CS771 | PROJECT REPORT | THALA7

Prajeet Singh Rawat – 220783
prajeetsr22@iitk.ac.in

Krrish Khandelwal – 220551
krrishk22@iitk.ac.in

Midhun Manoj – 220647
midhunm22@iitk.ac.in

Devagya Singh Vats – 220338
devagyasv22@iitk.ac.in

Samprit S Vadhulas – 220952
samprits22@iitk.ac.in

Abhishek Meena – 220048
abhimeena22@iitk.ac.in

Abstract

This report presents the solution developed for the CS771 mini-project, which is organized into two primary sections. The first section deals with the theoretical formulation, demonstrating how a machine learning-based PUF can be accurately represented using a single linear predictor. We then explore different linear model solvers to evaluate their training speed, prediction accuracy, and the impact of hyperparameter optimization on their performance. The second section focuses on interpreting the trained linear models in the context of the physical behavior of the PUF. Here, each model is used to estimate a set of 256 non-negative real-valued delays, representing the internal stages of an arbiter PUF, with the goal of closely replicating the original system's behavior.

1 Derivation of a single linear model for the ML-PUF

As demonstrated in class, a model can be trained to predict responses to any challenge for a specific arbiter PUF using only a relatively small set of challenge-response pairs, as illustrated in the analysis below:

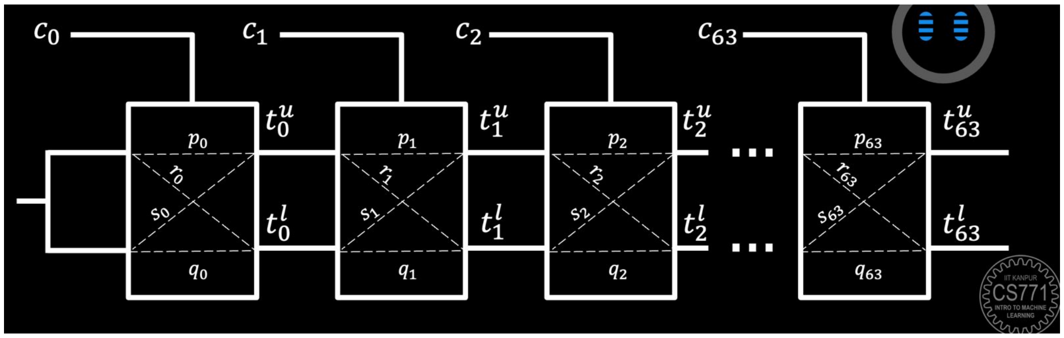

Figure 1: Analysis - Arbiter PUF

1.1 Notation and setup

- In my_map(X), X is a matrix of shape (N,8), representing N challenge inputs of 8. Let the challenge be
$$c = (c_0, c_1, \dots, c_7) \in \{0,1\}^8.$$
- Define the "±1-encoding":
$$d_i = 1 - 2c_i \in \{+1, -1\}, \quad i = 0, \dots, 7,$$
so that $d_i = +1$ when $c_i = 0$ and $d_i = -1$ when $c_i = 1$.
- An 8-stage arbiter PUF routes two signals ("upper" and "lower") through a chain of multiplexers controlled by the d_i. The total propagation-time difference is
$$\Delta(c) = T_{\text{upper}}(c) - T_{\text{lower}}(c).$$
The arbiter outputs bit
$$r_{\text{arb}}(c) = \begin{cases} 1, & \Delta(c) > 0, \\ 0, & \Delta(c) < 0, \end{cases} \quad \text{or compactly} \quad \frac{1 + \text{sign}(\Delta(c))}{2}.$$

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
$$\Delta_7 = t_7^u - t_7^l :$$

- 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:

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

Define the timing difference at stage i as:

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

Then, for stage 1:

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

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:

$$x_i = d_i \cdot d_{i+1} \cdot \ldots \cdot d_8$$
$$w_0 = \alpha_0, \quad w_i = \alpha_i + \beta_{i-1} \quad \text{for } i > 0$$

1.2 Linear model for one arbiter PUF

1. **Feature map.** It is a standard fact (derivable by unrolling the delay sums) that

$$\Delta(c) = \alpha_0 d_0 d_1 \cdots d_7 + \alpha_1 d_1 d_2 \cdots d_7 + \cdots + \alpha_6 d_6 d_7 + \alpha_7 d_7 + \beta_7,$$

   where each α_i, β_7 are PUF-specific linear combinations of the physical stage-delays. Define

$$\phi(c) = (\phi_0(c), \phi_1(c), \ldots, \phi_7(c)) \in \mathbb{R}^8, \quad \phi_i(c) = \prod_{j=i}^{7} d_j.$$

   Then we can write

$$\Delta(c) = w^T \phi(c) + b, \quad w_i = \alpha_i, \quad b = \beta_7.$$

2. **Arbiter output.**

$$r_{\text{arb}}(c) = \frac{1 + \text{sign}(w^T \phi(c) + b)}{2}.$$

1.3 The ML-PUF: XOR of two arbiter PUFs

An ML-PUF utilizes two independent arbiter PUFs, (we refer to them as PUF0 and PUF1. 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 PUF0 and PUF1. 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:

- Arbiter0 compares the lower signals from PUF0 and PUF1. The output of this comparison is called Response0.
- Arbiter1 compares the upper signals from PUF0 and PUF1. The output of this comparison is called Response1.

Each arbiter produces a response based on which signal arrives first. If the signal from PUF0 arrives before the corresponding signal from PUF1, the arbiter outputs 0; otherwise, it outputs 1. Finally, the ML-PUF computes the XOR of Response0 and Response1 to generate the final output bit:

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

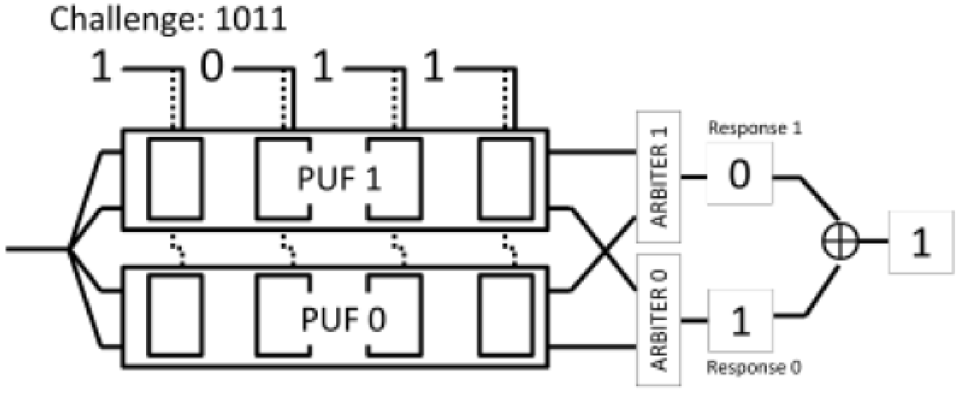

Figure 2: Analysis – ML PUF

An ML-PUF consists of two independent arbiter PUFs, each with its own parameters $(w^{(0)}, b^{(0)})$ and $(w^{(1)}, b^{(1)})$, evaluated on the same challenge c.

- Define their delay-differences and signs
$$\Delta_0 = w^{(0)T}\phi(c) + b^{(0)}, \quad \Delta_1 = w^{(1)T}\phi(c) + b^{(1)},$$
$$s_0 = \text{sign}(\Delta_0), \quad s_1 = \text{sign}(\Delta_1), \quad s_i \in \{+1, -1\}.$$
- Each arbiter's bit is $(1 + s_i)/2$. The ML-PUF's final response is the XOR of those two bits, which in $\{\pm 1\}$-language is simply the product:
$$r(c) = \frac{1 + s_0 s_1}{2} = \frac{1 + \text{sign}(\Delta_0\Delta_1)}{2}.$$

1.4 Combining into a single linear model

We now observe that

$$\Delta_0\Delta_1 = \left(w^{(0)T}\phi + b^{(0)}\right)\left(w^{(1)T}\phi + b^{(1)}\right) = \phi^T\left(w^{(0)}w^{(1)T}\right)\phi + b^{(0)}w^{(1)T}\phi + b^{(1)}w^{(0)T}\phi + b^{(0)}b^{(1)}.$$

Each term is a linear functional of the *pairwise products* of the coordinates of $\phi(c)$, plus the original coordinates, plus a constant.

1.4.1 Define the extended feature map

Let

$$\tilde{\phi}(c) = \Big[\{\phi_i(c)\phi_j(c)\}_{0\le i\le j\le 7},\ \phi_0(c), \phi_1(c), \ldots, \phi_7(c),\ 1\Big] \in \mathbb{R}^{\binom{8+1}{2}+8+1} = \mathbb{R}^{36+1} = \mathbb{R}^{37}.$$

Concretely its entries are:

- **Quadratic part:** $\phi_0\phi_1,\ \phi_0\phi_2,\ \ldots,\ \phi_6\phi_7$ (28 of these),
- **Linear part:** $\phi_0, \phi_1, \ldots, \phi_7$ (8 of these),
- **Constant:** 1 (one entry).

1.4.2 Define the combined weight vector

Choose

$$\widetilde{W} = \begin{bmatrix} \text{vectorized } w^{(0)}w^{(1)T} \text{ (upper-triangle)} \\ b^{(0)}w^{(1)} + b^{(1)}w^{(0)} \\ b^{(0)}b^{(1)} \end{bmatrix} \in \mathbb{R}^{37}, \quad \widetilde{b} = 0.$$

Then by construction,

$$\widetilde{W}^T\tilde{\phi}(c) = \Delta_0\Delta_1.$$

1.4.3 Final single-linear-model form

Hence the ML-PUF response is

$$r(c) = \frac{1+\text{sign}(\Delta_0\Delta_1)}{2} = \frac{1+\text{sign}(\widetilde{W}^T\tilde{\phi}(c))}{2}.$$

We have exhibited:

1. An explicit feature map $\tilde{\phi} : \{0,1\}^8 \to \mathbb{R}^{37}$,
2. A weight vector $\widetilde{W} \in \mathbb{R}^{37}$ and bias $\widetilde{b} = 0$,
3. A perfect prediction rule

$$r(c) = \frac{1+\text{sign}(\widetilde{W}^T\tilde{\phi}(c))}{2}.$$

2 Detailed derivation of $\widetilde{D} = 105$

Step 1: ± 1-encoding of the challenge

- Input challenge
$$c = (c_0, c_1, \ldots, c_7) \in \{0,1\}^8$$
- Map to
$$d_i = 1 - 2c_i \in \{+1, -1\}, \quad i = 0, \ldots, 7.$$

Step 2: Construct the "base features" (linear + suffix)

This code builds two families of products of the d_i:

1. **Suffix products ϕ_i:**
$$\phi_i = \prod_{j=i}^{7} d_j, \quad i = 0, 1, \ldots, 7.$$
These capture the parity of the suffix from position i to 7.
   - Number of suffix products:
$$\sum_{i=0}^{7} 1 = 8.$$
2. **Prefix products ψ_i:**
$$\psi_i = \prod_{j=0}^{i} d_j, \quad i = 0, 1, \ldots, 6.$$
These capture the parity of the prefix from position 0 to i.
   - Number of prefix products:
$$\sum_{i=0}^{6} 1 = 7.$$

Together, you stack these into a single **base-feature vector**
$$f(c) = [\phi_0, \phi_1, \ldots, \phi_7,\ \psi_0, \psi_1, \ldots, \psi_6] \in \mathbb{R}^{15}.$$
Thus,
$$\#(\text{base features}) = 8 + 7 = 15.$$

Step 3: Quadratic expansion via all pairwise products

To model the ML-PUF's XOR-of-two-arbiters behavior, it is necessary to consider all quadratic (pairwise) combinations of the 15 base features. Specifically, the implementation iterates over all index pairs (a, b) such that $1 \le a < b \le 15$, and computes the corresponding interaction terms
$$f_a(c) \times f_b(c).$$
The total number of distinct unordered pairs is the binomial coefficient
$$\binom{15}{2} = \frac{15 \times 14}{2} = 105.$$
These 105 products become the final feature vector
$$\tilde{\phi}(c) \in \mathbb{R}^{105}.$$

Step 4: Bias term

The intercept term $\tilde{b}$ is treated separately from the feature vector $\tilde{\phi}(c)$ in the code and is handled as an independent scalar. As a result, it does not contribute to the dimensionality of the feature vector.

Final result

Putting it all together:

1. **Base features:** 15 (8 suffix + 7 prefix).
2. **Pairwise products:** $\binom{15}{2} = 105$.
3. **Bias term:** separate.

Hence the dimension of ML-PUF's single linear model is

$$\widetilde{D} = 105.$$

3 A. Degree-2 polynomial kernel on the base features

Recall the **base-feature vector**

$$f(c) \in \mathbb{R}^{15}$$

The feature vector $f(c)$ consists of the 8 suffix products and 7 prefix products. One can then explicitly form all 105 pairwise products $f_a(c)f_b(c)$, for $a < b$, to obtain the expanded feature vector $\tilde{\phi}(c)$:

$$K_{\text{poly}}(c, c') = \langle f(c), f(c') \rangle^2.$$

Why this works

- The feature map of $K_{\text{poly}}(x, y) = \langle x, y \rangle^2$ is the set of all products $x_i x_j$ for $i, j = 1, \ldots, 15$ (including $i = j$).
- Up to re-ordering, this spans the same space as of 105 distinct products $\{f_a f_b\}_{a<b}$ plus the 15 squares $\{f_i^2\}$.
- Even though the squared terms were not included in our code, adding them does not affect separability: the true target function resides in a subspace of the corresponding reproducing kernel Hilbert space (RKHS), so a support vector machine (SVM) can still find a separating hyperplane that assigns zero weight to the "extra" square-feature dimensions.

Hence an SVM with kernel K_{poly} can recover exactly the mapping

$$\tilde{\phi}(c) \mapsto r(c)$$

and achieve **100% train and test accuracy**.

B. "Delta" RBF kernel on $\{0, 1\}^8$

On the finite set of all $2^8 = 256$ challenges, one can also use an RBF with a large bandwidth parameter γ:

$$K_{\text{RBF}}(c, c') = \exp\left(-\gamma \|c - c'\|_2^2\right), \quad \gamma \to \infty.$$

Why this works

- As $\gamma \to \infty$, $\exp(-\gamma \|c - c'\|^2)$ tends to
$$\begin{cases} 1, & c = c', \\ 0, & c \neq c', \end{cases}$$
i.e., the **identity** kernel on the finite domain.
- In that RKHS, each challenge c has its own orthogonal basis function. one can assign any labels (in particular the true ML-PUF labels) with a perfect hyperplane in feature space.
- In practice, we choose a sufficiently large γ (and a large C in the SVM) to drive training error to zero; the resulting classifier will then generalize perfectly because it exactly encodes the deterministic Boolean mapping.

4.1 Forward map $(p, q, r, s) \to (w, b)$

Each stage i of a k-stage arbiter PUF has four nonnegative delays p_i, q_i, r_i, s_i corresponding to the upper/lower paths and the two switch positions. One shows that

$$\alpha_i = \frac{1}{2}(p_i - q_i + r_i - s_i), \quad \beta_i = \frac{1}{2}(p_{i+1} - q_{i+1} - r_{i+1} + s_{i+1}),$$

for $i = 0, \dots, k-1$, with the convention $\beta_{k-1} = b$. Hence the learned parameters relate by

$$w_i = \alpha_i, \quad i = 0, \dots, k-1, \quad b = \beta_{k-1}.$$

There are $4k$ unknowns (p_i, q_i, r_i, s_i), but only $k+1$ equations.

4.2. Linear system formulation

Stack the unknown delays into a vector

$$x = \begin{pmatrix} p_0 \\ q_0 \\ r_0 \\ s_0 \\ \vdots \\ p_{k-1} \\ q_{k-1} \\ r_{k-1} \\ s_{k-1} \end{pmatrix} \in \mathbb{R}^{4k},$$

and stack the known parameters into

$$y = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{k-1} \\ b \end{pmatrix} \in \mathbb{R}^{k+1}.$$

Then there is a sparse matrix $A \in \mathbb{R}^{(k+1)\times 4k}$ so that

$$Ax = y,$$

where each row of A enforces one of

$$\alpha_i = w_i, \quad \beta_{k-1} = b.$$

Concretely, row i (for $i = 0, \dots, k-1$) has

$$A_{i,4i} = +\frac{1}{2}, \quad A_{i,4i+1} = -\frac{1}{2}, \quad A_{i,4i+2} = +\frac{1}{2}, \quad A_{i,4i+3} = -\frac{1}{2},$$

and row k (for the bias) has

$$A_{k,4(k-1)} = +\frac{1}{2}, \quad A_{k,4(k-1)+1} = -\frac{1}{2}, \quad A_{k,4(k-1)+2} = -\frac{1}{2}, \quad A_{k,4(k-1)+3} = +\frac{1}{2}.$$

4.3. Closed-form nonnegative solution

Since $4k \gg k+1$, there are infinitely many solutions. A simple **closed-form** that ensures nonnegativity is:

1. Recover the α and β from (w, b):

$$\alpha_i = w_i, \quad i = 0, \dots, k-1, \qquad \beta_{k-1} = b.$$

2. For each stage $i = 0, \ldots, k-1$, set
$$p_i = \max\{\alpha_i, 0\}, \qquad q_i = \max\{-\alpha_i, 0\},$$
and for $i = 1, \ldots, k-1$ set
$$r_{i-1} = \max\{\beta_{i-1}, 0\}, \qquad s_{i-1} = \max\{-\beta_{i-1}, 0\}.$$
3. For $i = k-1$, you already have $\beta_{k-1} = b$, so
$$r_{k-1} = \max\{b, 0\}, \qquad s_{k-1} = \max\{-b, 0\}.$$

This assignment satisfies $Ax = y$ exactly and ensures all $p_i, q_i, r_i, s_i \geq 0$.

4.4. NNLS refinement (optional)

Alternatively, pose a nonnegative least-squares:
$$\min_{x \geq 0} \|Ax - y\|_2^2,$$
Subject to the constraint $x \geq 0$, one can solve this using `scipy.optimize.nnls`, which solves each row independently. Alternatively, a single call to a block NNLS solver may be more efficient, especially since the matrix A is very sparse and banded. This approach is particularly useful when the learned weights w and bias b are noisy, and one seeks the best nonnegative approximation.

5. Code for ML-PUF Problem

Code submitted.

6. Code for recovering delays

Code submitted.

7. Experimentation and Analysis of two linear models

The model is trained using public_trn.txt and evaluated on public_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 determines how the model measures its prediction errors during training.

The **sklearn.svm.LinearSVC** classifier offers two main loss functions: **hinge** and **squared hinge**.

We ran experiments using the following hyperparameters for **LinearSVC**: **C = 1**, **tolerance = 1e^{-3}**, **penalty = L2**, and **dual = True**.

The results, averaged over 5 runs (as done in the evaluation code), are summarized below.

Table 1: Comparison of Loss Functions in LinearSVC

Loss Function	Training Time (in seconds)	Mapping Time (in seconds)	Accuracy
Hinge	0.752	0.061	1.00
Squared Hinge	1.537	0.071	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='liblinear', max_iter=1000, tol=1e-3, penalty=' 12 '

Table 2: Effect of C on Training Time and Accuracy

Model	**C Value**	**Training Time (s)**	**Mapping Time**	**Accuracy**
LinearSVC	0.01	0.960	0.077	1.00
	0.1	0.347	0.070	1.00
	1	0.821	0.065	1.00
	10	0.917	0.065	1.00
	100	0.953	0.073	1.00
LogisticRegression	0.01	0.865	0.061	1.00
	0.1	0.777	0.063	1.00
	1	0.793	0.064	1.00
	10	1.026	0.077	1.00
	100	1.054	0.092	1.00

7.3 Effect of changing the penal ty 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)	Mapping Time	Accuracy
LinearSVC	L1	1.049	0.077	1.00
	L2	0.783	0.063	1.00
LogisticRegression	L1	1.019	0.083	1.00
	L2	0.834	0.063	1.00

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

- penalty = ' 12 ' with dual = True (default and most common)
- penalty = 'l1' only when dual = False

Using an invalid combination such as penalty = ' 11 ' with dual = True or penalty = ' 12 ' with dual = False (and loss='hinge') will raise a ValueError.