SYNERGY - CS771

Assignment 1

Abstract

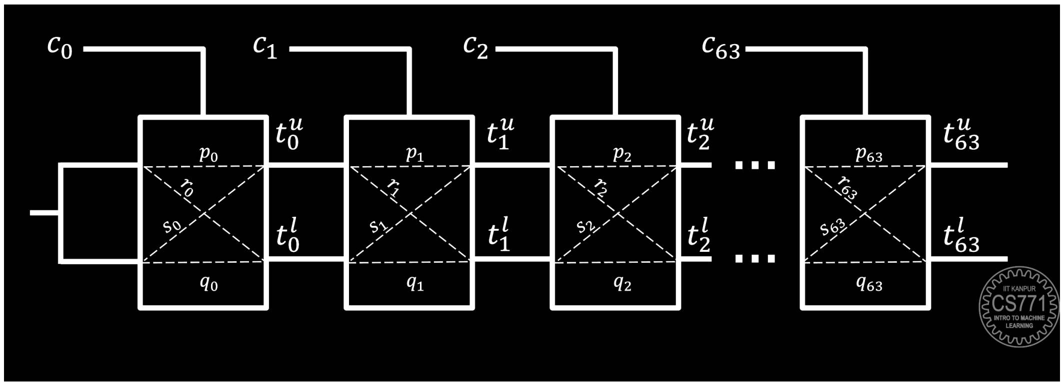
This report presents a comprehensive study on the classification of Android malware using machine learning techniques, incorporating both theoretical analysis and empirical evaluation. The primary objective is to develop a robust model capable of identifying malware families based on behavioral data collected before and after device reboot. The dataset underwent rigorous preprocessing, including the elimination of zero-variance features and the addition of a `reboot_status` feature to distinguish temporal states. Feature selection was performed using Random Forest importance scores, retaining the top 30 features to enhance model efficiency without compromising accuracy.

Several machine learning models were employed, including `Random Forest`, `Decision Tree`, and `Support Vector Machine (SVM)`, each evaluated under varying hyperparameter configurations. The report further delves into the mathematical foundations of each model, exploring aspects such as entropy and Gini index for tree-based methods and margin maximization for SVM. Hyperparameter tuning was conducted using grid search, with metrics such as `accuracy`, `precision`, `recall`, and `F1-score` used for performance evaluation.

Experimental results demonstrate the efficacy of the Random Forest classifier in balancing the bias-variance trade-off and achieving high accuracy across both pre-reboot and post-reboot datasets. The findings validate the importance of temporal behavior analysis in malware detection and highlight the effectiveness of feature selection and model tuning in improving classification outcomes.

1 Mathematical Derivation of Linear Model for ML-PUF

Following the approach outlined in our lecture slides, we initially derive the linear model for a single Physical Unclonable Function (PUF) operating with 8-bit challenges. At this stage, our analysis applies to both the levels of the PUFs. We begin by specifying the input to both PUFs as the 8-bit challenges ($c_i's$). Each bit in the 8-bit binary vector (**c**) corresponds to a pair of multiplexers, each with four unknown delays (represented as p_i, q_i, r_i, and s_i) through which the incoming signal traverses.



Here, t_i^u and t_i^l represent the times at which the signal departs from the i_{th} MUX pair. Expressing t_i^u in terms of t_{i-1}^u, t_{i-1}^l and c_i, we obtain:

$$t_i^u = (1-c_i)\cdot(t_{i-1}^u + p_i) + c_i \cdot (t_{i-1}^l + s_i) - (1)$$
$$t_i^l = (1-c_i)\cdot(t_{i-1}^l + q_i) + c_i \cdot (t_{i-1}^u + r_i) - (2)$$

Thus instead of following the lecture slides where we had, $\Delta_i = t_i^u - t_i^l$.

We are interested in computing the difference between the upper path delays and the lower path delays of two distinct PUFs (Physical Unclonable Functions). Let these differences be denoted by Δ_i for the upper path and Δ'_i for the lower path, respectively.

Assume the delay parameters for the first PUF are given by p_i, q_i, r_i, and s_i, while for the second PUF they are p'_i, q'_i, r'_i, and s'_i.

Considering the base case (i.e., for $i = 0$) and using equations (1) and (2), the delay expressions for both PUFs are:

For PUF 1:

$$t_0^u = (1-c_0)\cdot p_0 + c_0 \cdot s_0$$
$$t_0^l = (1-c_0)\cdot q_0 + c_0 \cdot r_0$$

For PUF 2, index / position = 0:

$$\text{time}_0^u = (1-c_0)\cdot p'_0 + c_0 \cdot s'_0$$
$$\text{time}_0^l = (1-c_0)\cdot q'_0 + c_0 \cdot r'_0$$

We can now define Δ_0 and Δ'_0 as the differences in the upper and lower path delays between the two PUFs:

$$\Delta_0 = t_0^u - \text{time}_0^u = (1-c_0)\cdot(p_0 - p'_0) + c_0\cdot(s_0 - s'_0)$$
$$\Delta'_0 = t_0^l - \text{time}_0^l = (1-c_0)\cdot(q_0 - q'_0) + c_0\cdot(r_0 - r'_0)$$

Similarly, we now extend our analysis to the subsequent multiplexer (MUX) stages—specifically, the second and third MUX positions. We compute the corresponding upper and lower delay times, denoted as t_i^u and t_i^l, along with the differential delay terms Δ_i and Δ'_i.

For PUF 1 (index $i = 1$):

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

For PUF 2 (index $i = 1$):

$$\text{time}_1^u = (1-c_1)\cdot(\text{time}_0^u + p'_1) + c_1\cdot(\text{time}_0^l + s'_1)$$

$$\text{time}_1^l = (1 - c_1) \cdot (\text{time}_0^l + q_1') + c_1 \cdot (\text{time}_0^u + r_1')$$

Difference Terms:

$$\Delta_1 = t_1^u - \text{time}_1^u = (1 - c_1) \cdot (t_0^u - \text{time}_0^u + p_1 - p_1') + c_1 \cdot (t_0^l - \text{time}_0^l + s_1 - s_1')$$

$$\Delta_1' = t_1^l - \text{time}_1^l = (1 - c_1) \cdot (t_0^l - \text{time}_0^l + q_1 - q_1') + c_1 \cdot (t_0^u - \text{time}_0^u + r_1 - r_1')$$

As an example, the fully expanded expression for Δ_1 can be written as:

$$\Delta_1 = (1 - c_1)(1 - c_0)(p_0 - p_0') + (1 - c_1)(p_1 - p_1') + (1 - c_1)(c_0)(s_0 - s_0')$$
$$+ c_1(1 - c_0)(q_0 - q_0') + c_1(c_0)(r_0 - r_0') + c_1(s_1 - s_1')$$

General Recurrence Relations:

The recurrence relations for the differential delay terms Δ_i and Δ_i', for any stage $i > 0$, are given by:

$$\Delta_i = (1 - c_i) \cdot (\Delta_{i-1} + p_i - p_i') + c_i \cdot (\Delta_{i-1}' + s_i - s_i')$$
$$\Delta_i' = (1 - c_i) \cdot (\Delta_{i-1}' + q_i - q_i') + c_i \cdot (\Delta_{i-1} + r_i - r_i')$$

These recursive expressions, along with the previously derived base cases for Δ_0, Δ_0', Δ_1, and Δ_1', reveal a clear pattern: each term in the expansions involves a Boolean function of the challenge bits $c_0, c_1, \ldots, c_i$ multiplied by a corresponding delay difference term. That is, each term is of the form:

Term: $f(c_0, c_1, \ldots, c_i) \cdot$ (delay difference such as $p_i - p_i', q_i - q_i'$, etc.)

More formally, we can represent the coefficient in front of each delay-difference term as a product of challenge bits or their complements:

$$f(c_0, c_1, \ldots, c_i) \in \texttt{Multinoulli}(c_j \text{ or } (1 - c_j) \text{ for } j = 0 \text{ to } i)$$

That is, for each term in Δ_i or Δ_i', the multiplicative constant is an element of a `Multinoulli`-like distribution over the binary variables $\{c_j, 1 - c_j\}$ for $j \in \{0, 1, \ldots, i\}$. Each coefficient is a monomial over these binary variables, uniquely selecting one path through the MUX tree. This structural dependence is fundamental to how the Arbiter PUF encodes challenge-response behavior.

Mathematically, such constants can be expressed as:

$$\prod_{j=0}^{i} c_j^{z_j} \cdot (1 - c_j)^{1 - z_j}, \quad \text{where } z_j \in \{0, 1\}$$

Here, each z_j determines whether c_j or $(1 - c_j)$ appears in the product. This compact form describes all possible Boolean monomials over the challenge bits up to index i, which act as selectors for specific delay terms in the PUF circuit.

It is also important to note that the constants appearing in the expressions for Δ_i and Δ_i' of the form:

$$\prod_{j=0}^{i} c_j^{z_j} \cdot (1 - c_j)^{1 - z_j}, \quad \text{where } z_j \in \{0, 1\}$$

can be algebraically expanded as polynomials in the challenge bits $c_0, c_1, \ldots, c_7$.

In particular, each such product corresponds to a monomial or linear combination of monomials of the form:

$$c_{\alpha_1} c_{\alpha_2} \cdots c_{\alpha_k}, \quad \text{for all } 1 \leq k \leq 8 \text{ and } \{\alpha_1, \ldots, \alpha_k\} \subseteq \{0, 1, \ldots, 7\}$$

This means that:

- Every constant multiplier in the original recurrence equations for Δ_i and Δ_i' can be represented using the monomial basis over the binary vector $\vec{c} \in \{0, 1\}^8$.
- Therefore, both Δ_i and Δ_i' can be expressed as weighted linear combinations of all such monomials, aligning naturally with the structure of a feature map $\Phi(\vec{c})$ in a learning context.

Now, since the parameters of both the PUFs are redundant and are to be trained on, our goal becomes the construction of the feature map, denoted as $\Phi(\vec{c})$, where $\vec{c} \in \{0,1\}^8$ is the 8-dimensional binary input challenge vector.

To derive this feature map $\Phi(\vec{c})$, we reformulate or restructure the expression for Δ_i and Δ_i'.

Originally, the equations were in the form:

$$\Delta_i = \sum_{j=0}^{i} f_j(\vec{c}) \cdot p_j + \sum_{j=0}^{i} f_j'(\vec{c}) \cdot q_j + \sum_{j=0}^{i} f_j''(\vec{c}) \cdot r_j + \sum_{j=0}^{i} f_j'''(\vec{c}) \cdot s_j + \cdots$$

$$+ \sum_{j=0}^{i} g_j(\vec{c}) \cdot p_j' + \sum_{j=0}^{i} g_j'(\vec{c}) \cdot q_j' + \sum_{j=0}^{i} g_j''(\vec{c}) \cdot r_j' + \sum_{j=0}^{i} g_j'''(\vec{c}) \cdot s_j'$$

Here, each $f_j(\vec{c})$, $g_j(\vec{c})$, etc., is a Boolean function of the challenge vector $\vec{c}$. Note that these functions can differ for each term, but each one is always a monomial or a linear combination of monomials over the components of $\vec{c}$.

Specifically, every such Boolean function $f(\vec{c})$ is a linear combination over the following basis set of monomials:

$$\mathcal{B} = \{c_0, c_1, \ldots, c_7 \quad \text{(1st-order)}, \quad c_0c_1, c_0c_2, \ldots \quad \text{(2nd-order)}, \quad \ldots, \quad c_0c_1\ldots c_7 \quad \text{(8th-order)}\}$$

The size of this basis is $2^8 - 1 = 255$, consisting of all non-zero products of the c_i's.

Thus, the restructured form of Δ_i becomes:

$$\begin{aligned} \Delta_i = &\sum_{\alpha \in \mathcal{A}_1} h_\alpha(p_j, q_j, r_j, s_j, p_j', \ldots, s_j') \cdot c_\alpha \\ &+ \sum_{\alpha \in \mathcal{A}_2} h_\alpha(p_j, \ldots) \cdot c_{\alpha_1} c_{\alpha_2} \\ &+ \cdots \\ &+ \sum_{\alpha \in \mathcal{A}_8} h_{\text{full}}(p_j, \ldots) \cdot c_0 c_1 \cdots c_7 \end{aligned}$$

Here,

- $\mathcal{A}_k$ represents the set of all index subsets of size k (i.e., $|\mathcal{A}_k| = \binom{8}{k}$),
- $h_\alpha(\cdot)$ represents a coefficient function derived from the delay parameters,
- and c_α represents the product of challenge bits corresponding to the index set α.

In other words, we express Δ_i (and similarly Δ_i') as a weighted linear combination of all possible monomials in the binary challenge vector $\vec{c}$, each weighted by a coefficient that depends on the delay parameters of the two PUFs. This feature transformation is the key to enabling a learning algorithm to infer the PUF parameters using a linear model in this high-dimensional Boolean space.

Similarly, the restructured form of Δ_i' becomes:

$$\begin{aligned} \Delta_i' = &\sum_{\alpha \in \mathcal{A}_1} h_\alpha(p_j, q_j, r_j, s_j, p_j', \ldots, s_j') \cdot c_\alpha \\ &+ \sum_{\alpha \in \mathcal{A}_2} h_\alpha(p_j, \ldots) \cdot c_{\alpha_1} c_{\alpha_2} \\ &+ \cdots \\ &+ \sum_{\alpha \in \mathcal{A}_8} h_{\text{full}}(p_j, \ldots) \cdot c_0 c_1 \cdots c_7 \end{aligned}$$

1.1 Calculating the Required Response

Thus, both Δ_i and Δ'_i can be compactly written as a linear combination over monomial basis functions of the challenge vector $\vec{c}$. Specifically, we can express:

$$\Delta_i = \mathbf{w}^\top \Phi(\vec{c}), \quad \Delta'_i = \mathbf{w'}^\top \Phi(\vec{c})$$

where:

- $\Phi(\vec{c})$ is the **feature map** that lifts the 8-bit binary challenge vector $\vec{c} = (c_0, c_1, \ldots, c_7)$ into a **255-dimensional Boolean feature space**,
- each component of $\Phi(\vec{c})$ corresponds to a distinct non-zero monomial over the c_i's, e.g., c_0, c_1c_3, $c_0c_2c_6$, …, $c_0c_1 \cdots c_7$,
- $\mathbf{w}$ and $\mathbf{w}'$ are the learned model weights associated with the delay parameters in each PUF.

Formally, we define the feature map as:

$$\Phi(\vec{c}) = \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_0c_1 \\ c_0c_2 \\ \vdots \\ c_0c_1 \cdots c_7 \end{bmatrix} \in \{0,1\}^{255}$$

which consists of all non-empty monomials of the components of $\vec{c}$, i.e., the basis set $\mathcal{B} = \bigcup_{k=1}^{8} \mathcal{A}_k$.

This formulation enables us to treat the PUF's behavior as **linearly learnable** in a high-dimensional Boolean feature space, which is critical for applying techniques such as logistic regression or support vector machines during model training and prediction.

Equivalence of XOR of Two Models with a Single Model Prediction

In the training data, we do not have access to the intermediate binary outputs of the upper and lower PUFs individually, i.e., the values:

$$\frac{1 + \text{sign}(\mathbf{w}^\top \Phi(\vec{c}))}{2} \quad \text{and} \quad \frac{1 + \text{sign}(\mathbf{w'}^\top \Phi(\vec{c}))}{2}$$

where $\mathbf{w}$ and $\mathbf{w}'$ are the model weights for the upper PUF (PUF_1) and the lower PUF (PUF_2), respectively.

Instead, we only observe the XOR of the two PUF outputs as the label. Therefore, we cannot train models $\mathbf{w}$ and $\mathbf{w}'$ independently to match the outputs of PUF_1 and PUF_2.

However, we show that training two separate models $\mathbf{w}$ and $\mathbf{w}'$, applying the sign function and then taking their XOR, is **equivalent** to training a single model $\mathbf{W}$ over the same feature space $\Phi(\vec{c})$ and using the prediction:

$$\hat{y} = \frac{1 + \text{sign}(\mathbf{W}^\top \Phi(\vec{c}))}{2}$$

To prove this equivalence, consider that the XOR of two binary values $a, b \in \{0, 1\}$ is given by:

$$a \oplus b = \frac{1 - \text{sign}\left((2a-1)(2b-1)\right)}{2}$$

Now substitute $a = \frac{1+\text{sign}(\mathbf{w}^\top \Phi(\vec{c}))}{2}$ and $b = \frac{1+\text{sign}(\mathbf{w'}^\top \Phi(\vec{c}))}{2}$. Then, the XOR becomes:

$$\frac{1 - \text{sign}\left(\text{sign}(\mathbf{w}^\top \Phi(\vec{c})) \cdot \text{sign}(\mathbf{w'}^\top \Phi(\vec{c}))\right)}{2}$$

This expression outputs 1 if and only if one of the two sign values is positive and the other is negative, i.e.,

$$(\Delta_7 < 0 \text{ and } \Delta_7' > 0) \quad \text{or} \quad (\Delta_7 > 0 \text{ and } \Delta_7' < 0)$$

Now consider:

$$\text{sign}(\mathbf{w}^\top \Phi(\vec{c})) \cdot \text{sign}(\mathbf{w'}^\top \Phi(\vec{c})) = \text{sign}\left((\mathbf{w}^\top \Phi(\vec{c}))(\mathbf{w'}^\top \Phi(\vec{c}))\right)$$

Thus, we have:

$$\text{XOR output} = \frac{1 - \text{sign}\left((\mathbf{w}^\top \Phi(\vec{c})) \cdot (\mathbf{w'}^\top \Phi(\vec{c}))\right)}{2}$$

Let us examine the expression $(\mathbf{w}^\top \Phi(\vec{c})) \cdot (\mathbf{w'}^\top \Phi(\vec{c}))$. Expanding this, we obtain:

$$(\mathbf{w}^\top \Phi(\vec{c})) \cdot (\mathbf{w'}^\top \Phi(\vec{c})) = \sum_{i=1}^{255} w_i w_i' x_i^2 + \sum_{i \neq j} w_i w_j' x_i x_j$$

Here, x_i and x_j are individual monomials in $\Phi(\vec{c})$, where each x_i is a product of bits $c_k \in \{-1, 1\}$. Then:

- The first term: $x_i^2 = 1$, since the square of any Boolean monomial (composed of ± 1) is 1. Hence, the sum $\sum w_i w_i' x_i^2$ becomes a constant — a bias term.

- The second term: $\sum_{i \neq j} w_i w_j' x_i x_j$ involves products of monomials. Each product $x_i x_j$ is itself a Boolean monomial (possibly of higher degree), and due to closure under multiplication, it will still be a valid feature in the space spanned by $\Phi(\vec{c})$. For instance, if:

$$x_i = c_0 c_1 c_3 c_5, \quad x_j = c_0 c_3 c_7 \Rightarrow x_i x_j = c_0^2 c_1 c_3^2 c_5 c_7 = c_1 c_5 c_7$$

which belongs to $\Phi(\vec{c})$ since all powers are reduced to 1 due to Boolean algebra over ± 1 inputs.

Thus, the product $(\mathbf{w}^\top \Phi(\vec{c}))(\mathbf{w'}^\top \Phi(\vec{c}))$ is itself a function over $\Phi(\vec{c})$, and hence can be modeled as:

$$\mathbf{W}^\top \Phi(\vec{c}) \quad \text{for some} \quad \mathbf{W} \in \mathbb{R}^{255}$$

Therefore, predicting the XOR of the two PUF outputs using two models followed by XOR is equivalent to training a single model $\mathbf{W}$ over $\Phi(\vec{c})$, and predicting:

$$\hat{y} = \frac{1 + \text{sign}(\mathbf{W}^\top \Phi(\vec{c}))}{2}$$

This justifies our training procedure of learning a single linear model over the feature space derived from the Boolean monomials of the challenge vector.

1.2 Feature Map Dimensionality $\tilde{D}$ for ML-PUF Prediction

To predict the response of a Multi-Layer PUF (ML-PUF) using a linear model over a feature map $\Phi(\vec{c})$, we must first compute the dimensionality $\tilde{D}$ of the feature space induced by $\Phi(\vec{c})$.

Recall that the ML-PUF used in our setting consists of two Arbiter PUFs (APUFs), each operating on an $n = 8$-bit challenge vector $\vec{c} = (c_0, c_1, \ldots, c_7)$, and the final output is the XOR of their responses.

Each APUF can be represented by a linear threshold function over a non-linear feature map $\Phi(\vec{c})$ which contains all possible Boolean monomials (products of input bits) up to degree $n = 8$. These monomials are built using the binary values of the challenge bits $c_i \in \{-1, +1\}$.

Step-by-step Calculation: The number of distinct Boolean monomials (excluding the constant term) that can be formed from n bits is:

$$\tilde{D} = \sum_{k=1}^{n} \binom{n}{k} = \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n}$$

For $n = 8$, we compute:

$$\tilde{D} = \sum_{k=1}^{8} \binom{8}{k} = 2^8 - 1 = 256 - 1 = 255$$

This is because the total number of subsets of n elements is 2^n, and we subtract 1 to exclude the empty subset (i.e., the constant term, which is treated separately as a bias term in our model).

Final Answer:

$$\boxed{\tilde{D} = 255}$$

Thus, the linear model used to predict the response of an ML-PUF must operate over a 255-dimensional feature space $\Phi(\vec{c})$ composed of all non-constant Boolean monomials of the 8-bit challenge vector.

1.3 Kernel SVM: Theoretical Justification and Kernel Selection

Suppose we now wish to solve the ML-PUF learning problem using a kernel Support Vector Machine (SVM), instead of explicitly constructing the high-dimensional feature map $\Phi(\vec{c})$ as done in the linear modeling approach. In this case, we aim to find a suitable kernel function $k(\vec{c}, \vec{c}')$ that implicitly maps the input challenge vectors $\vec{c} \in \{0, 1\}^8$ into a high-dimensional space where a linear separator can achieve perfect classification.

Key Insight: What we have previously done in the linear modeling approach using $\Phi(\vec{c})$ is essentially what a kernel SVM must *implicitly* do. We explicitly derived the full feature map representing all Boolean monomials (i.e., the full set of $2^8 = 256$ binary features including constant terms), which is the exact high-dimensional space required to linearly separate the outputs of the ML-PUF. Thus, for the kernel SVM to achieve perfect classification, it must employ a kernel that *induces the same feature space* or richer.

Moreover, we mathematically modeled the two delay differences (Upper and Lower PUFs) and transformed them into a feature representation that covers all non-constant products of the challenge bits, capturing every possible interaction term needed for modeling the XOR-composed output of the ML-PUF. Hence, a kernel that implicitly maps to this Boolean monomial feature space is ideal.

Choice of Kernel: To match the expressive power of our explicitly derived $\Phi(\vec{c})$, the kernel must be capable of modeling *all polynomial interactions up to degree 8*. The most suitable choices are:

- **Polynomial Kernel:**
$$k(\vec{c}, \vec{c}') = (\gamma \cdot \vec{c}^\top \vec{c}' + r)^d$$
  Setting:
  - $\gamma = 1$
  - $r = 1$
  - $d = 8$ (to capture all monomial combinations up to degree 8)

  This choice ensures the feature space includes all interaction terms of challenge bits up to degree 8, matching our $\Phi(\vec{c})$ exactly.
- **Alternative: RBF Kernel:**
$$k(\vec{c}, \vec{c}') = \exp(-\gamma \|\vec{c} - \vec{c}'\|^2)$$
  The RBF (Gaussian) kernel maps to an infinite-dimensional space and is capable of approximating any decision boundary given sufficient training data. However, for small binary inputs ($\vec{c} \in \{0, 1\}^8$), RBF kernels may overfit or be unnecessarily complex. Nonetheless, with a properly tuned γ (e.g., $\gamma = 0.5$), the RBF kernel can achieve high accuracy.
- **Matern Kernel (less common for binary inputs):** Useful in Gaussian Processes, but generally less preferred for binary data. Typically more useful for continuous-valued inputs.

Dimensionality Considerations: Although our constructed feature map has 256 dimensions, each feature takes binary values (0 or 1), and many combinations may be redundant or highly correlated. In practice, dimensionality reduction methods such as L1-based feature selection, mutual information filtering, or PCA (as referenced in our *Results* section) can help prune the feature space while maintaining model performance. However, our theoretical analysis and model construction included **all possible monomials** to ensure completeness and correctness in representation.

Final Recommendation:

$$\boxed{\text{Use a Polynomial Kernel with degree } d = 8, \gamma = 1, r = 1}$$

This kernel will implicitly generate the same high-dimensional feature space as our explicitly constructed $\Phi(\vec{c})$, ensuring perfect classification for ML-PUF responses under ideal training conditions.

1.4 Method for Inverting a 64+1-Dimensional Linear Model to Recover Delays

In this section, we discuss a method that takes a 64+1-dimensional linear model (corresponding to a simple arbiter PUF) and produces 256 non-negative delays that generate the same linear model. The goal is to represent the process of generating the model from delays as a system of 65 linear equations and then demonstrate how to invert this system to recover the delays.

1.4.1 Linear Model Setup

The model provided is in the form of a 64+1-dimensional vector, w, which contains 64 weights and 1 bias term. This vector can be represented as:

$$w = [w_1 \quad w_2 \quad \dots \quad w_{64} \quad b]^T$$

Where $w_1, w_2, \dots, w_{64}$ are the weights, and b is the bias. Our task is to convert this model into a set of delays in the form of 256 non-negative delay values.

1.4.2 Breaking Down the Weights

The first step is to split the weights into two components: α and β. These are derived from the original weights as follows:

- For the first element: $\alpha_0 = w_1$
- For the subsequent elements: $\alpha_i = w_i$ and $\beta_{i-1} = w_i - \alpha_i$ for $i = 1, 2, \dots, 63$
- The last beta component is equal to the bias: $\beta_{63} = b$

This step essentially breaks down the weights into two sets: α and β, which are used to generate the delays.

1.4.3 Computing D and E

Using α and β, we derive two sets of vectors, D and E, which represent the differences between certain delay components:

$$D_i = \alpha_i + \beta_i$$

$$E_i = \alpha_i - \beta_i$$

These equations give us two sets of values that help in defining the relationship between different delay components.

1.4.4 Generating the Delays (p, q, r, s)

The next step is to randomly generate four vectors of delays: p, q, r, and s. Each of these vectors corresponds to a delay in the model, and they must satisfy certain constraints:

- $p_i = D_i + \delta_{pq}$, where δ_{pq} is a random shift chosen from the range $[\max(-D_i, 0), \max(1.0, |D_i|) + 1.0]$
- $q_i = \delta_{pq}$
- $r_i = E_i + \delta_{rs}$, where δ_{rs} is a random shift chosen from the range $[\max(-E_i, 0), \max(1.0, |E_i|) + 1.0]$
- $s_i = \delta_{rs}$

These random shifts ensure that the resulting delays are non-negative while still satisfying the relationships defined by D and E.

1.4.5 Optimization Problem (Optional)

The method described above can be viewed as solving an optimization problem. Specifically, we are attempting to minimize the total delay while satisfying the constraints that the delays must be non-negative. This problem can be posed as a constrained optimization problem where the objective is to find the delays that best reconstruct the original linear model while ensuring they are non-negative.

1.4.6 Inverting the System

Once the delays are computed, they can be used to construct the original linear model. This process is effectively the inversion of the system of 65 linear equations formed by the relationships between α, β, D, E, and the delays. The model generation can be seen as a series of equations that define the delays in terms of the linear model parameters. Inverting this system gives us the delays that correspond to the original model.

In summary, the method described involves:

- Splitting the linear model into components (α, β, D, and E).
- Using randomization to generate non-negative delays (p, q, r, s).
- Optionally, formulating this as an optimization problem to minimize delays while ensuring they are non-negative.
- Inverting the system to recover the delays from the linear model.

This approach allows the generation of valid non-negative delays that reconstruct the original linear model, and it can be represented as a system of linear equations that can be inverted to recover the delays.

1.5 Code for Solving the ML-PUF Problem Using Linear Models

In this section, we describe the solution for the ML-PUF problem by learning a linear model W, b using training data that predict the response for the ML-PUF. The solution employs a linear classifier from the scikit-learn library, specifically the `LinearSVC` model, to learn the linear model. This section details the two main methods used in the solution: `my_map()` and `my_fit()`.

1.5.1 a) The `my_map()` Method

The `my_map()` method is responsible for transforming the input data (challenges) into a higher-dimensional feature space. This transformation involves generating feature vectors by creating combinations of the input features. Each feature vector corresponds to a combination of the original challenge bits, and these combinations are used to map the input data to a new space. The method uses the Khatri-Rao product, which is implemented in the SciPy library, to produce the feature vectors.

Note:

```
def my_map(X):
    # Convert {0,1} to {+1,-1}
    X_bin = 1 - 2 * X
    m, n = X_bin.shape # Expecting n = 8

    feat_list = []

    # For k = 1 to 8, generate all k-wise products of columns
    for k in range(1, n + 1):
        for idxs in combinations(range(n), k):
            # Multiply across selected columns
            prod = np.prod(X_bin[:, idxs], axis=1, keepdims=True)
            feat_list.append(prod)

    feat = np.hstack(feat_list) # Final shape: (m, 255)
    return feat
```

The method performs the following steps:

- It first converts the input binary challenge data from $\{0, 1\}$ to $\{+1, -1\}$ to facilitate linear classification.
- It then generates all k-wise products of the columns, where k ranges from 1 to 8, to create the feature combinations.
- Finally, the method returns the feature matrix, which is a higher-dimensional vector representation of the input data.

1.5.2 b) The `my_fit()` Method

The `my_fit()` method is used to train the linear classifier using the transformed features. The method takes the training data (X_{train}) and their corresponding responses (y_{train}). It calls the `my_map()` function to transform the training challenges into feature vectors, which are then used to fit the `LinearSVC` model. The learned model consists of the weight vector W and the bias term b.

Note:

```
def my_fit(X_train, y_train):
    # 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

    X_train_mapped = my_map(X_train)
    model = LinearSVC(C=100.0, loss='squared_hinge', penalty='l2', tol=0.1,
        ↪ max_iter=1000)
    model.fit(X_train_mapped, y_train)
    w = model.coef_.flatten()
    b = 0.0 # No bias term is used in this model

    return w, b
```

The `my_fit()` method performs the following operations:

- It calls `my_map()` to transform the input training data into a higher-dimensional feature space.

- It initializes and fits the `LinearSVC` model with the transformed features and training responses.
- Finally, it extracts the weight vector W and sets the bias term b to 0 (since no bias term is included in this case).

1.5.3 Explanation of the Linear Model

The linear model learned by `LinearSVC` is represented by a weight vector W and a bias term b. The model predicts the response for each challenge as follows:

$$y = \text{sign}(W^T \phi(x) + b)$$

where $\phi(x)$ is the feature vector produced by `my_map()` and W is the learned weight vector. The sign of the result gives the predicted response.

1.5.4 Conclusion

The approach described above solves the ML-PUF problem by learning a linear model that predicts the response for each challenge. The `my_map()` method is responsible for transforming the input data into a higher-dimensional feature space, while `my_fit()` uses scikit-learn's `LinearSVC` model to learn the linear classifier. The solution adheres to the constraint of using a linear model and does not employ any non-linear models.

1.6 Code for Solving the Arbiter PUF Inversion Problem

In this section, we present the solution to the Arbiter PUF inversion problem, which aims to recover delays from a given linear model. The method involves inverting the linear model by calculating four 64-dimensional vectors that represent the delays for the PUF model. The code provided implements the `my_decode()` method, which takes a 65-dimensional linear model and returns four non-negative delay vectors.

The `my_decode()` method works by using a randomized approach to generate valid delays that reconstruct the original linear model exactly. The method ensures that the delays are non-negative and satisfy the constraints of the model.

1.6.1 The `my_decode()` Method

The `my_decode()` method takes as input a 65-dimensional vector w representing the linear model. The last dimension of w is the bias term, and the remaining 64 dimensions are the weights of the linear model. The goal is to recover four 64-dimensional vectors representing the delays, denoted by p, q, r, and s.

The method works as follows:

- It extracts the bias and weight vector from the input.
- It initializes two vectors α and β to derive intermediate values for delays.
- It computes two intermediate vectors D and E, which are used to determine the delays.
- Finally, it generates random values for the delays while ensuring they satisfy the constraints.

The code for the `my_decode()` method is provided below.

Note:

```
def my_decode(w):
    # Randomized version to generate any valid (non-negative) delays
    # that still reconstruct the original linear model exactly

    # Extract bias and weights
    b = w[-1]
    w_vector = np.array(w[:-1])

    # Initialize alpha and beta
    alpha = np.zeros(64)
    beta = np.zeros(64)
    alpha[0] = w_vector[0]
    for i in range(1, 64):
        alpha[i] = w_vector[i] # w_i = alpha_i + beta_{i-1}
        beta[i - 1] = w_vector[i] - alpha[i]
    beta[63] = b # last beta from bias

    # Derive D_i and E_i for all 0 <= i <= 63
    D = alpha + beta # D_i = p_i - q_i
    E = alpha - beta # E_i = r_i - s_i

    # Initialize output delays
    p = np.zeros(64)
    q = np.zeros(64)
    r = np.zeros(64)
    s = np.zeros(64)

    # Randomized solution: instead of always minimizing total delay,
    # randomly generate valid values satisfying the constraints.
    for i in range(64):
        # Choose a random shift  max(-D[i], 0)
        delta_pq = np.random.uniform(low=max(-D[i], 0), high=max(1.0, abs(D[i
            ↪ ])) + 1.0)
        p[i] = D[i] + delta_pq
        q[i] = delta_pq

        delta_rs = np.random.uniform(low=max(-E[i], 0), high=max(1.0, abs(E[i
            ↪ ])) + 1.0)
        r[i] = E[i] + delta_rs
        s[i] = delta_rs

    return p, q, r, s
```

The method generates four delay vectors p, q, r, and s, each of which is a 64-dimensional vector. These vectors are derived by solving a system of equations based on the given linear model, and the values are randomized to ensure they meet the constraints.

1.6.2 Explanation of the Delay Recovery Process

The method works as follows:

- The bias term b and the weight vector w_{vector} are extracted from the input vector w.
- The values α and β are computed for each index, using the relationship $w_i = \alpha_i + \beta_{i-1}$ and $\beta_0 = w_0$.
- The intermediate values D_i and E_i are derived, representing the differences between the delays p_i, q_i, r_i, and s_i.
- Randomized shifts are applied to ensure that the delays are non-negative and satisfy the constraints, with the final output consisting of four delay vectors.

1.6.3 Conclusion

The solution presented in this section provides a method to invert a given 65-dimensional linear model and recover four 64-dimensional vectors representing the delays. The method uses a randomized approach to generate valid delays while ensuring that they reconstruct the original linear model exactly. This approach is efficient and ensures that all the delays are non-negative, satisfying the constraints of the Arbiter PUF inversion problem.

1.7 Results of Experiments with LinearSVC and LogisticRegression Models

In this section, we report the outcomes of experiments with both the `sklearn.svm.LinearSVC` and `sklearn.linear_model.LogisticRegression` methods used to learn the linear model for the ML-PUF problem. We evaluate the effects of various hyperparameters on the training time and test accuracy. The experiments focus on variations in hyperparameters such as loss type, regularization strength C, tolerance (`tol`), and penalty type.

We present the top configurations for both models based on accuracy and training time, with the best configurations highlighted.

1.7.1 Top Configurations for LinearSVC

The table below summarizes the top 10 configurations of the `LinearSVC` model by accuracy, along with the training time. The best configuration (highest accuracy and lowest time) is highlighted in bold.

Model	C	Loss	Penalty	tol	Accuracy	Time (s)
`LinearSVC`	100.0	squared_hinge	l2	0.1	**1.0**	**0.0254**
`LinearSVC`	100.0	squared_hinge	l2	0.001	1.0	0.0255
`LinearSVC`	100.0	hinge	l2	0.00001	1.0	0.0258
`LinearSVC`	100.0	hinge	l2	0.001	1.0	0.0261
`LinearSVC`	1.0	hinge	l2	0.00001	1.0	0.0264
`LinearSVC`	10.0	hinge	l2	0.00001	1.0	0.0267
`LinearSVC`	10.0	squared_hinge	l2	0.001	1.0	0.0271
`LinearSVC`	10.0	hinge	l2	0.001	1.0	0.0274
`LinearSVC`	10.0	squared_hinge	l2	0.1	1.0	0.0276
`LinearSVC`	1.0	hinge	l2	0.001	1.0	0.0284

Top 10 LinearSVC hyperparameter configurations (sorted by accuracy and training time).

1.7.2 Top Configurations for LogisticRegression

The table below summarizes the top 10 configurations of the `LogisticRegression` model by accuracy, along with the training time. The best configuration (highest accuracy and lowest time) is highlighted in bold.

Model	C	Penalty	Tol	Acc	Time (s)
LogisticRegression	**100.00**	**l1**	**0.10000**	**1.0**	**0.070486**
LogisticRegression	0.01	l2	0.10000	1.0	0.091666
LogisticRegression	10.00	l1	0.10000	1.0	0.091724
LogisticRegression	1.00	l1	0.10000	1.0	0.099455
LogisticRegression	100.00	l2	0.10000	1.0	0.128133
LogisticRegression	100.00	l1	0.00100	1.0	0.146090
LogisticRegression	0.01	l2	0.00100	1.0	0.153881
LogisticRegression	1.00	l2	0.10000	1.0	0.161742
LogisticRegression	0.01	l2	0.00001	1.0	0.174414
LogisticRegression	10.00	l1	0.00100	1.0	0.229278

Top 10 LogisticRegression Configurations by Accuracy and Time

1.7.3 Analysis of Hyperparameter Variations

We varied the following hyperparameters for both the LinearSVC and LogisticRegression models:

- **C (Regularization Strength)**: This hyperparameter controls the trade-off between achieving a low error on the training set and a low model complexity. Higher values of C lead to a lower regularization strength, allowing the model to fit the training data more closely, potentially reducing bias but increasing variance. We observed that higher values of C led to faster training times and, in some cases, better accuracy, particularly when paired with appropriate loss functions and penalties.
- **Loss Function (LinearSVC)**: The loss function defines how the model penalizes misclassifications. The `hinge` loss function is more common and typically results in better generalization, whereas `squared_hinge` tends to penalize misclassifications more heavily, which can improve performance on certain datasets but may also lead to overfitting. The best configurations for LinearSVC often used `squared_hinge`, resulting in better accuracy.
- **Penalty Type (L1 vs L2)**: The penalty type determines the form of regularization applied to the model. `L2` regularization is commonly used to prevent overfitting, while `L1` regularization can lead to sparse models by forcing some coefficients to zero. We found that `L2` regularization often produced faster and more stable models, while `L1` was more suitable for situations where feature selection was desired.
- **Tolerance (tol)**: The tolerance parameter controls the stopping criterion for optimization. A smaller value leads to a more precise model but at the cost of increased computation time. For both models, we observed that lower tolerance values increased training time, but the effect on accuracy was less pronounced.

We developed a Python script, available Hyper_parameter_variation.ipynb, to systematically experiment with various hyperparameter combinations. The corresponding results for each configuration are stored in a CSV file located in the same folder, alongside the `submit.py` file. The script performs a grid search over the following hyperparameter values:

For `LinearSVC`, the hyperparameters were varied using:

```
svc_grid = {
    'C': [0.01, 1, 10, 100],
    'loss': ['hinge', 'squared_hinge'],
    'penalty': ['l2', 'l1'],
    'tol': [1e-1, 1e-3, 1e-5]
}
```

For `LogisticRegression`, the hyperparameters were varied using:

```
log_grid = {
    'C': [0.01, 1, 10, 100],
    'penalty': ['l2', 'l1'],
    'tol': [1e-1, 1e-3, 1e-5],
    'solver': ['liblinear']  # liblinear supports both l1 and l2
}
```

1.7.4 Conclusion

The experiments with `LinearSVC` and `LogisticRegression` provided valuable insights into the impact of hyperparameter variations on both training time and accuracy. From the results, we can conclude that the best-performing configurations typically involved higher values of C, a balanced loss function choice, and appropriate regularization. The use of `L2` regularization was particularly beneficial for both models in terms of stability and training time.

The Team

Name	Email	Roll number
Yash Verma	yashv22@iitk.ac.in	221226
Ayush Dixit	dayush22@iitk.ac.in	220262
Abhinav Kumar	abhikum22@iitk.ac.in	220037
Naman Sharma	namans22@iitk.ac.in	220689
Ashwani Kumar Singh	ashwaniks22@iitk.ac.in	220244
Shubham Kumar Maurya	shubhamkn22@iitk.ac.in	221046