CSAW ESC 2024 Final Report

TRX Technical Labs - Sapienza

Simone Di Maria
University of Verona
simonedimaria2004@gmail.com

Kristjan Tarantelli
Sapienza University of Rome
tarantelli.kristjan@gmail.com

Francesco Bianchi
Sapienza University of Rome
f.bianchi202@gmail.com

Lorenzo Colombini
Sapienza University of Rome
lorenzinco23@gmail.com

Emilio Coppa
Luiss University of Rome
ecoppa@luiss.it

***Abstract*—This report presents our approach to hardware-based challenges centered on analyzing signal patterns and matrix transformations. Prohibited from using code disassembly, we relied on a logic analyzer to capture and interpret binary signals, uncovering QR codes, encrypted sequences, and motor signal patterns. For tasks requiring data classification, we applied machine learning models to analyze audio and vibration data with high accuracy. This integrated approach enabled us to decode complex messages and solve each phase of the challenge.**

I. Introduction

Side-Channel Attacks (SCAs) exploit vulnerabilities in hardware by analyzing physical behaviors like timing variations or signal patterns, which can reveal confidential data without targeting software flaws. These attacks may impact systems otherwise considered secure, exposing sensitive information or modifying system behavior through indirect manipulation.

This report details our methods for tackling hardware-based challenges, focusing on signal capture and pattern analysis without conventional reverse engineering. Using a logic analyzer, we extracted meaningful data from signals, applied matrix transformations, and leveraged timing analysis to decode sequences. Designed by the CSAW ESC 2024 organizers, the challenges ran on an Arduino-based system, where we explored how encrypted data and encoded sequences could be interpreted through signal-based attacks.

II. Our Approach

For this year's challenge, we adopted a hardware-focused methodology, avoiding traditional code analysis. Our primary strategy involved using a logic analyzer to capture output signals from key components, enabling us to deduce device behaviors and functionality directly.

Through careful examination of signal data, we identified the structure and operations of system components, using matrix transformations and timing patterns to progress through the tasks. This approach, centered on signal interpretation rather than code disassembly, proved effective for nearly all hardware tasks.

In analyzing the hardware setup, we mapped the connections from the Arduino to its peripherals. This modular connectivity allowed us to connect selectively to the Arduino or peripheral inputs as needed, enabling targeted testing and efficient results.

For tasks involving machine learning, specifically in Week 2, we applied algorithmic models to classify audio and vibration data. Through segmentation, feature extraction, and model training, our approach delivered accurate classification results, advancing our progress in these challenges.

III. Challenge Solutions

We attempted all the challenges, but we focused on the first ones.

TABLE I: Summary of challenge results

Week	Challenge	Status	Secret
1	Normal Or Though	Solved	Kw1CkRe5p0Nze
1	Friendly Disposition	Solved	S3qu3NC3 (n0T sEQU!ns)
2	KeyRing 1	Solved	DDAACAAACDCA...
2	KeyRing 2	Attempted	-
3	Lizzy	Solved	ZLHNKSGON - IDONTKNOW
3	Fast & Max	Solved	6203882345218489
4	Safecracker 1	Solved	131S-92S-7S
4	Safecracker 2	Solved	196I-60D-41S-20S-15I

Week 1

1) Normal Or Though: After loading the firmware, we observed no activity in the servo, stepper motor, or fan, suggesting a need for further investigation. The challenge's "quiet room" hint prompted us to connect a logic analyzer to monitor signals directed to the Arduino. Through this analysis, we discovered that the fan was receiving alternating binary signals—either all 1s or all 0s—at intervals of 0.5 seconds. Sampling these signals produced a bit sequence that we arranged into a 34×34 matrix, with a distinctive 0000 pattern recurring every 34 bits.

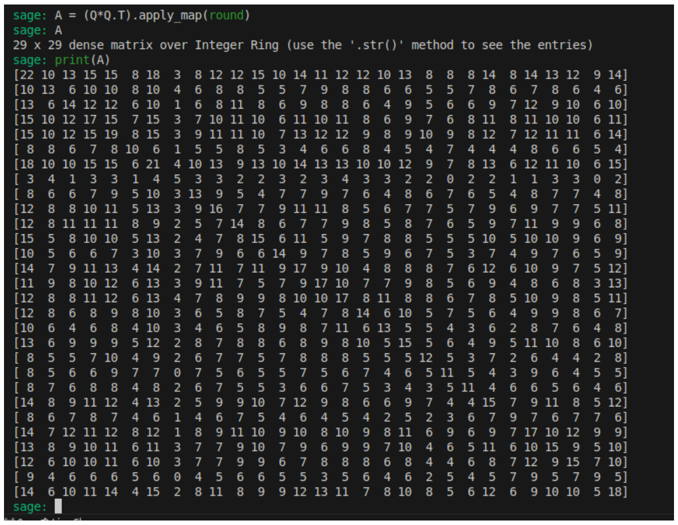
Fig. 1: First QR code extracted, which led to the challenge's gist

Visualizing the matrix revealed a QR code (see Figure 1), which led us to a GitHub gist containing the next part of the challenge. In this phase, we were given two matrices, Q and R, both defined over the real numbers. Interestingly, Q was a lower triangular matrix, with zeros in the upper right section, suggesting the potential for a Cholesky decomposition approach. Computing $Q \times Q^T$ yielded an integer-based matrix with a structure that hinted at an image—specifically, a QR code pattern when plotted in SageMath (see Figure 2).

At this stage, it became apparent that we needed to experiment with the transformations between Q and R. After several tests, we discovered that multiplying $R \times Q^T$ revealed another QR code, leading to a Google Form that contained the flag: **Kw1CkRe5p0Nze**.

2) Friendly Disposition: This challenge opened with a message on the serial console, signaling the start of Phase 1 and presenting an encoded sequence: **AABCE**. From initial observations, it appeared that this sequence might align with the Fibonacci series, as the letters could correspond to the series values: **A = 1, A = 1, B = 2, C = 3, E = 5**.

To test this theory, we examined the signal patterns sent to the stepper motor and noticed that the sequence matched ASCII uppercase letters under a modulo 26 transformation. Extending the Fibonacci series in this manner, we arrived at the next set of letters: **KJUEZEEJ**, representing positions 16 through 24 in the sequence. This guess proved correct, and we advanced to the next phase.


```
sage: A = (Q*Q.T).apply_map(round)
sage: A
29 x 29 dense matrix over Integer Ring (use the '.str()' method to see the entries)
sage: print(A)
[22 10 13 15 15  8 18  3  8 12 12 15 10 14 11 12 12 10 13  8  8  8 14  8 14 13 12  9 14]
[10 13  6 10 10  8 10  4  6  8  8  5  5  7  9  8  8  6  6  5  5  7  8  6  7  8  6  4  6]
[13  6 14 12 12  6 10  1  6  8 11  8  6  9  8  8  6  4  9  5  6  6  9  7 12  9 10  6 10]
[15 10 12 17 15  7 15  3  7 10 11 10  6 11 10 11  8  6  9  7  6  8 11  8 11 10 10  6 11]
[15 10 12 15 19  8 15  3  9 11 11 10  7 13 12 12  9  8  9 10  9  8 12  7 12 11 11  6 14]
[ 8  8  6  7  8 10  6  1  5  5  8  5  3  4  6  6  8  4  5  4  7  4  4  4  8  6  6  5  4]
[18 10 10 15 15  6 21  4 10 13  9 13 10 14 13 13 10 10 12  9  7  8 13  6 12 11 10  6 15]
[ 3  4  1  3  3  1  4  5  3  3  2  2  3  2  3  4  3  3  2  2  0  2  2  1  1  3  3  0  2]
[ 8  6  6  7  9  5 10  3 13  9  5  4  7  7  9  7  6  4  8  6  7  6  5  4  8  7  7  4  8]
[12  8  8 10 11  5 13  3  9 16  7  7  9 11 11  8  5  6  7  7  5  7  9  6  9  7  7  5 11]
[12  8 11 11 11  8  9  2  5  7 14  8  6  7  7  9  8  5  8  7  6  5  9  7 11  9  9  6  8]
[15  5  8 10 10  5 13  2  4  7  8 15  6 11  5  9  7  8  8  5  5  5 10  5 10 10  9  6  9]
[10  5  6  6  7  3 10  3  7  9  6  6 14  9  7  8  5  9  6  7  5  3  7  4  9  7  6  5  9]
[14  7  9 11 13  4 14  2  7 11  7 11  9 17  9 10  4  8  8  8  7  6 12  6 10  9  7  5 12]
[11  9  8 10 12  6 13  3  9 11  7  5  7  9 17 10  7  7  9  8  5  6  9  4  8  6  8  3 13]
[12  8  8 11 12  6 13  4  7  8  9  9  8 10 10 17  8 11  8  8  6  7  8  5 10  9  8  5 11]
[12  8  6  8  9  8 10  3  6  5  8  7  5  4  7  8 14  6 10  5  7  5  6  4  9  9  8  6  7]
[10  6  4  6  8  4 10  3  4  6  5  8  9  8  7 11  6 13  5  5  4  3  6  2  8  7  6  4  8]
[13  6  9  9  9  5 12  2  8  7  8  8  6  8  9  8 10  5 15  5  6  4  9  5 11 10  8  6 10]
[ 8  5  5  7 10  4  9  2  6  7  7  5  7  8  8  8  5  5  5 12  5  3  7  2  6  4  4  2  8]
[ 8  5  6  6  9  7  7  0  7  5  6  5  5  7  5  6  7  4  6  5 11  5  4  3  9  6  4  5  5]
[ 8  7  6  8  8  4  8  2  6  7  5  5  3  6  6  7  5  3  4  3  5 11  4  6  6  5  6  4  6]
[14  8  9 11 12  4 13  2  5  9  9 10  7 12  9  8  6  6  9  7  4  4 15  7  9 11  8  5 12]
[ 8  6  7  8  7  4  6  1  4  6  7  5  4  6  4  5  4  2  5  2  3  6  7  9  7  6  7  7  6]
[14  7 12 11 12  8 12  1  8  9 11 10  9 10  8 10  9  8 11  6  9  6  9  7 17 10 12  9  9]
[13  8  9 10 11  6 11  3  7  7  9 10  7  9  6  9  9  7 10  4  6  5 11  6 10 15  9  5 10]
[12  6 10 10 11  6 10  3  7  7  9  9  6  7  8  8  8  6  8  4  4  6  8  7 12  9 15  7 10]
[ 9  4  6  6  6  5  6  0  4  5  6  6  5  5  3  5  6  4  6  2  5  4  5  7  9  5  7  9  5]
[14  6 10 11 14  4 15  2  8 11  8  9  9 12 13 11  7  8 10  8  5  6 12  6  9 10 10  5 18]
sage: 
```
Fig. 2: SageMath rendering of $Q \times Q^T$

In Phase 2, the challenge provided a new sequence: **23457**. Using **oeis.org** to search for potential matches, we identified this as the *Powers of Primes* series. Given that this sequence used digits, we applied a modulo 10 transformation to maintain single digits. This yielded the next 16 numbers: **1271379391471391**, which successfully solved Phase 2.

Phase 3 began with a different type of encoded sequence: **bceg**. This time, lowercase letters indicated we were dealing with a separate series. After further analysis, we recognized this as the *Mersenne exponents* sequence. Extending this pattern, we decoded the next part of the sequence as **ssocqigu**.

Phase 4 introduced yet another variation, starting with the symbol sequence: **—!"**, where **-** represented a space. Guided by a hint, we identified this sequence as *Narayana's Cows*. Using this insight, we derived the following encoded symbols: **%&#(.!)'**.

With all four sequences correctly input, the motor began to rotate, with each rotation signifying a letter in the flag. At this point, we reset the Arduino to streamline the process. Instead of manually re-entering each sequence, we utilized a logic analyzer to map each motor movement to the encoded sequence, constructing a complete decoding dictionary.

Using this approach, we successfully uncovered the final flag:

FLAG: S3qu3NC3 (n0T sEQU!ns)

Week 2

KeyRing 1: This week's challenge requires us to leverage side-channel data from 3D printers collected during the printing of important physical keys. The objective is to use side-channel information to categorize the data into their respective keys. The dataset provided to the attacker in this first challenge is divided into *"labeled"* and *"unlabeled"* data.

Specifically, the *"labeled"* dataset contains files for four different types of keys: KeyA, KeyB, KeyC, and KeyD. For each key, a set of STL, MP3, and CSV files are provided, representing the audio of the print collected from a device near the printer, the vibrations of the print collected from an accelerometer near the printer, and the final representation of the printed object, respectively. The *"unlabeled"* dataset, on the other hand, is a collection of snippets from the previous prints, in the form of MP3 or CSV files. This latter detail is crucial for choosing the correct approach, as we will need to tailor our technique depending on whether we are provided with audio or vibration data for a print to classify.

The approach chosen to solve this challenge was to use *Machine Learning (ML)* models, training and validating them through *Supervised Learning*. The idea was to train two distinct models in parallel, one for audio data and one for vibration data, enabling us to categorize a data sample regardless of whether it is in audio or vibration form.

Detailed Approach

To achieve this objective, the final structure of the Python code was divided into the following phases: **Pre-processing**, **Segmentation and Synchronization**, **Features and Labels Extraction**, **Splitting into Training and Validation Sets**, **Hyperparameter Tuning**, **Training Phase**, **Predictions Phase**, and **Results Evaluation**.

0) Audio and Vibration Pre-processing

As the first step, we needed to process the available data to make it understandable to the model we intended to train. To do so, the audio files were corrected with a bash script because the audio duration reported in the file headers did not match the actual duration. Subsequently, they were loaded into the code using the `librosa` library, which also allowed us to identify that the audio is sampled at 44.1 kHz with 16-bit precision. Additionally, a band-pass filter was applied to reduce the present noise. Regarding the vibrations, the `pandas` library was used. These are sampled at a frequency of 500 kHz.

1) Segmentation and Synchronization of Audio and Vibrations

The next (crucial) step was to segment the audio and vibration data into frames of equal length, as they were collected at different frequencies and durations. To address the issue of MP3-CSV file tuples where one has slightly longer timestamps than the other, we chose to take the shorter duration and trim the longer one. Subsequently, the files were segmented into frames of 100 ms (0.1 seconds), although other segment lengths proved to be efficient with very similar results, such as 50 ms (0.05 seconds) and 200 ms (0.2 seconds). Shorter or longer segments each brought their own advantages and disadvantages in terms of precision and specificity; a 100 ms segment was chosen as the final selection for an optimal trade-off. Once we obtained equal-length data series segmented into equal frames, we ensured that the frames were synchronized with each other.

2) Features and Labels Extraction

To provide the processed data to the model in a usable format, we extracted for each audio and vibration file the feature matrices X_{audio} and $X_{\text{vibrations}}$, shaped as $\mathbf{X} \in \mathbb{R}^{n_{\text{frames}} \times n_{\text{features}}}$, where each row of the matrix $\mathbf{X}$ corresponds to a feature vector for a single frame.

Specifically, we extracted the following features:

Audio Time-Domain Features:

- Frame Energy
- Zero Crossing Rate (ZCR)

Audio Frequency-Domain Features:

- Short-Time Fourier Transform (STFT)
- Dominant Frequency
- Spectral Entropy
- Spectral Flux
- Spectral Centroid
- Spectral Bandwidth
- Spectral Contrast
- Chroma STFT
- Spectral Roll-off
- Tonnetz

Audio Combined Time and Frequency Features:

- MFCCs (Mel-Frequency Cepstral Coefficients)

Vibrations Time-Domain Features (for each axis X, Y, Z):

- Mean
- Standard Deviation (STD)
- Root Mean Square (RMS)
- Zero Crossing Rate (ZCR)
- Peak Values
- Skewness
- Kurtosis

Vibrations Frequency-Domain Features (for each axis X, Y, Z):

- Spectral Energy
- Dominant Frequency
- Spectral Entropy

Vibrations Cross-Axis Features:

- Correlation XY
- Correlation XZ
- Correlation YZ

For each frame's feature vector, a label was associated based on the file name from which $\mathbf{X}$ was extracted. For example, if the feature matrix $\mathbf{X}$ was extracted from the file `KeyA.mp3`, then each vector would be associated with the label `KeyA`. Since our assigned labels are in the form of strings, to make them understandable to the models, we encoded them using the `fit_transform` function of `LabelEncoder` from the `scikit-learn` library.

3) Splitting the Dataset into Training Set and Validation Set

Given the limited dataset provided for training our models, and especially the lack of feedback to verify our results within the challenge system, we believed that employing a Cross-Validation method was essential to present more valid results. Specifically, we implemented **Stratified K-Fold Cross-Validation**, adding `StratifiedKFold` to the model training pipeline.

4) Hyperparameter Tuning

To maximize the performance of our pipelines, we conducted Hyperparameter Tuning using a `GridSearch` to explore the parameters for our models. Specifically, the following parameters were optimized within the given value ranges:

Hyperparameter	**Value Range**
`classifier__n_estimators`	50, 100, 200, 500, 1000
`classifier__learning_rate`	0.001, 0.01, 0.1, 1.0, 10
`classifier__estimator__max_depth`	1, 5, 10, None
`classifier__estimator__min_samples_split`	2, 5, 10

TABLE II: Hyperparameter Tuning Parameters

Training, validating, and evaluating all parameter combinations took approximately $\sim$40 minutes. Finally, one of the best combinations of hyperparameters found for both pipelines, audio and vibrations, was:

Hyperparameter	**Audio Pipeline**	**Vibration Pipeline**
`classifier__n_estimators`	100	100
`classifier__learning_rate`	1.00	0.01
`classifier__estimator__max_depth`	1	1
`classifier__estimator__min_samples_split`	10	2

TABLE III: Optimal Hyperparameter Combinations

5) Training Phase

For the training phase, we chose to use `DecisionTreeClassifier` boosted with `AdaBoostClassifier` for both models (Audio and Vibrations). The parameters passed to these models were the `.best_params_` previously obtained from the GridSearch.

6) Predictions Phase

Predictions were made in two stages. The initial predictions were conducted on the dataset split into training and testing sets within the *“labeled”* dataset to evaluate the accuracy of the freshly trained models and obtain reliable results. Once satisfied with the obtained accuracies, the models proceeded to an additional prediction phase, this time on the *“unlabeled”* dataset. However, this time, predictions on the models were made using the `.predict_proba` method, which allows us to obtain the probability of a sample belonging to each key type (KeyA—KeyB—KeyC—KeyD). The results of these predictions would be the categorizations of the MP3/CSV files for which we need to determine the corresponding key.

7) Results Evaluation

The results of all the preceding steps were finally evaluated using scikit-learn’s accuracy calculation functions, such as `accuracy_score`. To better visualize the categorizations made by the models, confusion matrices were also calculated and plotted for each model.

Our research results concluded with excellent outcomes: an average accuracy of $\geq 90\%$ for the audio classifier model and an average accuracy of $\geq 95\%$ for the vibration classifier model. The average accuracies were calculated from the various results obtained from different optimal hyperparameter configurations, different training and validation set splits, and different frame granularities. Below are final classifications with respective probabilities for each prediction, based on the aforementioned results.

TABLE IV: Class Probabilities for Each File

File	**Predicted Class**	**KeyA (%)**	**KeyB (%)**	**KeyC (%)**	**KeyD (%)**
sample1.csv	KeyD	0.00	0.00	0.00	100.00
sample2.csv	KeyD	0.00	0.00	0.00	100.00
sample3.mp3	KeyA	99.90	0.00	0.02	0.08
sample4.mp3	KeyA	70.66	0.00	12.59	16.75
sample5.mp3	KeyC	33.45	0.00	34.93	31.62
sample6.mp3	KeyA	65.47	0.00	22.19	12.33
sample7.mp3	KeyA	99.99	0.00	0.01	0.00
sample8.mp3	KeyA	99.34	0.00	0.66	0.00
sample9.csv	KeyC	0.00	0.00	100.00	0.00
sample10.mp3	KeyD	5.99	44.04	0.00	49.97
sample11.csv	KeyC	0.00	0.00	100.00	0.00
sample12.csv	KeyA	100.00	0.00	0.00	0.00
sample13.mp3	KeyD	44.13	0.00	0.01	55.86
sample14.mp3	KeyC	26.99	0.00	73.01	0.00
sample15.csv	KeyA	100.00	0.00	0.00	0.00
sample16.csv	KeyD	0.00	0.00	0.00	100.00
sample17.mp3	KeyA	99.99	0.00	0.01	0.00
sample18.csv	KeyB	0.00	100.00	0.00	0.00
sample19.mp3	KeyA	90.13	0.00	9.84	0.03
sample20.mp3	KeyA	100.00	0.00	0.00	0.00
sample21.csv	KeyB	0.00	100.00	0.00	0.00
sample22.csv	KeyA	100.00	0.00	0.00	0.00
sample23.csv	KeyA	100.00	0.00	0.00	0.00
sample24.mp3	KeyD	0.00	12.50	0.00	87.50
sample25.mp3	KeyC	6.98	0.00	93.02	0.00
sample26.mp3	KeyC	33.66	0.00	66.34	0.00
sample27.csv	KeyC	0.00	0.00	100.00	0.00
sample28.csv	KeyB	0.00	100.00	0.00	0.00
sample29.mp3	KeyA	98.65	0.00	1.29	0.06
sample30.csv	KeyC	0.00	0.00	100.00	0.00
sample31.csv	KeyA	100.00	0.00	0.00	0.00
sample32.csv	KeyD	0.00	0.00	0.00	100.00
sample33.csv	KeyC	0.00	0.00	100.00	0.00
sample34.csv	KeyB	0.00	100.00	0.00	0.00
sample35.mp3	KeyA	82.44	0.00	17.56	0.00
sample36.mp3	KeyA	99.52	0.00	0.47	0.02
sample37.mp3	KeyA	60.71	0.00	0.00	39.29
sample38.csv	KeyC	0.00	0.00	100.00	0.00
sample39.csv	KeyB	0.00	100.00	0.00	0.00
sample40.mp3	KeyC	49.08	0.00	50.20	0.72

Week 3

Lizzy: Upon loading the firmware, the system prompted us with a series of integer values for calibration. It required numbers between 1 and 200 to proceed.

Once calibrated, the system began outputting pairs of numbers while rotating the servo motor. After running some tests, we observed that this "calibration" value was effectively controlling the servomotor's speed: the higher the value, the slower the motor moved.

Initially, the system printed the following values:

135	45
135	0

These values corresponded with movements of the servo motor, after which it stopped printing and only moved the servomotor.

Referring to the challenge image and a description mentioning boats, we quickly deduced that it represented a semaphore flag pattern. In this protocol, each angle represents an arm position, allowing us to reconstruct letters from the servo motor movements.

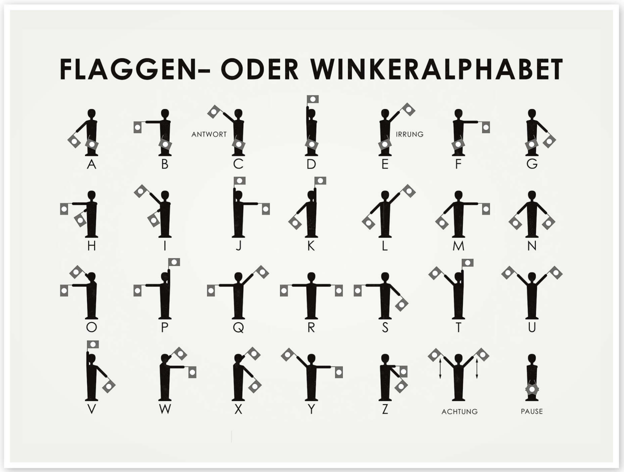


Fig. 3: Visual representation of the flag semaphore

The challenge now consisted of reconstructing arm positions using the servo output. Fortunately, we had access to an oracle that allowed us to input our sequences and observe the servo's reactions.

The servomotor exhibited specific patterns:

- Clockwise/counterclockwise rotation by a fixed angle, followed by a variable stop time.
- By measuring this stop time (achieved by connecting a servomotor and parsing the PWM to extract data), we noted that the time was not random. Instead, it was dependent on the given input.

Through logic and analysis, we tested inputs such as:

0	0
45	45
90	90

Each time, we observed either:

- Clockwise rotation with a 15 ms stop, or
- Counterclockwise rotation with a 110 ms stop.

We inferred that the stop duration represented an angle, rescaled by $360/125$, as an initial output stated "rotating by 360 degrees" and the servo halted at 125 ms intervals. After this rescaling, the stop-time closely matched 45° multiples. Clockwise and counterclockwise rotations were indicators of the delta's sign:

$$\text{Right} : +45^\circ$$
$$\text{Left} : -45^\circ$$

For example, starting from 0°:

- Rotate right by 45° to reach 45°.
- Rotate right by another 45° to reach 90°.

Using this methodology, we translated servo movements to arm positions and vice versa, ultimately revealing the message:

CIPHER ZJXLZMOEZVREJBXRUIRWYGLHVPTD-KWZO

Testing the Vigenère cipher decoding using `dcode.fr/vigenere-cipher` with "Automatic Decryption," we identified the probable key: `RITA`. The resulting plaintext read: **IBELIEVEINYESTERDAY-WHYSHEHADTOGO**, a well-known line from the Beatles' song "Yesterday."

Continuing with this, we encoded the phrase "**IDONTKNOW**" (a further line from the lyrics) back into the Vigenère cipher, yielding the ciphertext **ZLHNKSGON**. Converting this into semaphore angles for flag signaling, we derived the following sequence:

```
[(315,270), (225,45), (90,45), (315,45),
(180,45), (315,90), (315,0), (135,90),
(315,45)]
```

Upon submission, the console displayed the message:

```
/********************  YOU BEAT THE
    CHALLENGE!!!   ********************/
/********************
    Congrats!!!
    ********************/
```

Fast & Max: In this challenge, we were tasked with analyzing a simulated banking system to retrieve an employee's card number and PIN. The challenge was divided into two main tasks:

1) Decrypting an employee card number encrypted using an RSA-like scheme.
2) Unlocking a second safe with a rotor-lock mechanism.

To reverse engineer the code, we used `Ghidra` and compiled Arduino code ourselves to pattern match the function signatures. This allowed us to approximate the Arduino code, including standard functions like `print`, `delay`, and `input`.

Upon booting, the challenge initiated a rotation of the servo motor, outputting the card number in 4-digit chunks. The motor's PWM signals and timing followed a pattern:

$$X, Y, Z, Y, Z, Y, X, Y, \dots$$

Here, Y appeared as a divisor of time, while X and Z represented bits. Extracting the bit sequence yielded `1101110000011`. In the reversed code, we observed the sequence was transmitted from the *Least Significant Bit* (LSB) to the *Most Significant Bit* (MSB). By reversing the string and converting it to a number, we obtained:

First quartet: 1101110000011 $\rightarrow$ 1100000111011 $\rightarrow$ 6203
Second quartet: 11101110010001 $\rightarrow$ 10001001110111 $\rightarrow$ 8823
Third quartet: 1001010110001 $\rightarrow$ 1000110101001 $\rightarrow$ 4521
Fourth quartet: 10010100100001 $\rightarrow$ 10000100101001 $\rightarrow$ 8489

Concatenating the four decoded quartets, we obtained the complete 16-digit Employee Card Number:

6203 8823 4521 8489

For the second part, we observed that the stepper motor rotated clockwise for X times and counterclockwise for Y times, depending on the length of the PIN input. The rotation patterns alternated between clockwise (**R**) and counterclockwise (**L**) for various PIN inputs. Examples include:

1 : RL
1 : RLR
1 : RLRR
2 : RRLLR
2 : RRLLRR
2 : RRLLRRR
...

Each letter in the alphabet corresponded to a specific degree of rotation, with **A** mapping to 1°, **B** to 2°, **C** to 3°, and so forth. For instance, sending the PIN `AB` caused the motor to rotate right by 1° and left by 2°. The input `AABBB` resulted in 1° right, 1° right, 2° left, 2° left, and finally 2° right.

Beyond a certain rotation threshold, the motor emitted a deeper sound. By testing the rotation sequence `ZZZZZ`, we observed that halfway through the second `Z` rotation, the sound shifted to a lower pitch. This sound change was reflected in the motor's signal, as it shifted from a sample rate of 6.23 ms to 4.1 ms. By counting the number of 4.1 ms samples, we could deduce the correct degrees of rotation.

Week 4

Safecracker 1: This challenge involved analyzing an MP3 recording of a safe with a rotor lock mechanism. The task was to identify the code required to unlock the safe by interpreting the sound produced as the rotor moved.

Fortunately, the lock is driven by a motor, so the time taken for each rotation increases linearly with the input size. From the demo, we extracted the time required for one step rotation, approximately 31.63 ms. By examining the MP3 file (or the waveform graph), we could determine the number of steps taken by measuring the duration of each rotation.

Below is an example of the recorded movements, and in Figure 4, the waveform plot clearly illustrates variations in sound duration, corresponding to different lengths of rotation.

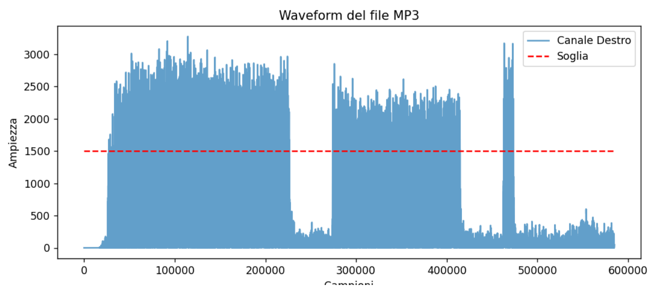


Fig. 4: Waveform plot showing variations in rotation duration

```
Movement lasted 4144.54 ms
Movement lasted 2926.77 ms
Movement lasted 215.29 ms

Interval: 4144.54 ms
Number of steps: 131.01
Interval: 2926.77 ms
Number of steps: 92.51
Interval: 215.29 ms
Number of steps: 6.81
```

By extracting the duration of each rotation from the recording during Phase 1, we determined the code sequence as follows:

1) 131 S
2) 92 S
3) 7 S

A. *Safecracker 2*

Following the same methodology as in Phase 1, we extracted the duration of each of the five rotations. To determine the rotation mode, we leveraged the distinct mean absolute amplitude associated with each mode as we can see from the Fig 5, which we obtained from the audio signal during the demo phase.

Below are the recorded intervals, number of steps, and mean amplitude values:

```
Interval: 6283.21 ms
```

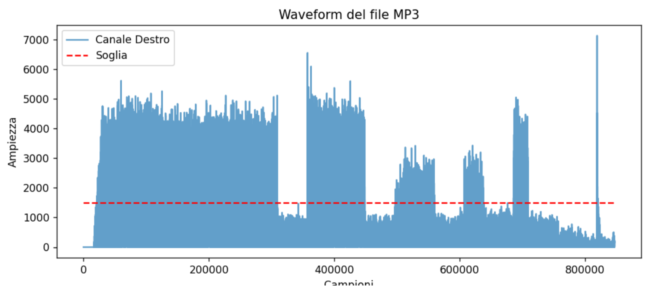


Fig. 5: Waveform plot showing variations in rotation duration and amplitude

```
Number of steps: 195.92
Mean amplitude: 670.01
Interval: 1909.81 ms
Number of steps: 60.37
Mean amplitude: 775.43
Interval: 1292.60 ms
Number of steps: 40.86
Mean amplitude: 412.85
Interval: 637.17 ms
Number of steps: 20.14
Mean amplitude: 403.55
Interval: 485.17 ms
Number of steps: 15.34
Mean amplitude: 686.32
```

By analyzing these values and comparing the mean amplitudes, we successfully determined the rotation modes for each interval and completed the final challenge. The resulting sequence was:

1) 196 I
2) 60 D
3) 41 S
4) 20 S
5) 15 I

IV. Conclusion

In this report, we demonstrated a practical approach to hardware-based challenges by focusing on signal analysis and hardware interaction rather than conventional reverse engineering. Using a logic analyzer to capture binary signals, we were able to interpret encoded data, identify sequence patterns, and solve matrix transformations, which ultimately revealed encrypted information. This signal-driven methodology proved effective in analyzing timing patterns, motor sequences, and encrypted messages, enabling us to decode complex challenges on an Arduino-based system.

For challenges involving machine learning, we applied models to classify audio and vibration data, achieving high accuracy and reinforcing the versatility of combining hardware analysis with data-driven techniques. By systematically approaching each phase, we developed a deep understanding of side-channel and timing-based vulnerabilities, showcasing how even standard hardware components can be leveraged to uncover sensitive information.

Overall, this experience highlighted the value of signal-based analysis and non-invasive techniques in embedded security contexts, where insights into system behavior can reveal critical information without direct code manipulation. The methodologies employed here provide a foundation for future exploration of similar side-channel challenges, particularly in the context of embedded and IoT systems.