CS771: Introduction to Machine Learning
Mini-Project
Team: NotAnLLM

Wadkar Srujan Nitin
221212
Dept. of Computer Science and Engineering
IIT Kanpur
nsrujan22@iitk.ac.in

Sanyam
220971
Dept. of Material Sciences and Engineering
IIT Kanpur
sanyam22@iitk.ac.in

Nishtha Gupta
220727
Dept. of Material Sciences and Engineering
IIT Kanpur
nishthag22@iitk.ac.in

Anisurya Jana
220152
Department of Chemical Engineering
IIT Kanpur
anisuryaj22@iitk.ac.in

Abstract

This report presents our solution to the mini-project for the course CS771 which consists of two parts In the first part, we mathematically derive how an ML PUF can be broken using a single linear model. We then use and evaluate different linear model solvers, analyzing their performance and the influence of hyperparameter tuning on training time and prediction accuracy. The second part of the project involves interpreting the learned linear models in terms of physical PUF parameters. Specifically, for each linear model, we are required to generate a list of 256 non-negative real numbers that can be interpreted as delay values of an arbiter PUF. These delays must reproduce the same linear behavior as the corresponding model.

Preprint. Under review.

1 Cracking the ML-PUF

1.1 Introduction

Arbiter Physically Unclonable Functions (PUFs) are hardware-based security mechanisms that leverage inherent, unpredictable variations in signal transmission speeds across different instances of the same circuit design. These systems are built using a series of multiplexers (muxes), each controlled by a selection bit. A specific configuration of these selection bits forms a "challenge" or "question." Based on the challenge, the signal path that reaches the output fastest determines the "response" or "answer." Because these variations are unique to each physical device, the response to any given challenge is also unique to that particular PUF.

1.2 Using ML to crack the Arbiter PUFs

As shown in class, a model can be trained to predict answers to any other challenge for a particular arbiter PUF given a relatively small number of challenges , the analysis for which is shown below:

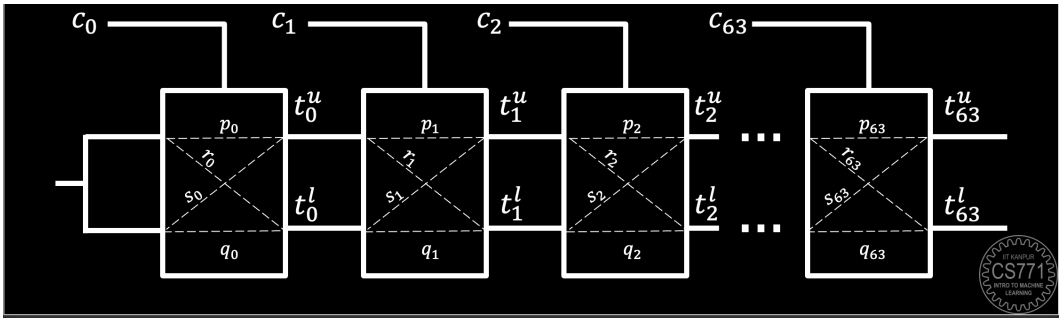

Figure 1: Analysis – Arbiter PUF

Let t_i^u denote the (unknown) time at which the upper signal exits the i-th multiplexer, and t_i^l the time for the lower signal. Since all multiplexers are distinct, their associated delays p_i, q_i, r_i, s_i are unique.

The final output bit is determined based on the sign of the timing difference at stage 7, denoted by $\Delta_8 = t_7^u - t_7^l$:

- If $\Delta_7 < 0$, the output is 0.
- Otherwise, the output is 1.

Starting with stage 1, the upper and lower signal timings are given by:

$$t_1^u = (1 - c_1)(t_0^u + p_1) + c_1(t_0^l + s_1)$$
$$t_1^l = (1 - c_1)(t_0^l + q_1) + c_1(t_0^u + r_1)$$

Define the timing difference at stage i as:

$$\Delta_i = t_i^u - t_i^l$$

Then, for stage 1:

$$\begin{aligned}\Delta_1 &= (1 - c_1)(t_0^u + p_1 - t_0^l - q_1) + c_1(t_0^l + s_1 - t_0^u - r_1) \\ &= (1 - c_1)(\Delta_0 + p_1 - q_1) + c_1(-\Delta_0 + s_1 - r_1) \\ &= (1 - 2c_1)\Delta_0 + c_1(q_1 - p_1 + s_1 - r_1) + (p_1 - q_1)\end{aligned}$$

Let:

$$d_i = 1 - 2c_i$$
$$\alpha_i = \frac{p_i - q_i + r_i - s_i}{2}, \quad \beta_i = \frac{p_i - q_i - r_i + s_i}{2}$$

Then, we can rewrite:

$$\Delta_i = d_i \cdot \Delta_{i-1} + \alpha_i \cdot d_i + \beta_i$$

Assuming $\Delta_{-1} = 0$, by incorporating initial delays into p_0, q_0, r_0, s_0, we recursively compute:

$$\Delta_0 = \alpha_0 \cdot d_0 + \beta_0$$

$$\Delta_1 = \Delta_0 \cdot d_1 + \alpha_1 \cdot d_1 + \beta_1$$

Substituting for Δ_0:

$$\Delta_1 = (\alpha_0 \cdot d_0 \cdot d_1) + (\alpha_1 + \beta_0) \cdot d_1 + \beta_1$$

Continuing this pattern:

$$\Delta_2 = \alpha_0 \cdot d_0 \cdot d_1 \cdot d_2 + (\alpha_1 + \beta_0) \cdot d_1 \cdot d_2 + (\alpha_2 + \beta_1) \cdot d_2 + \beta_2$$

From this, we see that Δ_8 can be expressed in the form:

$$\Delta_7 = \mathbf{w}^\top \mathbf{x} + b$$

where:

$$x_i = d_i \cdot d_{i+1} \cdot \ldots \cdot d_8$$

$$w_0 = \alpha_0, \quad w_i = \alpha_i + \beta_{i-1} \quad \text{for } i > 0$$

1.3 Deriving a Linear Model to crack ML-PUF

An **ML-PUF** utilizes two independent arbiter PUFs, (we refer to them as PUF_0 and PUF_1. Each of these PUFs consists of its own series of multiplexers with unique internal delays.

For a given challenge input, the same challenge is simultaneously applied to both PUF_0 and PUF_1. Each PUF generates two signals—an upper and a lower signal—that propagate through its delay chain. These outputs are then fed into a pair of arbiters:

- $Arbiter_0$ compares the *lower* signals from PUF_0 and PUF_1. The output of this comparison is called $Response_0$.
- $Arbiter_1$ compares the *upper* signals from PUF_0 and PUF_1. The output of this comparison is called $Response_1$.

Each arbiter produces a response based on which signal arrives first. If the signal from PUF_0 arrives before the corresponding signal from PUF_1, the arbiter outputs 0; otherwise, it outputs 1.

Finally, the ML-PUF computes the XOR of $Response_0$ and $Response_1$ to generate the final output bit:

$$\text{Output} = Response_0 \oplus Response_1$$

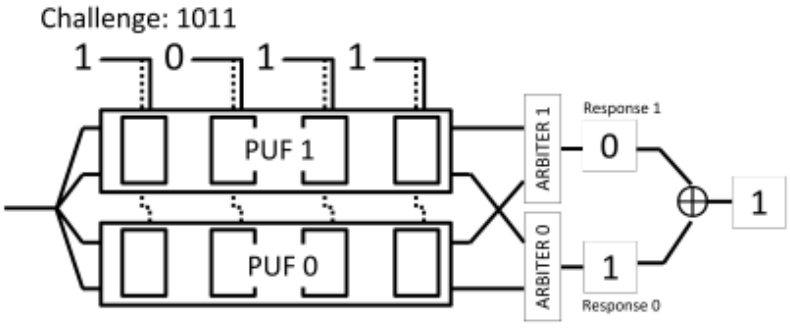

Figure 2: Analysis – ML PUF

In this section, we demonstrate that an ML-PUF can be modeled and potentially cracked using a linear machine learning (ML) model. According to the setup, the ML-PUF under consideration

operates on an 8-bit challenge. But we will conduct our analysis based on an ML-PUF with n-bit challenge inputs.

In the previous section we had shown that the time at which the upper signal leaves stage i for an arbiter PUF is given by :

$$t_i^u = (1 - c_i)(t_{i-1}^u + p_i) + c_i(t_{i-1}^l + s_i)$$

Using this we can write the time difference of upper signals between PUF_1 and PUF_0 as,

$$\Delta_i^u = t_{i,1}^u - t_{i,0}^u$$

where $t_{i,j}^u$ represents the time at which the upper signal of PUF_j exits stage i. Hence,

$$\Delta_i^u = (1 - c_i)(t_{i-1,1}^u + p_{i,1}) + c_i(t_{i-1,1}^l + s_{i,1}) - (1 - c_i)(t_{i-1,0}^u + p_{i,0}) + c_i(t_{i-1,0}^l + s_{i,0})$$

$$\Delta_i^u = (1 - c_i)(\Delta_{i-1}^u + \Delta p_i) + c_i(\Delta_{i-1}^l + \Delta s_i)$$

where Δp_i = $p_{i,1}$ - $p_{i,0}$ and Δs_i is defined in a similar manner.

Using $c_i = (1 - d_i)/2$ from the previous section

$$\Delta_i^u = \frac{1}{2}(\Delta_{i-1}^u + \Delta_{i-1}^l + \Delta p_i + \Delta s_i) + \frac{d_i}{2}(\Delta_{i-1}^u - \Delta_{i-1}^l + \Delta p_i - \Delta s_i)$$

Similarly

$$\Delta_i^l = \frac{1}{2}(\Delta_{i-1}^l + \Delta_{i-1}^u + \Delta q_i + \Delta r_i) + \frac{d_i}{2}(\Delta_{i-1}^l - \Delta_{i-1}^u + \Delta q_i - \Delta r_i)$$

We can see that applying recursion for Δ_i^l and Δ_i^u directly is difficult. Hence, we try to apply recursion for $\Delta_i^u + \Delta_i^l$, (call it a_i) and $\Delta_i^u - \Delta_i^l$ (call it b_i). Thus, adding the two equations we get,

$$a_i = a_{i-1} + \frac{1}{2}(\Delta p_i + \Delta q_i + \Delta r_i + \Delta s_i) + \frac{d_i}{2}(\Delta p_i + \Delta q_i - \Delta r_i - \Delta s_i)$$

Let

$$\alpha_i = \frac{\Delta p_i + \Delta q_i + \Delta r_i + \Delta s_i}{2}, \quad \beta_i = \frac{\Delta p_i + \Delta q_i - \Delta r_i - \Delta s_i}{2}$$

Thus,

$$a_i = a_{i-1} + \frac{\alpha_i}{2} + \frac{d_i\beta_i}{2}$$

We can unroll the recurrence:

$$a_0 = \frac{\alpha_0}{2} + \frac{d_0\beta_0}{2}$$

$$a_1 = a_0 + \frac{\alpha_1}{2} + \frac{d_1\beta_1}{2} = \left(\frac{\alpha_0}{2} + \frac{d_0\beta_0}{2}\right) + \frac{\alpha_1}{2} + \frac{d_1\beta_1}{2} = \frac{\alpha_0 + \alpha_1}{2} + \frac{d_0\beta_0 + d_1\beta_1}{2}$$

$$a_2 = a_1 + \frac{\alpha_2}{2} + \frac{d_2\beta_2}{2} = \frac{\alpha_0 + \alpha_1 + \alpha_2}{2} + \frac{d_0\beta_0 + d_1\beta_1 + d_2\beta_2}{2}$$

$$\Rightarrow a_2 = \sum_{i=0}^{2}\left(\frac{\alpha_i}{2} + \frac{d_i\beta_i}{2}\right)$$

In general, for any n, $\quad a_n = \sum_{i=0}^{n}\left(\frac{\alpha_i}{2} + \frac{d_i\beta_i}{2}\right) = \frac{1}{2}\sum_{i=0}^{n}\alpha_i + \frac{1}{2}\sum_{i=0}^{n}d_i\beta_i$

$$\Rightarrow a_n = \mathbf{w}^\top \mathbf{d} + b$$

$$\text{where } \mathbf{w} = \frac{1}{2}\begin{bmatrix}\beta_0\\ \beta_1\\ \vdots\\ \beta_n\end{bmatrix}, \quad \mathbf{d} = \begin{bmatrix}d_0\\ d_1\\ \vdots\\ d_n\end{bmatrix}, \quad b = \frac{1}{2}\sum_{i=0}^{n}\alpha_i$$

The output a_n is a linear function of $d_0, d_1, \ldots, d_n$

Now we subtract the earlier equations to yield b_i as,

$$b_i = b_{i-1}d_i + \frac{1}{2}(\Delta p_i - \Delta q_i + \Delta r_i - \Delta s_i) + \frac{d_i}{2}(\Delta p_i - \Delta q_i + \Delta r_i - \Delta s_i)$$

Let,

$$\gamma = \frac{1}{2}(\Delta p_i - \Delta q_i + \Delta r_i - \Delta s_i), \delta = \frac{1}{2}(\Delta p_i - \Delta q_i + \Delta r_i - \Delta s_i)$$

We get,

$$b_i = b_{i-1}d_i + \frac{\gamma_i}{2} + \frac{d_i\delta_i}{2}$$

We can unroll the recurrence,

$$b_0 = \frac{\gamma_0}{2} + \frac{d_0\delta_0}{2}$$

$$b_1 = b_0d_1 + \frac{\gamma_1}{2} + \frac{d_1\delta_1}{2} = \left(\frac{\gamma_0}{2} + \frac{d_0\delta_0}{2}\right)d_1 + \frac{\gamma_1}{2} + \frac{d_1\delta_1}{2} = \frac{\gamma_0}{2}d_1 + \frac{d_0\delta_0}{2}d_1 + \frac{\gamma_1}{2} + \frac{d_1\delta_1}{2}$$

$$b_2 = b_1d_2 + \frac{\gamma_2}{2} + \frac{d_2\delta_2}{2} = \left(\frac{\gamma_0}{2}d_1 + \frac{d_0\delta_0}{2}d_1 + \frac{\gamma_1}{2} + \frac{d_1\delta_1}{2}\right)d_2 + \frac{\gamma_2}{2} + \frac{d_2\delta_2}{2}$$

$$= \frac{\gamma_0}{2}d_1d_2 + \frac{d_0\delta_0}{2}d_1d_2 + \frac{\gamma_1}{2}d_2 + \frac{d_1\delta_1}{2}d_2 + \frac{\gamma_2}{2} + \frac{d_2\delta_2}{2}$$

$$\Rightarrow b_2 = w_0 \cdot (d_0d_1d_2) + w_1 \cdot (d_1d_2) + w_2 \cdot d_2 + b$$

Where each term is a product of some d_i's, and the coefficients w_i and b are combinations of γ_i and δ_i

$$\text{In general, } b_n = \sum_{i=0}^{n} w_i \cdot \left(\prod_{j=i}^{n} d_j\right) + b$$

That is, b_n is a linear function of $d_n,\ d_nd_{n-1},\ \ldots,\ d_0d_1 \ldots d_n$

We know that, $\Delta^u_{n-1} = a_{n-1} + b_{n-1}$. Since a_{n-1} is a linear function of $d_0, d_1, \ldots, d_{n-1}$ and b_{n-1} is a linear function of $d_{n-1},\ d_{n-1}d_{n-2},\ \ldots,\ d_0d_1 \ldots d_{n-1}$ we conclude that Δ^u_{n-1} is a linear function of $d_0, d_1, \ldots, d_{n-1},\ d_{n-1}d_{n-2},\ \ldots,\ d_0d_1 \ldots d_{n-1}$. Similarly, using $\Delta^l_{n-1} = a_{n-1} - b_{n-1}$ we conclude that Δ^l_{n-1} is a linear function of $d_0, d_1, \ldots, d_{n-1},\ d_{n-1}d_{n-2},\ \ldots,\ d_0d_1 \ldots d_{n-1}$

Hence, Δ^u_{n-1} and Δ^l_{n-1} themselves can be expressed as linear functions over the following feature map:

$$\phi(\mathbf{c}) = [d_0, d_1, \ldots, d_{n-1}, d_{n-1}d_{n-2},\ d_{n-1}d_{n-2}d_{n-3},\ \ldots,\ d_{n-1}d_{n-2} \cdots d_0]$$

where

$$d_i = 1 - 2c_i$$

Therefore, $\phi(\mathbf{c})$ serves as the common feature vector for both Δ^u_{n-1} and Δ^l_{n-1}.

We define the responses as follows:

$$Response_0 = \frac{\text{sign}(\Delta^u_{n-1}) + 1}{2}, \quad Response_1 = \frac{\text{sign}(\Delta^l_{n-1}) + 1}{2}$$

To obtain the final output, we need to compute the XOR of these two responses. As discussed in class, we can achieve this by using a simple trick: multiply Δ^u_{n-1} and Δ^l_{n-1}, and then use the sign of the product. Specifically, we compute:

$$\frac{\text{sign}(\Delta^l_{n-1} \cdot \Delta^u_{n-1}) + 1}{2}$$

Now, from our discussion in class we can see that our final feature map $\tilde{\phi}(c)$ for $(\Delta^l_{n-1} \cdot \Delta^u_{n-1})$ is simply the pairwise product of the original feature maps i.e,

$$\tilde{\phi}(\mathbf{c}) = [\phi(\mathbf{c})_0\phi(\mathbf{c})_1,\ \phi(\mathbf{c})_0\phi(\mathbf{c})_2,\ \ldots,\ \phi(\mathbf{c})_0\phi(\mathbf{c})_{n-1},\ \phi(\mathbf{c})_1\phi(\mathbf{c})_2,\ \ldots,\ \phi(\mathbf{c})_{n-2}\phi(\mathbf{c})_{n-1}]$$

where $\phi(\mathbf{c})$ has been described above. In this feature map we ignore the square terms as d_i can only be 1 or -1 and hence the square would always be 1 which is constant.

We can now use this feature map to obtain $\text{sign}(\Delta^l_{n-1} \cdot \Delta^u_{n-1})$ as

$$\Delta^u_{n-1} \cdot \Delta^l_{n-1} = \mathbf{w}^\top \tilde{\phi}(\mathbf{c}) + b$$

We can now put $n = 8$ for our purpose. This concludes the mathematical derivation.

Note: One might argue that while computing the product

$$\Delta^u_{n-1} \cdot \Delta^l_{n-1} = (\mathbf{w}_u^\top \phi(\mathbf{c}) + b_u)(\mathbf{w}_l^\top \phi(\mathbf{c}) + b_l),$$

the expansion would include not only the pairwise products (i.e., elements of $\tilde{\phi}(\mathbf{c})$), but also individual terms from $\phi(\mathbf{c})$ and a constant bias term $b_u b_l$. This would seemingly require us to include the elements of $\phi(\mathbf{c})$ itself in the final feature map $\tilde{\phi}(\mathbf{c})$.

However, upon closer inspection, we observe that such terms are implicitly accounted for in the pairwise structure. Specifically, since each $d_i \in \{-1, 1\}$, we have $d_i^2 = 1$, and thus any squared or redundant product terms collapse to constants. For instance, a term like

$$d_i = (d_i d_{i+1} \ldots d_{n-1}) \cdot (d_{i+1} d_{i+2} \ldots d_{n-1})$$

$$d_i d_{i+1} \ldots d_{n-1} = d_i \cdot (d_{i+1} d_{i+2} \ldots d_{n-1})$$

illustrates how a single bit d_i as well as the suffix product starting from d_i can be expressed as a product of terms that are already present in the feature map. Hence, these linear terms from $\phi(\mathbf{c})$ do not need to be explicitly added again, as they are subsumed in $\tilde{\phi}(\mathbf{c})$ due to the algebraic properties of $\{-1, 1\}$ variables.

2 Deriving the dimensionality

From the previous section, we observe that the intermediate feature map $\phi(\mathbf{c})$, used for both Δ^u_{n-1} and Δ^l_{n-1}, has dimensionality $2n - 1$.

This is because $\phi(\mathbf{c})$ contains:

- The n linear terms: $d_0, d_1, \ldots, d_{n-1}$
- The n suffix product terms: $d_{n-1},\ d_{n-1}d_{n-2},\ \ldots,\ d_{n-1}d_{n-2}\cdots d_0$

Since the term d_{n-1} is shared between the two sets above, it is counted twice.

Therefore, the total number of distinct terms in the feature map $\phi(\mathbf{c})$, i.e., its dimensionality, is:

$$n + (n - 1) = 2n - 1$$

The final feature map, denoted as $\tilde{\phi}(\mathbf{c})$, is constructed by taking all pairwise products of the intermediate feature map $\phi(\mathbf{c})$. The total number of such pairwise products is given by the number of ways to choose 2 elements from $\phi(\mathbf{c})$.

Since the dimensionality of $\phi(\mathbf{c})$ is $2n-1$, the dimensionality of the final feature map $\tilde{\phi}(\mathbf{c})$ is:

$$\dim(\tilde{\phi}(\mathbf{c})) = \binom{\mathbf{2n-1}}{\mathbf{2}} = \mathbf{(2n-1)(n-1)}$$

For $n = 8$, we compute:

$$\dim(\tilde{\phi}(\mathbf{c})) = \binom{\mathbf{15}}{\mathbf{2}} = \mathbf{105}$$

Hence, the dimensionality of our final feature map $\tilde{\phi}(\mathbf{c})$ for $n = 8$ is **105**.

3 Choosing an appropriate Kernel SVM

From our previous analysis, we concluded that the output depends linearly on $d_0, d_1, \ldots, d_{n-1}$ and their cumulative products (i.e., suffix products). Substituting $d_i = 1 - 2c_i$, we observe that the terms involved in the feature map are simply products of some subset of the input binary variables (e.g., $c_0c_3c_7$, etc.).

Therefore, a possible choice for feature map is:

$$\phi(\mathbf{c}) = [c_0, c_1, \ldots, c_{n-1},\ c_0c_1,\ c_0c_2,\ \ldots,\ c_0c_1c_2,\ \ldots,\ c_0c_1 \cdots c_{n-1}]$$

This essentially captures all possible products (monomials) over subsets of the input binary variables.

Now, consider the dot product of two input challenges $\mathbf{x}$ and $\mathbf{y}$:

$$K(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^\top \phi(\mathbf{y})$$

The resulting dot product $K(\mathbf{x}, \mathbf{y})$ will consist of addition of terms like $c_0^x c_0^y, c_1^x c_1^y, \ldots, c_0^x c_1^x \cdots c_{n-1}^x \cdot c_0^y c_1^y \cdots c_{n-1}^y$.

Since the dot product involves a polynomial combination of binary variables, the **polynomial kernel SVM** is a perfect fit for our problem as it can produce the required dot product:

$$K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^\top \mathbf{y} + r)^d = (c_0^x c_0^y + c_1^x c_1^y, + \ldots + c_{n-1}^x c_{n-1}^y + r)^d$$

Choice of hyperparameters

- **Degree (d):** Our feature map includes all possible products of up to n input variables and to make sure the kernel captures these highest-order terms, the degree of the polynomial kernel must be at least n. But for a polynomial kernel the lower the degree the better and hence we set it to the lowest possible value n. Thus, $d = n$.
- **Coefficient (r):** We set $r = 1$ to ensure that the kernel includes not only the highest-degree interaction terms (like $c_0c_1...c_{n-1}$), but also all lower-degree combinations (e.g., c_0c_1, c_0, etc.). This is important because our feature map involves monomials of all degrees—from individual variables to full-length products. A non-zero r (like $r = 1$) allows the SVM to learn from all levels of interactions, not just the highest-order ones.

Thus, a polynomial kernel of degree n allows us to compute the dot product in the high-dimensional feature space implicitly, without explicitly constructing $\phi(\mathbf{c})$.

4 Recovering delays from a model

From our analysis in the first section, we know that the weights of the linear model that successfully predicts the output of an Arbiter PUF can be expressed as:

$$w_0 = \alpha_0, \quad w_i = \alpha_i + \beta_{i-1} \quad \text{for } i > 0$$
$$b = \beta_{n-1}$$

where the terms α_i and β_i are defined as:

$$\alpha_i = \frac{p_i - q_i + r_i - s_i}{2}, \quad \beta_i = \frac{p_i - q_i - r_i + s_i}{2}$$

In this problem, we are provided with the weights and bias of various Arbiter PUFs in the form of a 65-dimensional vector. Our task is to find a possible delay configuration for each of the delay components p, q, r, s, represented as 64-dimensional vectors with non-negative values, such that the above equations are satisfied.

Initially, we relax the non-negativity constraint and focus on finding any solution.

We have 65 equations but 256 unknowns (64 each for p, q, r, and s). To reduce the number of variables, we fix some of them:

- - Set $q_i = r_i = s_i = 0$ for $i = 0$ to 62.
- - Fix $p_0 = 0$.

Using the relation:

$$p_i = 2w_i - p_{i-1}$$

We compute p_i recursively for $i = 1$ to 62.

To determine p_{63} and r_{63}, we fix $q_{63} = s_{63} = 0$, simplifying the last two equations:

$$w_{63} = \frac{p_{63} + r_{63}}{2} + \frac{p_{62}}{2}, \quad w_{64} = \frac{p_{63} - r_{63}}{2}$$

Solving these:

$$p_{63} = w_{63} + w_{64} - \frac{p_{62}}{2}, \quad r_{63} = p_{63} - 2w_{64}$$

After computing all delays, we check for negative values. A key observation is that the weights depend only on differences between delays and adding a constant to all delays doesn't affect the solution. Therefore, if any delay is negative, we add the absolute value of the most negative delay to all delays to make them non-negative. If all delays are already non-negative, no adjustment is needed.

Thus we have obtained a valid delay configuration satisfying all the required constraints for a given model.

5 Code for ML-PUF

Code submitted

6 Code for recovering delays

Code submitted

7 Experimentation and Analysis of two linear models

The model is trained using `sample_trn.txt` and evaluated on `sample_tst.txt`, with different hyperparameter configurations. For each configuration, both the training time and test accuracy are recorded. All the results were estimated by taking average over 5 iterations (as done in evaluation code)

7.1 Effect of changing the $loss$ hyperparameter in `LinearSVC`

The **loss function** defines how the model evaluates its prediction errors during training. The `sklearn.svm.LinearSVC` classifier supports two primary loss functions: **hinge** and **squared hinge**.

We conducted experiments using the following hyperparameters for **LinearSVC**: `C = 1`, `tolerance = 1e-3`, `penalty = L2`, and `dual = True`. The results were calculated by taking average over 5 iterations (as done in the evaluation code) and are summarized below.

Table 1: Comparison of Loss Functions in LinearSVC

Loss Function	Training Time (in seconds)	Accuracy
Hinge	0.466	1.00
Squared Hinge	1.242	1.00

7.2 Effect of Changing the C Hyperparameter in `LinearSVC` and `LogisticRegression`

The regularization parameter C controls the trade-off between minimizing training error and achieving good generalization.

- A smaller C enforces stronger regularization, potentially underfitting the data.
- A larger C reduces regularization, allowing more flexibility, but may lead to overfitting and increased training time.

The following hyperparameters were fixed for the models:

- **LinearSVC**: `tol=1e-3`, `penalty='l2'`, `max_iter=10000`, `loss='hinge'`, `dual=True`
- **LogisticRegression**: `solver='lbfgs'`, `max_iter=1000`, `tol=1e-3`, `penalty='l2'`

Table 2: Effect of C on Training Time and Accuracy

Model	C Value	Training Time (s)	Accuracy
LinearSVC	0.01	0.073	1.00
	0.1	0.347	1.00
	1	0.573	1.00
	10	0.548	1.00
	100	0.561	1.00
LogisticRegression	0.01	0.194	1.00
	0.1	0.387	1.00
	1	0.252	1.00
	10	0.167	1.00
	100	0.175	1.00

7.3 Effect of changing the `tol` hyperparameter

The **tolerance** parameter (`tol`) determines the threshold for the stopping condition of the optimization solver. A lower tolerance leads to more accurate convergence but may increase training time, while a higher value may speed up convergence at the cost of precision.

We conducted experiments by varying the `tol` value for both **LinearSVC** and **LogisticRegression**. The other hyperparameters were fixed as follows:

- **LinearSVC:** `C = 1`, `loss = 'hinge'`, `penalty = 'l2'`, `dual = True`, `max_iter = 10000`
- **LogisticRegression:** `C = 1`, `solver = 'lbfgs'`, `penalty = 'l2'`, `max_iter = 1000`

Table 3: Effect of `tol` on Training Time and Accuracy

Model	`tol` Value	Training Time (s)	Accuracy
LinearSVC	$1e{-1}$	0.197	1.00
	$1e{-2}$	0.287	1.00
	$1e{-3}$	0.517	1.00
	$1e{-4}$	0.647	1.00
	$1e{-5}$	1.061	1.00
LogisticRegression	$1e{-1}$	0.014	1.00
	$1e{-2}$	0.134	1.00
	$1e{-3}$	0.242	1.00
	$1e{-4}$	0.994	1.00
	$1e{-5}$	1.499	1.00

7.4 Effect of changing the `penalty` hyperparameter

The **penalty** term determines the type of regularization applied to avoid overfitting. `L2` regularization penalizes the squared magnitude of the coefficients, while `L1` encourages sparsity in the model by driving some coefficients to zero.

We evaluated the models using both `L1` and `L2` penalties. The other hyperparameters were kept constant:

- **LinearSVC:** `C = 1`, `loss = 'hinge'`, `tol = 1e-3`, `dual = True/False`, `max_iter = 10000`
- **LogisticRegression:** `C = 1`, `solver = 'liblinear'`, `tol = 1e-3`, `max_iter = 1000`

Table 4: Effect of `penalty` on Training Time and Accuracy

Model	Penalty Type	Training Time (s)	Accuracy
LinearSVC	L1	4.714	1.00
	L2	0.465	1.00
LogisticRegression	L1	0.831	1.00
	L2	0.340	1.00

Note: In `LinearSVC`, the valid combinations of the `penalty` and `dual` parameters are:

- `penalty = 'l2'` with `dual = True` (default and most common)
- `penalty = 'l1'` only when `dual = False`

Using an invalid combination such as `penalty = 'l1'` with `dual = True` or `penalty = 'l2'` with `dual = False` (and `loss='hinge'`) will raise a `ValueError`.