The Mask of XORRO

Binchilling
CS771 - Introduction to Machine Learning
Indian Institute of Technology, Kanpur
2022-23 Semester II

1 Notation

We define the notations which will be consistently used throughout this report.

$R \rightarrow$ The number of XOR gates in a single XORRO

$S \rightarrow$ The number of select bits used in the MUX

$c \rightarrow$ The input challenge string whose length is $R + 2S$

$c_i \rightarrow$ The first R bits of the challenge, that is the non-oscillating input of each XOR gate

$p_i \rightarrow$ The next S bits of the challenge, that is the select bits to obtain the first XORRO

$r_i \rightarrow$ The last S bits of the challenge, that is the select bits to obtain the second XORRO

$y_c \rightarrow$ The output of the XORRO when given a challenge c

So, the input challenge looks like:

$$c = c_0c_1c_2 \cdots c_{R-1}p_0p_1p_2 \cdots p_{S-1}r_0r_1r_2 \cdots r_{S-1}$$

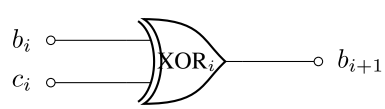
The inputs and output of the i^{th} XOR is as follows:

$b_i \rightarrow$ The first input to the XOR. It is the **oscillating** input

$c_i \rightarrow$ The second input to the XOR. It is the **non-oscillating** input from the challenge

$b_{i+1} \rightarrow$ It is the output of the XOR

Note that the indices are taken **modulo** R. An illustration of the i^{th} XOR is given below.



Finally, for a XORRO A:

$\delta_{xy}^{A_i} \rightarrow$ Denotes the signal propagation delay in the i^{th} XOR when the input to the first terminal is x and the input to the second terminal is y

$t_0^A \rightarrow$ The time taken for the output of the XORRO to toggle from a 0 to a 1

$t_1^A \rightarrow$ The time taken for the output of the XORRO to toggle from a 1 to a 0

$f^A \rightarrow$ The frequency of the XORRO. It is equal to $\frac{1}{t_0^A + t_1^A}$

Preprint. Under review.

2 Cracking a XORRO PUF

The key is to realize that the value of $t_0^A + t_1^A$ is independent of the second input to each XOR at an instant of time for any XORRO A. Label the two XORROs as A and B. We have

$$
\begin{aligned}
t_0^A + t_1^A &= \sum_{i=0}^{R-1} \delta_{0c_i}^{A_i} + \delta_{1c_i}^{A_i} \\
t_0^B + t_1^B &= \sum_{i=0}^{R-1} \delta_{0c_i}^{B_i} + \delta_{1c_i}^{B_i}
\end{aligned}
$$

Considering the time difference,

$$
\begin{aligned}
(t_0^A + t_1^A) - (t_0^B + t_1^B) &= \sum_{i=0}^{R-1} \delta_{0c_i}^{A_i} + \delta_{1c_i}^{A_i} - \sum_{i=0}^{R-1} \delta_{0c_i}^{B_i} + \delta_{1c_i}^{B_i} \\
&= \sum_{i=0}^{R-1} \xi_{c_i}^{A_i} - \xi_{c_i}^{B_i} \\
&= \sum_{i=0}^{R-1} \Delta_{c_i}^{i}
\end{aligned}
$$

where $\xi_{c_i}^{A_i} = \delta_{0c_i}^{A_i} + \delta_{1c_i}^{A_i}$ and $\Delta_{c_i}^{i} = \xi_{c_i}^{A_i} - \xi_{c_i}^{B_i}$

$$
\begin{aligned}
(t_0^A + t_1^A) - (t_0^B + t_1^B) &= \sum_{i=0}^{R-1} \Delta_{c_i}^{i} \\
&= \sum_{i=0}^{R-1} (1 - c_i)\Delta_0^i + c_i\Delta_1^i \\
&= \sum_{i=0}^{R-1} \Delta_0^i + c_i(\Delta_1^i - \Delta_0^i) \\
&= b + \sum_{i=0}^{R-1} c_i \cdot w_i \\
&= w'^T c + b
\end{aligned}
$$

With w' and c being $\mathbb{R}^R$ dimensional vectors.

$$
\begin{aligned}
w'^T &= [w_0\, w_1\, \cdots\, w_{R-1}] \\
c &= [c_0\, c_1\, \cdots\, c_{R-1}]^T
\end{aligned}
$$

Now,

$$
f^A - f^B > 0 \Leftrightarrow (t_0^A + t_1^A) - (t_0^B + t_1^B) < 0 \Leftrightarrow w'^T c + b < 0
$$

Absorb a negative sign into w' and let $w = -w'$. Then, $w^T c + b > 0 \Rightarrow$ Output $= 1$ and $w^T c + b < 0 \Rightarrow$ Output $= 0$

Hence, we have successfully obtained a linear model(w, b) where

$$
y_c = \frac{1 + \text{sign}(w^T c + b)}{2}
$$

The feature vector is simply the challenge vector bits - no feature engineering was necessary.

3 Cracking an Advanced XORRO PUF

We notice that the time period of a single XORRO A is a linear combination of its parameters, that is, a linear model $M^A = (w^A, b^A)$ can precisely capture its time period. The derivation is analogous to the previous one. We borrow some notation from the previous derivation:

$$\begin{aligned} t_0^A + t_1^A &= \sum_{i=0}^{R-1} \delta_{0c_i}^{A_i} + \delta_{1c_i}^{A_i} \\ &= \sum_{i=0}^{R-1} \xi_{c_i}^{A_i} \\ &= \sum_{i=0}^{R-1} (1 - c_i)\xi_0^{A_i} + c_i \xi_1^{A_i} \\ &= \sum_{i=0}^{R-1} \xi_0^{A_i} + c_i(\xi_1^{A_i} - \xi_0^{A_i}) \\ &= b^A + \sum_{i=0}^{R-1} c_i \cdot w_i^A \\ &= (w^A)^T c + b^A \end{aligned}$$

This linear model has a $\mathbb{R}^R$ dimensional weight vector and a bias. In fact, the linear model to crack the XORRO PUF in the first part was simply $M^B - M^A = (w^B - w^A,\ b^B - b^A)$.

For the advanced XORRO PUF, we index the 2^S XORROs using binary bits. The XORROs are labelled as $X_{x_0x_1x_2\cdots x_{S-1}}$ with $x_i \in \{0, 1\}$. Hence in the challenge c, $X_{p_0p_1\cdots p_{S-1}}$ is chosen as the first XORRO and $X_{r_0r_1\cdots r_{S-1}}$ is chosen as the second XORRO. Also, let $M^{X_{x_0x_1\cdots x_{S-1}}}$ denote the linear model which captures the time period of each of the XORROs. For the sake of brevity, let x denote the index of each XORRO, $x = x_0x_1 \cdots x_{S-1}$. We aim to capture all these linear models **using a single linear model**. Define two functions f, g which takes inputs from the index of a XORRO and outputs a function of the select bits in c:

$$f(x_i) = \begin{cases} 1 - p_i & \text{if } x_i = 0 \\ p_i & \text{if } x_i = 1 \end{cases}$$

$$g(x_i) = \begin{cases} 1 - r_i & \text{if } x_i = 0 \\ r_i & \text{if } x_i = 1 \end{cases}$$

We can now condense all the models into a single model $\mathbb{M}$ as follows:

$$\mathbb{M} = \sum_{x \in \{0,1\}^S} \left(\left(\prod_{i=0}^{S-1} f(x_i) \right) - \left(\prod_{i=0}^{S-1} g(x_i) \right) \right) M^{X_x}$$

Notice that for a challenge c, $\mathbb{M}$ reduces to $M^{X_{p_0p_1\cdots p_{S-1}}} - M^{X_{r_0r_1\cdots r_{S-1}}}$. All other models go to 0 by the construction of f and g. Hence,

$$\mathbb{M} > 0 \Leftrightarrow M^{X_{p_0p_1\cdots p_{S-1}}} > M^{X_{r_0r_1\cdots r_{S-1}}} \Leftrightarrow \text{Output: } 0$$

Finally, we do a formal feature analysis. Let $\mathbb{J}^* = 2^{\{p_0, p_1, \cdots, p_{S-1}\}}$ denote the power set of the first set of select bits **except the empty set**. Let $\mathbb{T}^* = 2^{\{q_0, q_1, \cdots, q_{S-1}\}}$ denote the power set of the second set of select bits **except the empty set**. Let $\mathbb{F}$ map an element from $(\mathbb{J}^* \cup \mathbb{T}^*)$ to a feature, which is simply the product of the elements of the set. More formally,

$$\mathbb{F}(B) = \prod_{b \in B} b, \quad \text{where } B \in (\mathbb{J}^* \cup \mathbb{T}^*)$$

Let $\mathbb{U} = \{1\} \cup \{c_0, c_1, \cdots, c_{R-1}\}$. Then the features of $\mathbb{M}$ are $((\mathbb{J}^* \times \mathbb{U}) \cup (\mathbb{T}^* \times \mathbb{U}))$ ($\times$ denotes the cartesian product) and a bias term. This gives a total of $(R+1) \cdot (2^S - 1) \cdot 2$ features and a bias. However, we can halve the number of features with a simple observation. For any feature $p_{l_0} p_{l_1} \cdots p_{l_k} u$ (where $u \in \mathbb{U}$) the corresponding feature $r_{l_0} r_{l_1} \cdots r_{l_k} u$ will have just the negative weight of the former. So we can combine the two into a single feature namely $(p_{l_0} p_{l_1} \cdots p_{l_k} - r_{l_0} r_{l_1} \cdots r_{l_k})u$. Hence this gives us a total of $(R+1) \cdot (2^S - 1)$ features. In our example, we have 975 features and a bias.

4 The Code

```python
import numpy as np
from sklearn.svm import LinearSVC as SVC
from sklearn.linear_model import LogisticRegression as LR
from sklearn.metrics import accuracy_score

# You are allowed to import any submodules of sklearn as well e.g. sklearn.svm etc
# You are not allowed to use other libraries such as scipy, keras, tensorflow etc

# SUBMIT YOUR CODE AS A SINGLE PYTHON (.PY) FILE INSIDE A ZIP ARCHIVE
# THE NAME OF THE PYTHON FILE MUST BE submit.py
# DO NOT INCLUDE OTHER PACKAGES LIKE SCIPY, KERAS ETC IN YOUR CODE
# THE USE OF ANY MACHINE LEARNING LIBRARIES OTHER THAN SKLEARN WILL RESULT IN A STRAIGHT ZERO

# DO NOT CHANGE THE NAME OF THE METHODS my_fit, my_predict etc BELOW
# THESE WILL BE INVOKED BY THE EVALUATION SCRIPT. CHANGING THESE NAMES WILL CAUSE EVALUATION FAILURE

# You may define any new functions, variables, classes here
# For example, functions to calculate next coordinate or step length

################################
# Non Editable Region Starting #
################################
def my_fit( Z_train ):
################################
#  Non Editable Region Ending  #
################################

	# Use this method to train your model using training CRPs
	# The first 64 columns contain the config bits
	# The next 4 columns contain the select bits for the first mux
	# The next 4 columns contain the select bits for the second mux
	# The first 64 + 4 + 4 = 72 columns constitute the challenge
	# The last column contains the response
	def get_combo(Z):
		return np.concatenate([Z[:,0].reshape(-1,1)*Z[:,1:4], Z[:,1].reshape(-1,1)*Z[:,2:4], Z[:,2].reshape(-1,1)*Z[:,3:4], Z[:,:], (Z[:,0]*Z[:,1]*Z[:,2]).reshape(-1,1), (Z[:,0]*Z[:,1]*Z[:,3]).reshape(-1,1), (Z[:,0]*Z[:,2]*Z[:,3]).reshape(-1,1), (Z[:,1]*Z[:,2]*Z[:,3]).reshape(-1,1),(Z[:,0]*Z[:,1]*Z[:,2]*Z[:,3]).reshape(-1,1)], axis=1)

	def new_features(Z):
		Z1 = np.einsum('ij,ik->ikj', get_combo(Z[:,64:68]) - get_combo(Z[:,68:72]), Z[:,0:64])
		Z1 = Z1.reshape(Z1.shape[0],-1)
		Z2 = get_combo(Z[:,64:68]) - get_combo(Z[:,68:72])
		return np.concatenate([Z1, Z2], axis=1)

	model = SVC(penalty='l2', C=100, max_iter=int(1e3), tol=1e-4, loss = 'squared_hinge', dual = False)
	features = new_features(Z_train[:,:72])
	model.fit(features , Z_train[:,-1])

	return model    # Return the trained model


################################
# Non Editable Region Starting #
```

```
################################
def my_predict( X_tst, model ):
################################
#  Non Editable Region Ending  #
################################

        # Use this method to make predictions on test challenges
        def get_combo(Z):
                return np.concatenate([Z[:,0].reshape(-1,1) * Z[:,1:4],
                    Z[:,1].reshape(-1,1)*Z[:,2:4], Z[:,2].reshape(-1,1)*Z[:,3:4],
                    Z[:,:], (Z[:,0]*Z[:,1]*Z[:,2]).reshape(-1,1),
                    (Z[:,0]*Z[:,1]*Z[:,3]).reshape(-1,1),
                    (Z[:,0]*Z[:,2]*Z[:,3]).reshape(-1,1),
                    (Z[:,1]*Z[:,2]*Z[:,3]).reshape(-1,1),
                    (Z[:,0]*Z[:,1]*Z[:,2]*Z[:,3]).reshape(-1,1)], axis=1)

        def new_features(Z):
                Z1 = np.einsum('ij,ik->ikj', get_combo(Z[:,64:68]) -
                    get_combo(Z[:,68:72]), Z[:,0:64])
                Z1 = Z1.reshape(Z1.shape[0],-1)
                Z2 = get_combo(Z[:,64:68]) - get_combo(Z[:,68:72])
                return np.concatenate([Z1, Z2], axis=1)

        pred = model.predict(new_features(X_tst[:,:72]))
        return pred

```

5 Model Experimentation

5.1 Varying the C Hyperparameter

The C value was varied logarithmically from 10^{-3} to 10^{4} for both the `LinearSVC` and `LogisticRegression` models. The penalty was `l2`, `tolerance` was 10^{-4}, the loss (for the `LinearSVC`) was `squared hinge loss` and the number of iterations were 1000. All these parameters were kept constant while C was varied.

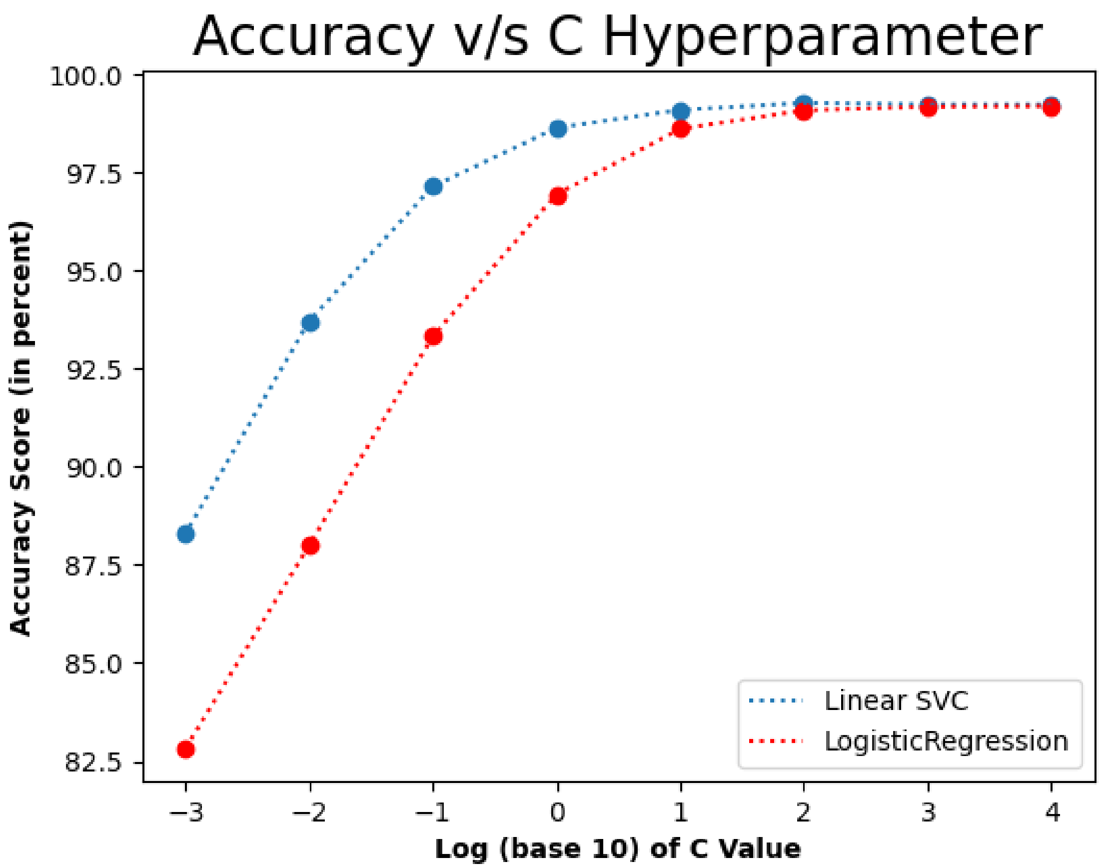


Figure 1: Plot of Accuracy

As the value of C increases, the accuracy increases for both the models. In fact, it converges to a value of around 99.3. On further increasing C beyond 100, the accuracy starts to slightly decrease for both the models. The accuracy of `LinearSVC` is always more than `LogisticRegression`. It should be noted that the `LogisticRegression` model did not converge for higher values of C.

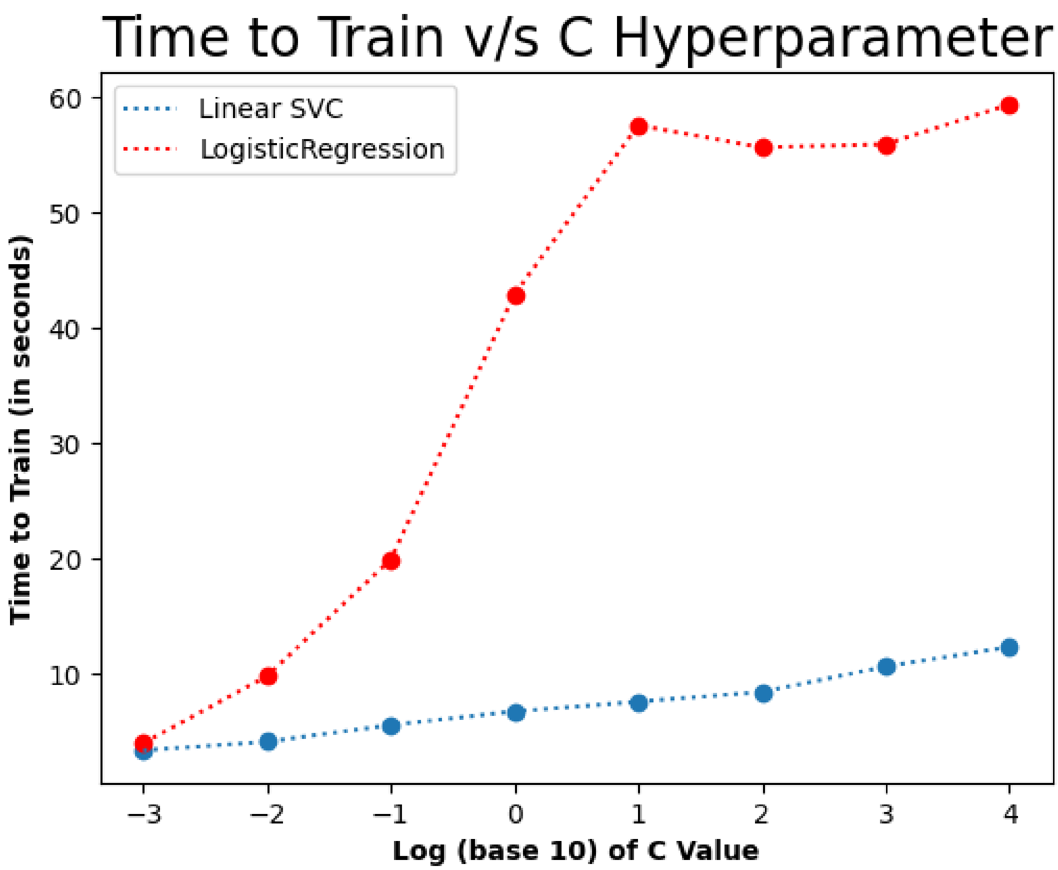


Figure 2: Plot of Time Taken to Train

For `LinearSVC`, the time taken to train monotonically increases with increasing C. For `LogisticRegression` however, the time taken has an abnormal graph with a peak at around $C = 10$. However, `LinearSVC` takes a lot less time compared to `LogisticRegression` making it the better choice for cracking advanced XORRO PUFs.

5.2 Varying the Tolerance Hyperparameter

The *tol* value was varied logarithmically from 10^{-6} to 10^{-1} for both the `LinearSVC` and `LogisticRegression` models. The penalty was `l2`, `C` was 100, the loss (for the `LinearSVC`) was `squared hinge loss` and the number of iterations were 1000. All these parameters were kept constant while *tol* was varied.

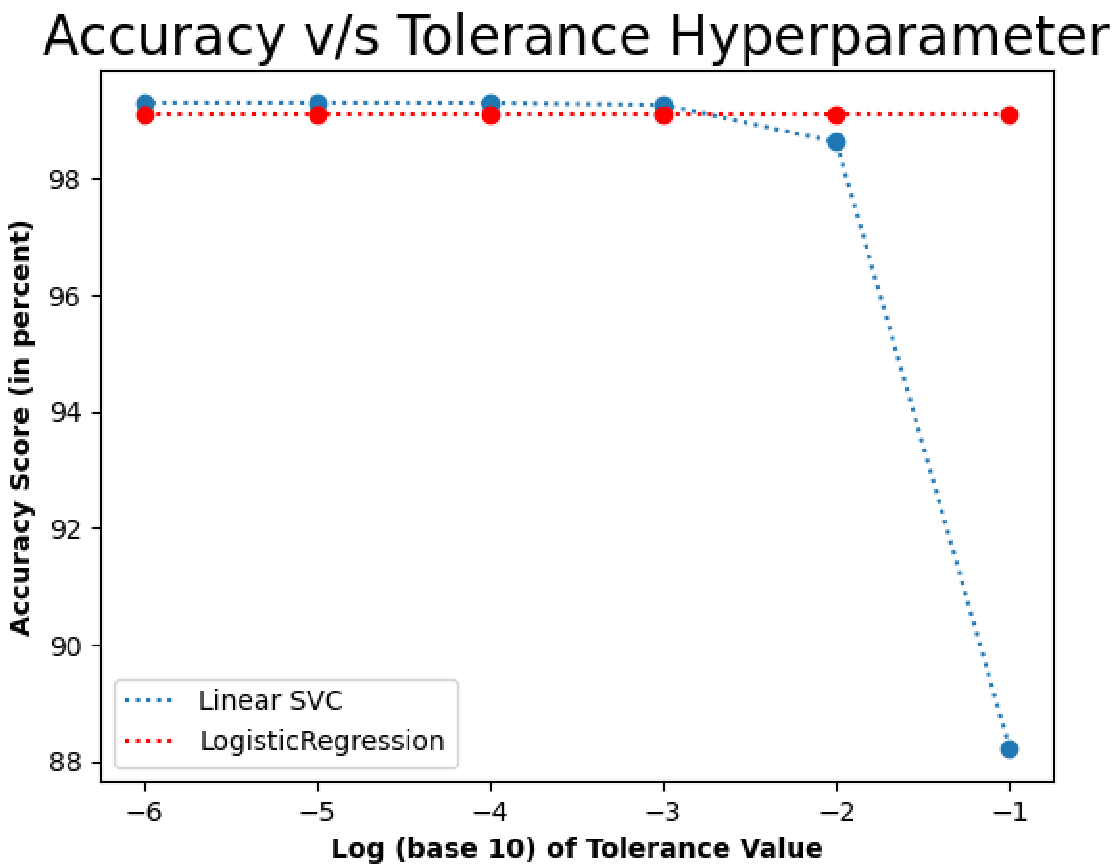


Figure 3: Plot of Accuracy

As tolerance increases, the accuracy decreases slowly for `LinearSVC` with a sudden drop at 10^{-1}. The accuracy remains unchanged for `LogisticRegression`.

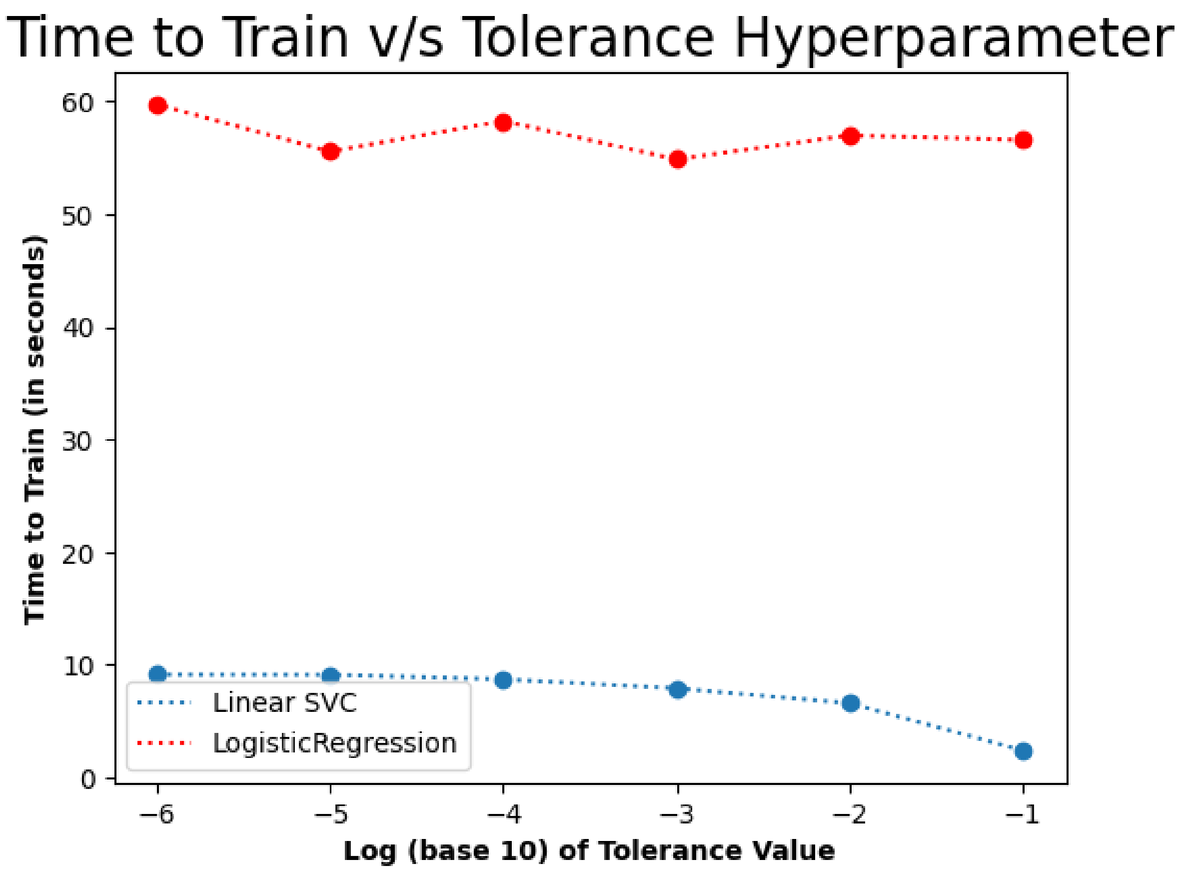


Figure 4: Plot of Time Taken to Train

For `LinearSVC`, the time taken to train decreases with increasing tol. For `LogisticRegression` however, the time taken oscillates slightly around an average time of about 57 seconds. However, `LinearSVC` takes a lot less time compared to `LogisticRegression` making it the better choice for cracking advanced XORRO PUFs.