CS771 Group Project

Team 405Found

Part 1

Give a detailed mathematical derivation (as given in the lecture slides) how a single linear model can predict the responses of an ML-PUF. Specifically, give an explicit map

$$\tilde{\phi} : \{0,1\}^8 \to \mathbb{R}^{\tilde{D}}$$

and a corresponding linear model

$$\tilde{\mathbf{W}} \in \mathbb{R}^{\tilde{D}}, \quad \tilde{b} \in \mathbb{R}$$

that predicts the responses, i.e., for all CRPs $\mathbf{c} \in \{0,1\}^8$, we have

$$\frac{1 + \text{sign}(\tilde{\mathbf{W}}^\top \tilde{\phi}(\mathbf{c}) + \tilde{b})}{2} = r(\mathbf{c})$$

where $r(\mathbf{c})$ is the response of the ML-PUF on the challenge $\mathbf{c}$. Note that $\tilde{\mathbf{W}}, \tilde{b}$ may depend on the PUF-specific constants such as delays in the multiplexers. However, the map $\tilde{\phi}(\mathbf{c})$ must depend only on $\mathbf{c}$ (and perhaps universal constants such as 2, $\sqrt{2}$, etc). The map $\tilde{\phi}$ must not use PUF-specific constants such as delays.

Solution

The delay outputs of each stage are given by:

$$\begin{aligned}
t_i^u &= (1 - C_i)(t_{i-1}^u + p_i) + C_i(t_{i-1}^l + s_i), && (1)\\
t_i^l &= (1 - C_i)(t_{i-1}^l + q_i) + C_i(t_{i-1}^u + r_i), && (2)\\
T_i^u &= (1 - C_i)(T_{i-1}^u + p_i') + C_i(T_{i-1}^l + s_i'), && (3)\\
T_i^l &= (1 - C_i)(T_{i-1}^l + q_i') + C_i(T_{i-1}^u + r_i'). && (4)
\end{aligned}$$

In the above equations:

- t_i's represent the time delays of PUF 1
- T_i's represent the time delays of PUF 0
- $p_i,\ q_i,\ r_i,\ s_i$ represent the delay parameters of PUF 1
- $p_i',\ q_i',\ r_i',\ s_i'$ represent the delay parameters of PUF 0

We define the difference variables:

$$\Delta_i = t_i^u - T_i^u, \qquad \delta_i = t_i^l - T_i^l.$$

Here, Δ_i governs response 1, and δ_i governs response 0.

$$\begin{aligned}
\Delta_1 &= (1 - C_1)(\Delta_0 + p_1 - p_1') + C_1(\delta_0 + s_1 - s_1'), && (5)\\
\delta_1 &= (1 - C_1)(\delta_0 + q_1 - q_1') + C_1(\Delta_0 + r_1 - r_1'). && (6)
\end{aligned}$$

$$\begin{aligned}
\Delta_i &= (1 - C_i)(\Delta_{i-1} + p_i - p_i') + C_i(\delta_i + s_i - s_i'), && (7)\\
\delta_i &= (1 - C_i)(\delta_{i-1} + q_i - q_i') + C_i(\Delta_{i-1} + r_i - r_i'). && (8)
\end{aligned}$$

Modelling sum and difference

$$D_i = \Delta_i - \delta_i, \quad S_i = \Delta_i + \delta_i. \tag{16}$$

$$D_i = d_i D_{i-1} + d_i(\alpha_i - \beta_i) + (A_i - B_i), \tag{17}$$
$$S_i = d_i(\alpha_i + \beta_i) + (A_i + B_i) + S_{i-1}. \tag{18}$$

$$S_i = \sum_{k=0}^{i} d_k(\alpha_k + \beta_k) + B'. \tag{19}$$

$$2\,\Delta_i = D_i + S_i, \quad 2\,\delta_i = S_i - D_i. \tag{20}$$

$$S_7 = W^T X + b = \sum_{i=0}^{7} d_i(\alpha_i + \beta_i) + b. \tag{21}$$

$$X = [d_0, d_1, \ldots, d_7]^T. \tag{22}$$

$$D_0 = d_0\alpha_0' + b_0, \tag{23}$$
$$D_1 = d_1 D_0 + d_1\alpha_1' + b_1 = \alpha_0' d_1 d_0 + (\alpha_1' + b_0)d_1 + b_1, \tag{24}$$
$$D_7 = (W')^T Y + b'. \tag{25}$$

$$Y = [\prod_{i=0}^{7} d_i, \prod_{i=1}^{7} d_i, \ldots, d_6 d_7, d_7]^T,$$
$$w_0 = \alpha_0', \quad w_i = \alpha_i' + b_{i-1}, \quad b' = b_7.$$

Final Formulation

Final response is calculated as XOR of response 1 and 0.
We know that XOR of two responses is given as:

$$\mathrm{bit}_i = \frac{1 + \mathrm{sgn}(\Delta_i\delta_i)}{2} \tag{26}$$

$$\Delta_7\delta_7 = (W^T X + b + (W')^T Y + b')(W^T X + b - (W')^T Y + b'). \tag{27}$$

$$= [W^T X + b]^2 - [(W')^T Y + b']^2. \tag{28}$$

$$= W^T Q + b_1. \tag{29}$$

$$= [W^T X]^2 + 2b(W^T X) + b^2 - [(W')^T Y]^2 - 2b(W')^T Y - b'^2. \tag{30}$$

$$X^T X = \sum_{i \le j} x_i x_j, \tag{31}$$
$$Y^T Y = \sum_{i \le j} y_i y_j. \tag{32}$$

$$\phi(X,Y) = \{X, X^T X, Y, Y^T Y\}. \tag{33}$$

$$X = [d_0, d_1, \ldots, d_7]^T, \tag{34}$$

$$Y = [\prod_{i=0}^{7} d_i, \prod_{i=1}^{7} d_i, \ldots, d_6 d_7, d_7]^T. \tag{35}$$

Part 2

What dimensionality $\tilde{D}$ does the linear model need to have to predict the response for an ML-PUF?
Give calculations showing how you arrived at that dimensionality. The dimensionality should be stated clearly and separately in your report, and not be implicit or hidden away in some calculations.

Answer:
The required dimensionality $\tilde{D}$ of the feature map for a linear model to predict the response for an ML-PUF:

$$\boxed{\tilde{D} = 72}$$

Calculations:

We define the feature map:

$$\Phi(x, y) = \{x,\ x^T x,\ y,\ y^T y\}$$

Let:

$$x = [d_0, d_1, \ldots, d_7]^T \quad \text{(8 elements)}$$

$$y = \left[\prod_{i=0}^{7} d_i,\quad \prod_{i=1}^{7} d_i,\quad \prod_{i=2}^{7} d_i,\quad \ldots,\quad d_6, d_7\right]^T \quad \text{(8 elements)}$$

Now computing the total dimensionality of $\Phi(x, y)$:

- 8 terms from x
- $8 \times 8 = 64$ elements in $x^T x$, but since it's symmetric, only $\binom{8}{2} = 28$ unique terms are needed
- 8 terms from y
- 28 terms from $y^T y$

$$\Rightarrow \tilde{D} = 8 + 28 + 8 + 28 = \boxed{72}$$

Part 3

Problem: Suppose we wish to use a kernel SVM to solve the problem instead of creating our own feature map. Thus, we wish to use the original challenges $c \in \{0,1\}^8$ as input to a kernel SVM (i.e., without converting challenge bits to $+1, -1$ or taking cumulative products). What kernel should we use so that we get perfect classification? Justify your answer with calculations and give suggestions for kernel type (RBF, poly, Matern etc.) and kernel parameters (gamma, degree, coef0, etc).

Solution:

Step 1: Understanding the Target Feature Space

From earlier parts of the assignment, we created a 72-dimensional feature vector $\phi(c)$ from the 8-bit challenge c. The feature vector included:

- 8 original bits: $d_0, d_1, \ldots, d_7$
- 28 second-order monomials: $d_i d_j$
- 8 cumulative products: $y_i = \prod_{j=i}^{7} d_j$
- 28 second-order products: $y_i y_j$

The highest degree term observed is $y_0^2 = \left(\prod_{j=0}^{7} d_j\right)^2$, which is a monomial of degree 16. However, due to the binary nature of the inputs and redundancy in the cumulative products, most useful features are represented using monomials up to degree 4. Hence, we target a kernel that captures monomials up to degree 4.

Step 2: Choosing the Kernel

We consider the **polynomial kernel**:

$$K(x,z) = (\gamma \cdot x^\top z + \text{coef0})^d$$

This kernel implicitly computes all monomials up to degree d over the input features.

To reproduce the 72-dimensional feature map used previously, including all relevant cumulative and quadratic interaction terms, it suffices to use a kernel that includes monomials up to **degree 4**.

Why not RBF or Matern? While RBF and Matern kernels are universal approximators, they do not align naturally with the structure of our handcrafted polynomial features. A polynomial kernel provides a simpler and more interpretable solution in this case.

Step 3: Setting Kernel Parameters

- **Degree** $d = 4$: Includes all monomials up to degree 4, capturing the necessary interaction terms in the challenge bits.
- **Gamma** $\gamma = 1$: Since the inputs $c \in \{0,1\}^8$, $c^\top z \in [0,8]$. A gamma of 1 keeps the kernel value in a reasonable scale and maintains numerical stability.

- **coef0 = 1**: Ensures that constant and lower-degree terms (linear, quadratic, cubic) are included in the expansion. This yields:

$$(x^\top z + 1)^4 = \sum_{k=0}^{4} \binom{4}{k} (x^\top z)^k$$

Final Kernel Recommendation

$$K(c_i, c_j) = (c_i^\top c_j + 1)^4$$

where:

- Kernel type: Polynomial
- Degree: 4
- Gamma: 1
- Coef0: 1

Conclusion: This kernel captures the required feature interactions from the original challenges and is capable of achieving perfect classification for the PUF model.

Part 4

Outline a method which can take a 64 + 1-dimensional linear model corresponding to a simple arbiter PUF (unrelated to the ML-PUF in the above parts) and produce 256 non negative delays that generate the same linear model. This method should show how the model generation process of taking 256 delays and converting them to a 64+1-dimensional linear model can be represented as a system of 65 linear equations and then showing how to invert this system to recover 256 non-negative delays that generate the same linear model. This could be done, for example, by posing it as an (constrained) optimization problem or other ways– see hints below

Solution

We are given a $(64+1)$-dimensional linear model corresponding to a simple arbiter PUF. The goal is to recover 256 non-negative delays that reproduce the same linear model. The linear model weights are defined as:

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

For $i = 0$ to $n-2$, the parameters are defined as:

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

Assume:

$$\beta_{i-1} = 0, \quad q_i = 0, \quad s_i = 0 \Rightarrow \beta_i = 0 \Rightarrow p_i = r_i$$

Substituting back:

$$\alpha_i = \frac{p_i - 0 + p_i - 0}{2} = p_i$$

Hence, for $i \neq n-1$,

$$\boxed{\alpha_i = w_i = p_i = c_i}$$

For $i = n-1$, we generalize the relation:

$$\alpha_{n-1} = \frac{p_{n-1} + r_{n-1}}{2}, \quad b = \frac{p_{n-1} - r_{n-1}}{2}$$

Solving:

$$p_{n-1} = \alpha_{n-1} + b = w_{n-1} + b, \quad r_{n-1} = \alpha_{n-1} - b = w_{n-1} - b$$

To ensure non-negativity of the delay vectors $\mathbf{p}$ and $\mathbf{r}$, we offset the delays:

$$\begin{aligned} a &= \max(0, \ -\min(p_i)) \\ q &= a \\ p &= p + a \end{aligned}$$

$$\begin{aligned} b &= \max(0, \ -\min(r_i)) \\ s &= b \\ r &= r + b \end{aligned}$$

This ensures:

$$p_i, r_i \geq 0, \quad \text{and the linear model } w \text{ remains unchanged.}$$

Part 7

Report outcomes of experiments with both the sklearn.svm.LinearSVC and sklearn.linear model.LogisticRegression methods when used to learn the linear model for problem 1.1 (breaking the ML-PUF). In particular, report how various hyperparameters affected training time and test accuracy using tables and/or charts. Report these experi ments with both LinearSVC and LogisticRegression methods even if your own submission uses just one of these methods or some totally different linear model learning method

Solution

Experiment Results Summary

(a) Changing Loss Parameter in LinearSVC

Loss Function	Training Time	Test Accuracy
Squared hinge	0.24214874520021112	0.9868749999999998
Hinge	0.995615072400032	0.9962500000000001

(b) Changing C in LinearSVC

C Value	Training Time	Test Accuracy
0.1 (low)	0.18884108040001593	0.9512500000000002
1 (medium)	0.64553845160025958	0.9868749999999998
100 (high)	4.568650690399954	1.0000000000000000

(c) Changing C in Logistic Regression

C Value	Training Time	Test Accuracy
0.1 (low)	0.7560820826003691	0.9193750000000002
1 (medium)	1.106241112600037	0.9748750000000002
100 (high)	2.4295156336002037	1.0000000000000

(d) Changing Tolerance in Logistic Regression

Tolerance	Training Time	Test Accuracy
10^{-2} (high)	0.38940004979995135	0.9193750000000002
10^{-4} (medium)	0.670714160600437	0.9193750000000002
10^{-6} (low)	1.341018523599996	1.000000000000000

(e) Changing Tolerance in LinearSVC

Tolerance	Training Time	Test Accuracy
10^{-2} (high)	0.262991986400084	0.9512500000000002
10^{-4} (medium)	0.2854793641998185	0.9868749999999998
10^{-6} (low)	0.32018029679984467	1.0000000000000

Conclusion

Both `LinearSVC` and `LogisticRegression` effectively model ML-PUF behavior, achieving perfect accuracy with proper hyperparameter tuning. Increasing the regularization parameter `C` and decreasing tolerance improves accuracy, though at the cost of longer training times. Overall, `LinearSVC` offers slightly faster training with comparable performance, making it a more efficient choice in practice.

```
# -*- coding: utf-8 -*-
"""submit.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/12GFyFcgKzX7zVBjrQqNr43fM1ZejCjA_
"""

import numpy as np
import sklearn
from scipy.linalg import khatri_rao
from sklearn.svm import LinearSVC

# You are allowed to import any submodules of sklearn that learn linear models e.g. sklearn.s
# You are not allowed to use other libraries such as keras, tensorflow etc
# You are not allowed to use any scipy routine other than khatri_rao

# 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 CHANGE THE NAME OF THE METHODS my_fit, my_map, my_decode etc BELOW
# THESE WILL BE INVOKED BY THE EVALUATION SCRIPT. CHANGING THESE NAMES WILL CAUSE EVALUATION

# 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(X_train, y_train):
################################
#  Non Editable Region Ending  #
################################

    # Use this method to train your models using training CRPs
    # X_train has 8 columns containing the challenge bits
    # y_train contains the values for responses

    # THE RETURNED MODEL SHOULD BE ONE VECTOR AND ONE BIAS TERM
    # If you do not wish to use a bias term, set it to 0
    Z = my_map(X_train)  # (N, D■)
    y = np.asarray(y_train)
    clf = LinearSVC(penalty='l2', C=1, loss='squared_hinge', tol=1e-2, max_iter=1000)
    clf.fit(Z, y)
    W = clf.coef_.flatten()  # shape: (D■,)
    b = clf.intercept_[0]  # scalar

    return W, b  # Changed w to W to match variable definition


################################
# Non Editable Region Starting #
################################
def my_map(X):
################################
#  Non Editable Region Ending  #
################################

    # Use this method to create features.
    # It is likely that my_fit will internally call my_map to create features for train point
    X = np.asarray(X)
    X_bin = 1 - 2 * X  # map {0,1} → {+1,-1}
    N, k = X.shape
```

```python
    # Compute phi1: cumulative product from end to start
    phi1 = np.cumprod(X_bin[:, ::-1], axis=1)[:, ::-1]  # shape: (N, k)
    # Precompute upper triangle indices once
    i, j = np.triu_indices(8, k=1)
    # Compare pairwise products for phi1 and X
    pairwise_phi1 = phi1[:, i] * phi1[:, j]  # shape: (N, 28)
    pairwise_X = X[:, i] * X[:, j]
    # Stack all features together
    feat = np.concatenate([phi1, pairwise_phi1, X, pairwise_X], axis=1)

    return feat


################################
# Non Editable Region Starting #
################################
def my_decode(w):
################################
#  Non Editable Region Ending  #
################################

    # Use this method to invert a PUF linear model to get back delays
    # w is a single 65-dim vector (last dimension being the bias term)
    # The output should be four 64-dimensional vectors

    # Extract w0 to w64
    w = np.array(w)
    assert len(w) == 65
    # Recover raw p and r (assume p = r = wi)
    raw_p = np.zeros(64)
    raw_r = np.zeros(64)
    raw_p[0] = w[0]
    raw_r[0] = w[0]

    for i in range(1, 64):
        # From wi = alpha_i + beta_{i-1} = (p_i - r_i)/2 + (p_{i-1} - r_{i-1})/2
        # Assuming p_i = r_i
        raw_p[i] = w[i]
        raw_r[i] = w[i]
    b = w[64]
    raw_p[63] += b
    raw_r[63] -= b
    # Ensure non-negative delays
    a = max(0, -np.min(raw_p))
    l = max(0, -np.min(raw_r))
    p = raw_p + a
    r = raw_r + l
    # Construct q and s
    q = np.full(64, a)
    s = np.full(64, l)

    return p, q, r, s
```