ML-PUF Linear Model

CS771 Course Project: Group MLNG89

Project Report

Authors: Shiuli Kumar (**221010**) Samhitha Pathloth (**220951**) Raunak Jalan (**220875**) Abhishek Bharadwaj (**210034**) Pradhumna Bakshi (**210254**) Vaibhav Kadiyan (**211135**)

Part 1: Predicting ML-PUF Response

$$\begin{aligned}
t_i^u &= \left(t_{i-1}^u + p_i\right)(1-c_i) + c_i\left(t_{i-1}^i + s_i\right) \\
t_i^l &= \left(t_{i-1}^l + q_i\right)(1-c_i) + c_i\left(t_{i-1}^u + r_i\right) \\
\Delta_i^u &= t_i^u - t_i^l \qquad\qquad \text{where,} \quad \Delta_i^u \text{ is the lag} \\
\Delta_i^u &= (1-c_i)\left(\Delta_{i-1} + p_i - q_i\right) + c_i\left(s_i - r_i - \Delta_{i-1}\right) \\
\Delta_i^u &= (1-2c_i)\,\Delta_{i-1} + \left(q_i - p_i + s_i - r_i\right)c_i + \left(p_i - q_i\right)
\end{aligned}$$

assuming

$$\begin{aligned}
\alpha_i &= \left(p_i - q_i + r_i + s_i\right)/2 \\
\beta_i &= \left(p_i - q_i - r_i + s_i\right)/2
\end{aligned}$$

Solving the arbiter PUF gives the above solution as

$$\begin{aligned}
\Delta_T &= \omega_0 \cdot x_0 + \omega_1 \cdot x_1 + \ldots + \omega_7 \cdot x_7 + \beta_7 \\
\text{where,} \quad x_i &= di + d_{i+1} + \ldots + d_7 \\
d_i &= (1 - 2C_i)
\end{aligned}$$

adding ${t_i}^u$ and t_i^l

$$\begin{aligned}
s_i &= t_i^u + t_i^l \\
&= (1-c_i)\left(t_{i-1}^u + t_{i-1}^l + p_i + q_i\right) + C_i\left(t_{i-1}^u + t_{i-1}^l + s_i + r_i\right) \\
&= (1-c_i)\left(s_{i-1} + p_i + q_i\right) + c_i\left(s_{i-1} + s_i + r_i\right) \\
&= s_{i-1} + (1-c_i)\left(p_i + q_i\right) + c_i\left(s_i + r_i\right) \\
&= s_{i-1} + c_i\left(s_i + r_i - p_i - q_i\right) + p_i + q_i
\end{aligned}$$

Solving this will give

$$s_7 = \omega_0' \cdot x_0' + \omega_1' \cdot x_1' + \ldots + \omega_7' \cdot x_7' + \gamma_7$$

where,

$$\begin{aligned}
x_i' &= c_i \\
\omega_i' &= s_i + r_i - p_i - q_i \\
\gamma_i &= \sum \left(p_i + q_i\right)
\end{aligned}$$

Now, we have solved both sum and difference for upper and lower time in an arbiter PUF, so now

$$t_7^u = \left(\Delta_7 + S_7\right)/2$$

$${t_7}^u = \frac{\omega_0 \cdot x_0}{2} + \frac{\omega_1 \cdot x_1}{2} + \cdots + \frac{\omega_7 \cdot x_7}{2} + \frac{\omega_0' \cdot x_0'}{2} + \cdots + \frac{\omega_7' \cdot x_7'}{2} + \frac{\beta_7 + \gamma_7}{2}$$

Simplyfing,

$$\begin{aligned}
t_7^u &= \omega_0 \cdot x_0 + \omega_1 \cdot x_1 + \cdots + \omega_7 \cdot x_7 + \omega_0' \cdot x_0' + \ldots + \omega_7' \cdot x_7' + \beta_7' \\
&= \omega^\top x + b
\end{aligned}$$

Where,

$$x_i = d_i + d_{i-1} + \ldots + d_7$$
$$d_i = (1 - 2c_i)$$
$$x'_i = c_i$$

Similarly, we can find

$$t^l_7 = (s_7 - \Delta_7)/2$$
$$\text{or, } t^l_7 = \omega'_0 \cdot x'_0 + \omega'_1 \cdot x'_1 + \cdots + \omega'_7 \cdot x'_7 - \omega_0 \cdot x_0 - \omega_1 x_1 - \ldots - \omega_7 \cdot x_7 + \beta''$$
$$= w^\top x' + b'$$

where,

$$x_i = d_i + d_{i+1} + \ldots + d_7$$
$$d_i = (1 - 2c_i)$$
$$x'_i = c_i$$

From above derivation, it is clear that a linear model can Predict the time it takes for the upper signal to reach the finish line for a simple arbiter PUF & similarly for the lower signal.

Now, we have to show that a linear model can predict response 0 and a different linear model can predict response 1.

Let,

$$A_i = p_i - a_i$$
$$B_i = s_i - d_i$$
$$Q_i = q_i - b_i$$
$$P_i = r_i - e_i$$
$$T^u_i = \left(t^u_{i-1} + p_i\right)(1 - C_i) + C_i\left(t^l_{i-1} + s_i\right)$$
$$T^l_i = \left(t^l_{i-1} + q_i\right)(1 - C_i) + C_i\left(t^u_{i-1} + r_i\right)$$
$$K^u_i = \left(K^u_{i-1} + a_i\right)(1 - C_i) + C_i\left(K^l_{i-1} + d_i\right)$$
$$K^l_i = \left(K^l_{i-1} + b_i\right)(1 - C_i) + C_i\left(K^u_{i-1} + e_i\right)$$

$$\Delta^u_i = (T^u_i - K^u_i) = \left(T^u_{i-1} - K^u_{i-1} + p_i - a_i\right)$$
$$\Delta^u_i = \left(T^u_{i-1} - K^u_{i-1} + p_i - a_i\right) * (1 - C_i) + C_i\left(T^l_{i-1} - K^l_{i-1} + s_i - d_i\right)$$
$$\Delta^u_i = \left(\Delta^u_{i-1} + p_i - a_i\right)(1 - C_i) + C_i\left(\Delta^l_{i-1} + s_i - d_i\right)$$
$$\Delta^u_i = C_i\left(\Delta^l_{i-1} - \Delta^u_{i-1}\right) + (B_i - A_i)(C_i) + \Delta^u_{i-1} + A_i$$
$$\Delta^l_i = C_i\left(\Delta^l_{i-1}\right) + \Delta^u_{i-1} + (P_i - Q_i)C_i + \Delta^l_{i-1} + Q_i$$
$$X_i = \Delta^u_i + \Delta^l_i = (B_i + P_i + Q_i - A_i)C_i + \Delta^u_{i-1} + \Delta^l_{i-1} + A_i + Q_i$$
$$Y_i = \Delta^u_i - \Delta^l_i = (1 - 2C_i)\left(\Delta^u_{i-1} - \Delta^l_{i-1}\right) + (B_i - A_i + Q_i - P_i)C_i + A_i - Q_i$$

Y_i is similar to arbiter PUF X :

$$\alpha_i = (A_i - B_i + P_i - Q_i)/2$$
$$\beta_i = (A_i - B_i - P_i + Q_i)/2$$
$$d_i = (1 - 2C_i)$$
$$Y_7 = \omega_0 x_0 + \omega_1 x_1 + \cdots + \omega_7 x_7 + \beta_7 = \omega^\top x + b$$
$$\text{where }, x_i = d_i \cdot d_{i+1} \cdot \ldots d_7$$
$$\omega_0 = \alpha_0$$
$$\omega_i = \alpha_i + \beta_{i-1}$$

Now for X :

$$\gamma_i = B_i - A_i + P_i - Q_i$$
$$X_7 = \omega_0' \cdot x_0' + \omega_1' \cdot x_1' + \cdots + x_7' \omega_7' + E_7$$
$$\text{where,} \quad \omega_i = \gamma_i$$
$$E_i = \sum (A_j + Q_j)$$
$$E_i = \beta_i'$$
$$x_i' = C_i$$

So, we get

$$\Delta_7^u = (X_7 + Y_7)/2$$
$$\Delta_7^l = (X_7 - Y_7)/2$$

Hence Δ_7^u represents a linear modest $\frac{1+\operatorname{sign}(\omega_1^\top x+b)}{2}$ and Δ_7^l represents a linear model $\frac{1+\operatorname{sign}(\omega_2^\top x+b)}{2}$ The output of the XOR is.

$$\frac{\left(1 + (-1)^{k+1} \pi_i \operatorname{sign}\left(\omega_i^\top x\right)\right)}{2}$$
$$= \frac{1 - \operatorname{sign}\left(\pi_i \left(\omega_i^\top x\right)\right)}{2}$$

where k is the number of PUFs

Part 2: Dimensionality Of The Linear Model

The ML-PUF takes an 8-bit challenge $c = [c_0, c_1, \ldots, c_7]$, where $c_i \in \{0, 1\}$.

We represent this challenge as variables $C_0, C_1, \ldots, C_7$. To capture non-linear behavior introduced by the XOR operations in the ML-PUF, we include a bias term (constant), denoted by 1.

Thus, we now consider a 9-variable input vector:

$$[C_0, C_1, \ldots, C_7, 1]$$

We apply a **degree-2 polynomial transformation** over these 9 variables to model pairwise interactions:

- Degree-1 terms (linear):
  $C_0, C_1, \ldots, C_7, 1 \quad \Rightarrow \quad \binom{9}{1} = 9$ terms
- Degree-2 terms (quadratic):
  $C_0C_1, C_0C_2, \ldots, C_7 \cdot 1 \quad \Rightarrow \quad \binom{9}{2} = 36$ terms

Hence, the local dimensionality $\widetilde{D}$ is:

$$\widetilde{D} = \binom{9}{1} + \binom{9}{2} = 9 + 36 = 45$$

The linear model therefore requires a 45-dimensional feature space ($\widetilde{D} = 45$) to predict the response of an ML-PUF.

Part 3: Kernel Selection For Classification Of ML-PUF Response

The ML-PUF response $r(c)$ is computed as the XOR of two linear models (Response 0, Response 1). Since XOR is non linear, it behaves like a degree -3 polynomial in the challenge bits. A Kernel-SVM must map C to a higher-dimensional space where the responses are linearly separable.

The Polynomial Kernel definition:

$$k(x,z) = \left(\gamma \cdot x^\top z + \text{coeff } 0\right)^d$$

Where,

$d = 3$: degree of the polynomial
γ : Scale factor for the dot product. A reasonable default is $\gamma = 1$
Coeff0: Constant added before exponentiation. Coeff $0 = 1$ to include bias and lower degree terms.

$$k(x,z) = \left(x^T Z + 1\right)^3 = \tilde{\varnothing}(x)^\top \tilde{\varnothing}(z)$$

This kernel implicitly spans all monomials of x and z up to degree 3 , including linear. Quadratic and cubic terms and the bias term - exactly matching the 93-dimensional feature map.

Why not other Kernels:

- **RBF Kernel:**

$$\kappa(x,z) = \exp\left(-\gamma \|x - z\|^2\right)$$

The RBF Vernal maps data to an infinite-dimensional space, which is overkill for this task. It may work but lack interpretability and risks overfitting on small datasets.

- **Matern Kernel:**

This is typically used in Gaussian Processes for continuous -valued data and is not well-suited to discrete binary inputs like challenges bits.

Part 4: Method For Recovering Non-Negative Delays

For a k-bit Arbiter PUF, We define

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

Then the linear model ω, b is given by:

$$\begin{aligned} &\omega_0 = \alpha_0 \\ &\omega_i = \alpha_i + \beta_{i-1} \quad \text{for } i = 1, 2, 3 \ldots .63 \\ &b = \beta_{63} \end{aligned}$$

Thus, the vector $\omega \in \mathbb{R}^{64}$ and scalar b are computed from:

$$\begin{cases} \omega_0 = \alpha_0 \\ \omega_1 = \alpha_1 + \beta_0 \\ \vdots \\ \omega_{63} = \alpha_{63} + \beta_{62} \\ b = \beta_{63} \end{cases}$$

Now, let $x = [p_0, q_0, r_0, S_0, \ldots.., p_{63}, q_{63}, r_{63}, s_{63}]^\top \in \mathbb{R}^{256}$

The system $Ax = b$ is

$$\begin{bmatrix} A_\alpha \\ A_\beta \end{bmatrix} x = \begin{bmatrix} w \\ b \end{bmatrix}$$

where,

$A_\alpha \in \mathbb{R}^{64\times 256}$ is a sparse matrix encodes α_i *equations*
$A_\beta \in \mathbb{R}^{1\times 256}$ encodes $\beta_{63} \; = b$

We have a linear system with **more variables (256) than equations (65) -an underdetermined system. Thus, there are infinitely many solutions.**

To get a feasible and meaningful solution:-

Option 1: Non-Negative Least Squares (NNLS)

$$\min_{x \geqslant 0} \|Ax - b\|_2^2$$

Pros: Simple, satisfies the non-negativity constraint directly.
Tools: scipy.optimize.nnls or sklearn.linear_model.LinearRegression with constraint.

Option 2: Sparse Regularized Least Squares (SRLS)

$$\min_{x \geqslant 0} \|Ax - y\|_2^2 + \lambda \|x_1\|$$

Pros: Encourages sparse delay vectors.
Tools: skearn.linear_model.Lasso with Positive Constraint.

Option 3: Recursive Back-Substitution (Custom Method)

Let us define

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

Compute α_i, β_i recursively

$$\alpha_0 = \omega_0$$

for $i \geqslant 1; \alpha_i = \omega_i - \beta_{i-1}$
$\beta_{63} = b$
Choose $a_i = r_i = 0$ then

$$\begin{gathered} p_i = \alpha_i + \beta_i, \quad s_i = \alpha_i - \beta_i \\ \text{if } p_i < 0 : \text{set } q_i = -\min(0, \alpha_i + \beta_i) + \varepsilon \\ \text{if } s_i < 0 : \text{set } r_i = -\min(0, \alpha_i - \beta_i) + \varepsilon \end{gathered}$$

Implementation Notes

- Scarcity utilization: The matrix A is sparce - using solvers like scipy. sparse linalg. lsan improves performance.
- Numerical Stability: Normalize rows of A and scale inputs to avoid ill -conditioning
- Validation: Ensure $\|A_x - y\|_2 \approx 0$ and $x \geqslant 0$.

Part 7: LinearSVC vs. LogisticRegression

Key observations.

- **Effect of C.** A very small C (0.01) under-fits, giving noticeably lower accuracy. Medium and high values (1 and 100) saturate at 100% accuracy for all models, but training time grows with C.
- **Effect of loss (LinearSVC).** `squared_hinge` is slightly slower for large C yet attains the same accuracy ceiling; for $C = 0.01$ it outperforms plain `hinge`.

Table 1: LinearSVC – influence of `loss` and C on training time and test accuracy

Loss	C	**Train time (s)**	**Test acc. (%)**
hinge	0.01	0.10	88.38
hinge	1	1.67	100.00
hinge	100	1.84	100.00
squared_hinge	0.01	0.16	93.50
squared_hinge	1	1.51	100.00
squared_hinge	100	2.94	100.00

Table 2: LogisticRegression – influence of `penalty` and C

Penalty	C	**Train time (s)**	**Test acc. (%)**
ℓ_2	0.01	0.04	81.69
ℓ_2	1	0.09	100.00
ℓ_2	100	0.07	100.00
ℓ_1	0.01	0.04	78.12
ℓ_1	1	0.92	100.00
ℓ_1	100	1.25	100.00

- **Effect of penalty (LogReg).** With $C \geq 1$, both ℓ_1 and ℓ_2 reach perfect accuracy. ℓ_1 costs more time because of the iterative feature-selection nature of the solver.

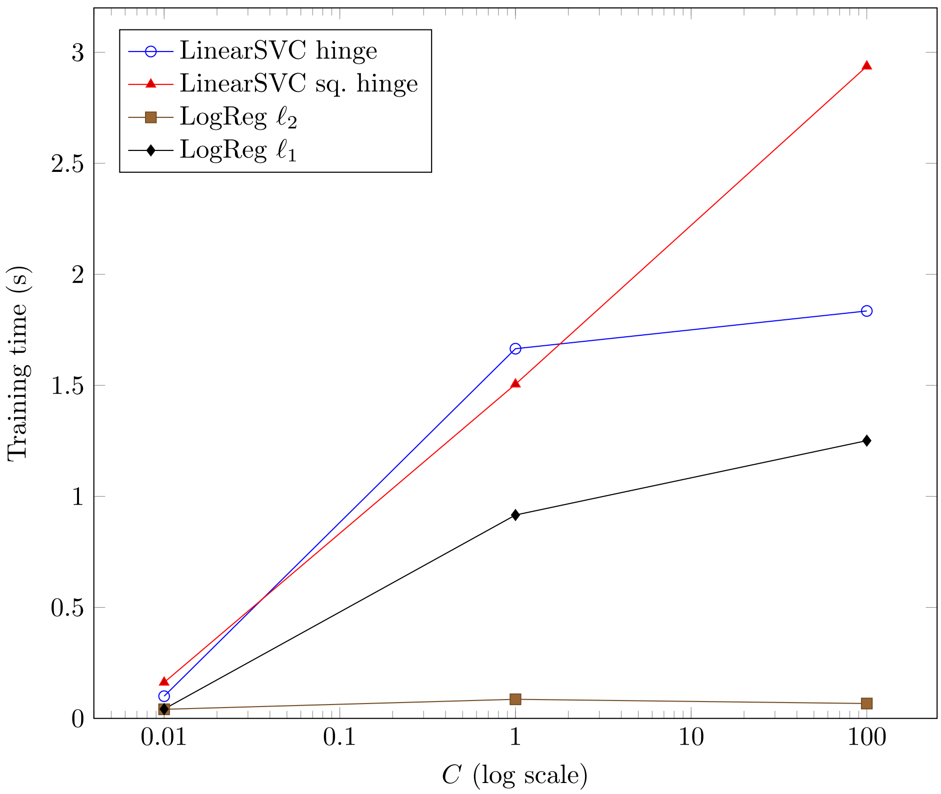

Figure 1: Training time vs. regularisation strength C for each loss/penalty setting.