a figure like the one shown in Figure 15. In this figure, we can see that the poles and zeros of the system (which includes computer plus controller) are all within the unit circle, and that the robust stability criterion is less than 1. We can also see that the nominal performance (i.e., keeping output deviations within the given bounds when there is no modeling error) and the robust performance are also less than 1. At the command line, Matlab outputs text describing the largest disturbance (or uncertainty) for which the controller can keep the system stable and maintain its performance.

```
Ktest = Kmu4;

% define closed loop
uncertLoop = lft(uncertStruct, Ktest);

figure
%% nominal stability
subplot(2, 2, 1)
loopTf = uncertModel.NominalValue*Ktest;
pzmap(loopTf);

%% robust stability
robstabopt = robOptions('Display', 'on', 'VaryFrequency', 'on');
[stabmargin, ~, info] = robstab(uncertLoop, omega, robstabopt);
subplot(2, 2, 2);
semilogx(info.Frequency, info.Bounds.^-1);
title('Robust stability')
ylabel('Mu bounds')
```

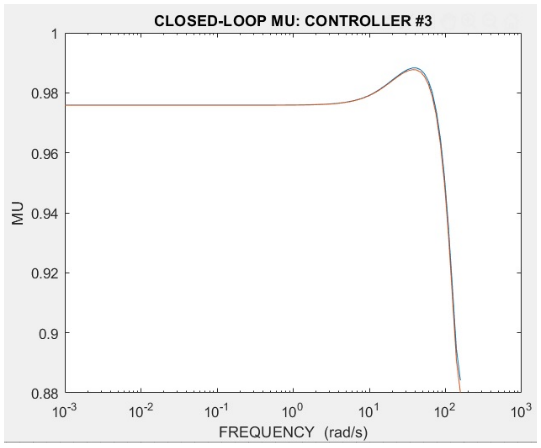


Figure 14. SSV (Mu) of the system with the generated controller.

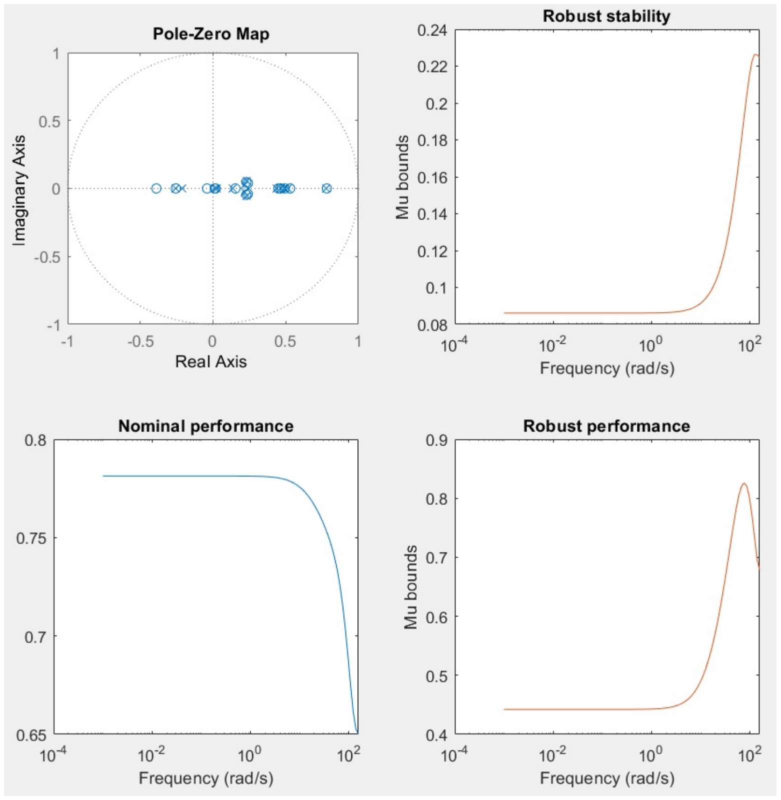


Figure 15. Analyzing the robust stability and robust performance of the closed loop system (which includes computer plus controller).

```
xlabel('Frequency (rad/s)')

%% nominal performance
[singularValues, ~] = sigma(uncertLoop.NominalValue, omega);
subplot(2, 2, 3)
semilogx(omega, singularValues(1, :))
title('Nominal performance')
xlabel('Frequency (rad/s)')

%% robust performance
getPeakGain(uncertLoop)
robperfopt = robOptions('Display', 'on', 'VaryFrequency', 'on');
[perfmargin, ~, perfinfo] = robgain(uncertLoop, 1, omega, robperfopt);
numPoints = size(perfinfo.Bounds, 1);
subplot(2, 2, 4)
if (numPoints == 1)
    % Nominal performance is not met. Use dksynperf
    display('Nominal performance not met');
    [~, ~, perfinfo] = dksynperf(uncertLoop, omega);
    semilogx(perfinfo.Frequency, perfinfo.Bounds)
else
    semilogx(perfinfo.Frequency, perfinfo.Bounds.^-1)
end

title('Robust performance')
```

```
ylabel('Mu bounds')
xlabel('Frequency (rad/s)')
```

It can be seen from Figure 15 that the peak robust stability bound is 0.226, while the peak robust performance is 0.826. This means that the controller has a larger margin for robust stability than for robust performance. Indeed, the controller can remain stable for up to $\frac{1}{0.226} = 4.42$ times the modeled uncertainty (as given by Matlab at the command line), and can maintain its tracking performance for up to $\frac{1}{0.826} = 1.21$ times the modeled uncertainty. So, after deployment, even if the difference between the real behavior of the system and the behavior given by the model is larger than the guardband, the controller will still be stable—although it might not keep outputs within 10% of the targets. Hence, this is a conservative guardband.

Now that we have successfully obtained a robust controller, we will validate and deploy it.

4 VALIDATING AND DEPLOYING THE CONTROLLER

We can test the step response and frequency response (Bode analysis) of the controller using the commands below. This brings up plots like the ones shown in Figures 16 and 17.

```
step(Kmu4, 10 * samplingTime);
 figure
plt =bodeplot(Kmu4, omega);
setoptions(plt, 'MagUnits', 'abs', 'FreqUnits', 'Hz', , 'grid', 'on');
```

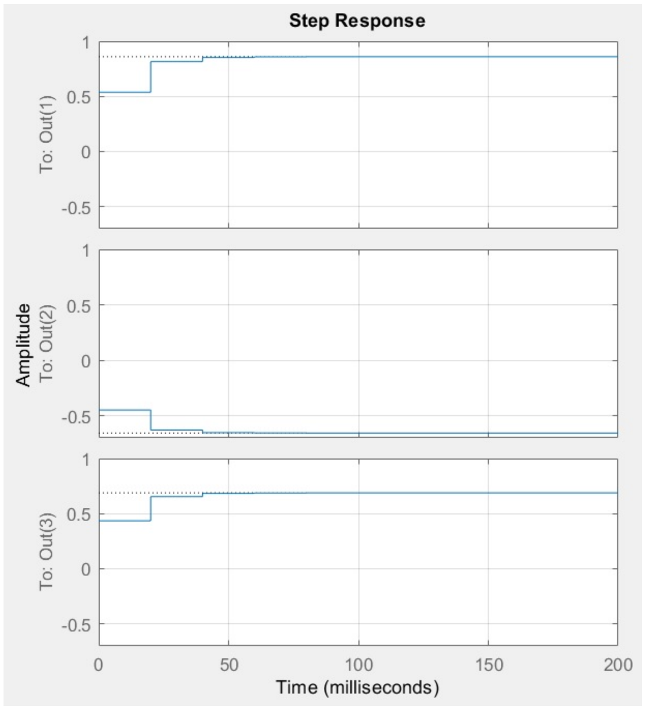


Figure 16. Step response of the generated controller.

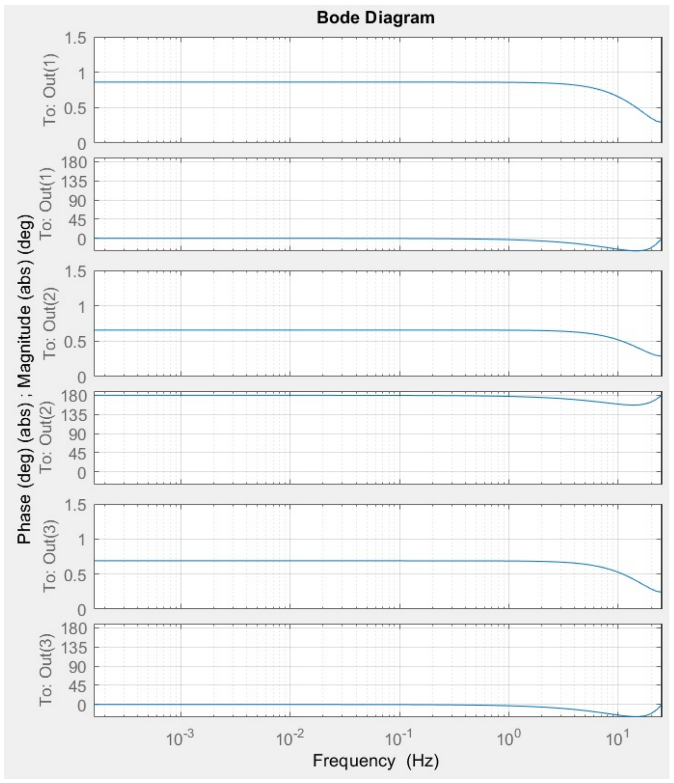


Figure 17. Bode analysis of the generated controller.

Figure 16 shows how the controller responds to a step change in the power deviation, i.e., when the power deviation goes from 0 to 1. Since the power deviation is measured as $target - measured\ power$, this means that the measured power has fallen much below the target value. From Figure 16, we can see that, in this case, the controller will increase the frequency (first input to the system or the first output of the controller, `Out(1)`), decrease the idle activity (`Out(2)`), and increase the power balloon activity (`Out(3)`). As a result, power consumption will increase. We can similarly analyze the controller from its Bode plot shown in Figure 17.

Sometimes, if the controller produces large changes in the system inputs, it might cause large overshoots and undershoots in the power. In that case, we can re-synthesize the controller with a higher input weight (W_{ip}) to keep inputs small and improve the smoothness of transitions. Furthermore, Matlab routines can sometimes synthesize controllers whose dimensions are too big. If that happens, the controller's order can be reduced similar to how the model's dimensions were reduced in Section 2.3. More information on reducing the order of robust controllers can be found in [12].

After we are satisfied with the controller, we can write the controller's information into files that will be read by the Maya software. This can be done with the following code. In this code, one needs to change the directory where this information is stored by setting the `ctlDir` variable appropriately. The prefix to each file can be updated by changing the `tag` variable. Note that one needs to specify the minimum and maximum values of the inputs and output in the variables `minIpValues, maxIpValues, minOpValues, maxOpValues`.

```
ctlDir = '..\Controller\';
ctl = Kmu4;
tag = 'mayaRobust';

ipScales = (maxIpValues - minIpValues)/2
opScales = (maxOpValues - minOpValues)/2
ipScaleUp = diag(ipScales);
opScaleDown = inv(opScaleUp);

opTargets = maxOpValues*0.5;
opMaxLimits = maxOpValues*0.9;
opMinLimits = minOpVales*1.2;

ctlName = strcat(ctlDir, tag);

ctlDim = length(ctl.A);

dlmwrite(strcat(ctlName, '_dimension.txt'), ctlDim, ' ')
dlmwrite(strcat(ctlName, '_numInputs.txt'), numInputs, ' ')
dlmwrite(strcat(ctlName, '_numYmeas.txt'), numOutputs + numExternals, ' ')

dlmwrite(strcat(ctlName, '_A.txt'), ctl.a, ' ')
dlmwrite(strcat(ctlName, '_B.txt'), ctl.b, ' ')
dlmwrite(strcat(ctlName, '_C.txt'), ctl.c, ' ')
dlmwrite(strcat(ctlName, '_D.txt'), ctl.d, ' ')

dlmwrite(strcat(ctlName, '_scaleYmeasDown.txt'), [ipScales(1 : numExternals).^-1
 ↪  opScales.^-1], ' ')
dlmwrite(strcat(ctlName, '_scaleInputsUp.txt'), ipScales(numExternals + 1 :
 ↪  numExternals + numInputs), ' ')

dlmwrite(strcat(ctlName, '_targets.txt'), opTargets, ' ')

dlmwrite(strcat(ctlName, '_maxLimits.txt'), opMaxLimits, ' ')
dlmwrite(strcat(ctlName, '_minLimits.txt'), opMinLimits, ' ')
```

After these files are written, we can transfer them to the system where we want to run Maya, and run the Maya software to mask the system's power using the designed controller. We can test Maya's tracking performance on a couple of validation applications. This can be done with the `Launch.sh` script as follows. We can view and visualize the logs to check if the controller's tracking ability is satisfactory. We can change the controller's design parameters to fix any issue we observe. For example, if there are sharp changes and the controller is over-reacting to small deviations, then we can increase the input weights. On the other hand, if the controller is too sluggish when tracking the outputs, then we can increase the output bounds and/or decrease the input weights.

```
# Run Maya with the Gaussian Sinusoid mask generator for validation applications.
↪ The robust controller files are in the ../../Controller directory and the
↪ files are prefixed with the name mayaRobust
sudo -E --preserve-env=PATH env "LD_LIBRARY_PATH=$LD_LIBRARY_PATH" bash
↪ ./Launch.sh --rundir "../Dist/Release/" --options "--mode Mask --mask
↪ GaussSine --ctldir ../../Controller --ctlfile mayaRobust" --logdir "<logdir>"
↪ --tag "<name>" --apps "<appname>" &
```

Finally, when we are satisfied with the controller, we can use it with the Maya software using the `Launch.sh` script or directly as follows.

```
# Run Maya with the Gaussian Sinusoid mask generator. The robust controller files
↪ are in the ../../Controller directory and the files are prefixed with the name
↪ mayaRobust
sudo LD_LIBRARY_PATH=<path to lib64>/:\$LD_LIBRARY_PATH ./Maya --mode Mask --mask
↪ GaussSine --ctldir ../../Controller --ctlfile mayaRobust > /dev/null 2>&1 &
```

5 CONCLUSIONS

In this technical report, we described the Matlab-assisted design procedure to synthesize the formal controller in Maya, a new security defense that uses formal control to obfuscate the power side channels of a computer. The controller reads the deviation of power from a target given to it, and changes the system knobs to minimize these deviations. This controller is designed with robust control theory principles, and enables Maya to thoroughly reshape a computer's power for security.

REFERENCES

[1] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, A. Schwing, and J. Torrellas, "Maya: Using Formal Control to Obfuscate Power Side Channels," in *International Symposium on Computer Architecture*, Jun. 2021.

[2] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Yukta: Multilayer Resource Controllers to Maximize Efficiency," in *International Symposium on Computer Architecture*, Jun. 2018.

[3] R. P. Pothukuchi, J. L. Greathouse, K. Rao, C. Erb, L. Piga, P. Voulgaris, and J. Torrellas, "Tangram: Integrated Control of Heterogeneous Computers," in *International Symposium on Microarchitecture*, Oct. 2019.

[4] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Structured Singular Value Control for Modular Resource Management in Multilayer Computers," in *IEEE Conference on Decision and Control*, Dec. 2018.

[5] R. P. Pothukuchi, S. Y. Pothukuchi, P. G. Voulgaris, and J. Torrellas, "Control Systems for Computing Systems: Making computers efficient with modular, coordinated, and robust control," *IEEE Control Systems Magazine*, vol. 40, no. 2, pp. 30–55, April 2020.

[6] R. P. Pothukuchi and J. Torrellas, *A Guide to Design MIMO Controllers for Processors*, http://iacoma.cs.uiuc.edu/iacoma-papers/mimoTR.pdf, Apr. 2016.

[7] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures," in *International Symposium on Computer Architecture*, Jun. 2016.

[8] R. P. Pothukuchi and S. Y. Pothukuchi, "Maya: Obfuscating Power Side Channels with Formal Control," 2021. [Online]. Available: https://github.com/mayadefense/maya

[9] L. Ljung, *System Identification: Theory for the User*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

[10] Y. Bao, C. Bienia, and K. Li, "PARSEC 3.0: The Parsec Benchmark Suite," 2011. [Online]. Available: https://parsec.cs.princeton.edu/parsec3-doc.htm

[11] J. C. Doyle, J. E. Wall, and G. Stein, "Performance and Robustness Analysis for Structured Uncertainty," in *IEEE Conference on Decision and Control*, Dec. 1982.

[12] "Improve Results of Mu Synthesis," The MathWorks Inc., Natick, Massachusetts, 2021. [Online]. Available: https://www.mathworks.com/help/robust/ug/improve-results-of-mu-synthesis.htm