CSAW ESC 2025 Final Report

Ilias Fiotakis
Department of EECE
Democritus University of Thrace
Thrace, Greece
iliafiot@ee.duth.gr

Vassilis Ananiadis
Department of Digital Systems
University of Piraeus
Piraeus, Greece
e21005@unipi.gr

Meletis Michail
Department of EECE
National Technical University
Athens, Greece
melmichail@gmail.com

Christos Papadopoulos
Department of Informatics
Open University
Athens, Greece
std168294@ac.eap.gr

***Abstract*—This document presents our team's response to the challenges proposed for the final phase of the CSAW 2025 Embedded Security Challenge. It details the methodologies and research conducted for each task, outlining our systematic approach to analysis, solution development, and optimization. The report is organized into three main sections: *Introduction*, *Methodology*, and *Challenges*. In the final section, we provide comprehensive evidence and explain the procedures followed to achieve each solution, including the integration of AI, machine learning, and large language models (LLMs), as well as mitigation strategies implemented for each challenge.**

***Index Terms*—Embedded systems, hardware security, fault injection attacks, side-channel analysis, power analysis, differential power analysis, timing attacks, cryptanalysis, artificial intelligence, machine learning, large language models.**

I. Introduction

We used the ChipWhisperer Nano platform to perform side-channel analysis and fault injections on the challenges provided. We combined the ChipWhisperer API [3], along with open-source signal processing and ML tools, to automate trace analysis and cryptanalysis.

The core team was based in Athens; to increase throughput, we acquired three additional ChipWhisperer units, maintained a private GitHub repository to track code and results, and coordinated daily via Discord.

II. Methodology

Our approach consisted of five interleaved phases - Analysis, Exploitation, Optimization, AI/ML Integration, and Mitigation. The phases were performed mostly linearly, and paved the way for each progression. The phases were assigned to one or more members according to proficiency and experience. The approach for each phase on the challenges will be discussed in Section III, and Fig. 1 is an optical visualization of our workflow.

A. Analysis

The first step to our methodology was to perform a thorough examination of each problem at hand and ensure that we have identified all the possible exploitation paths. To identify leaks, we checked the regions between the `trigger_high` and `trigger_low` functions, since that was the enumerable region through the ChipWhisperer.

The released source codes were also utilized, by adding the redacted parts and developing a process to flash our compiled firmware into our boards. This was achieved by leveraging ChipWhisperer build files found inside the official Github repo, where we replaced the target sources for compilation. This aided our analysis and understanding of each challenge's setup before trying our approaches on the tasks. It should be noted that this was possible due to our ability to flash custom firmware, as real world cases usually require a more black box approach.

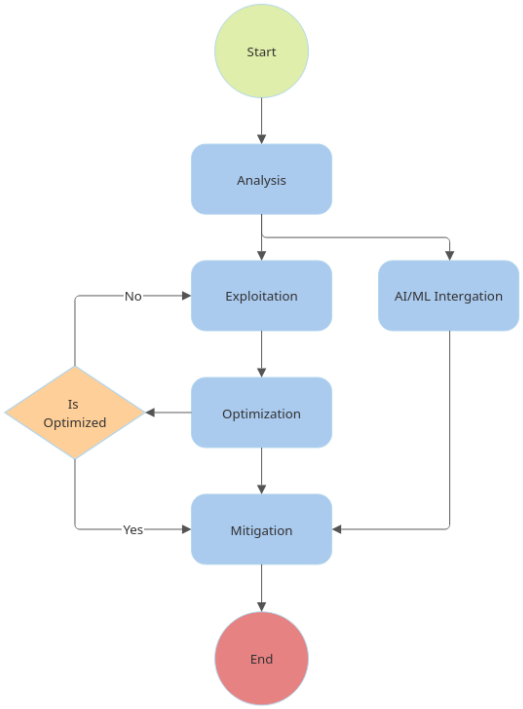

Fig. 1: Methodology Flow

Finally, for each challenge we created a ChatGPT conversation using the GPT-5 model. We would start the conversation by providing the source code, our initial ideas, and asking it for suggestions, and we would further the discussion with the model to refine our scripts and approach.

Towards the beginning of the competition, we developed a handful of scripts which could be used on every challenge. The tools developed are a script to perform trace capture, and an analyzer script to perform visualization analysis on

the generated graphs.

B. Solution

For developing our solutions, Python was our primary language of choice due to it's easy-to-use API to communicate with the ChipWhisperer board, and the many libraries available for use like numpy, matplotlib, z3, etc.

The aforementioned trace capturing script was always executed first. It collected a big sample of trace data, and afterwards it performed noise reduction on them by averaging out the traces, before finally storing them. Then the analysis script generated graphs, which would be used to start our investigation.

Based on our analysis of the provided source, the generated graphs, and searching for public research, we would identify what type of attack we should perform, and proceed with implementing it.

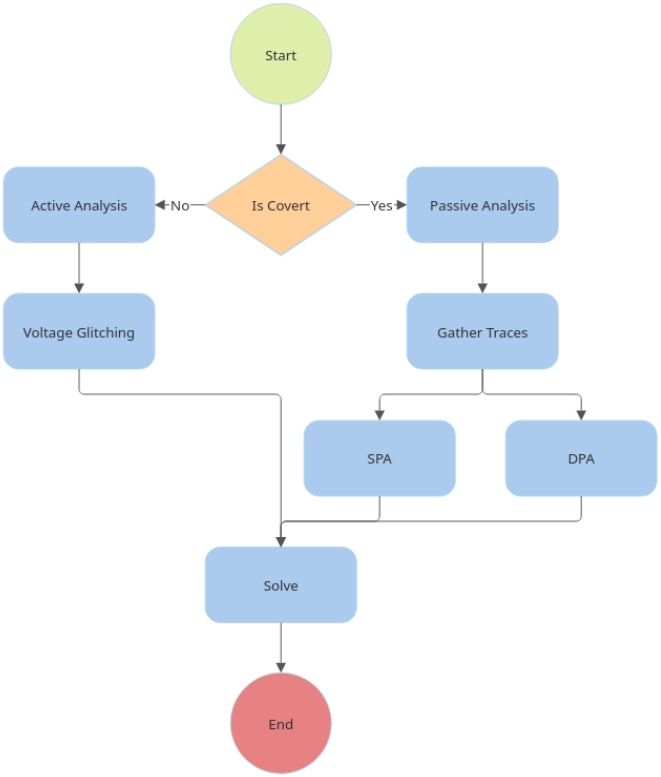

Fig. 2: Solution Flow

Exploitation was always attempted first on our test environment. After successful exploitation, we finalized the solver and attempted to run it against the real firmware. In case of failure, we investigated the root cause, which usually was fixed with some modifications to the solver. Once a complete proof of concept was developed, we would move the automation of the solver to the optimization part of our process. Fig. 2 shows the typical flow of our solution engineering process.

C. Optimization

Our optimization approach emphasized minimizing the queries performed. While researching ways to achieve this, we split the optimization techniques based on our categorization of the challenges. For password checking challenges, we defined what is considered a timing/power anomaly to reduce the range of letters to check. For sorting challenges we used caching and algorithms like binary search. And for the rest of the challenges, we developed custom algorithms.

D. AI/ML/LLM Integration

We first relied on LLMs to surface relevant hardware attacks, ChipWhisperer examples, and timing/DPA/CPA PoCs to get a headstart. We also researched the idea of wrapping the workflow behind an MCP-style server to unify traces, notes, and code generation, which was developed but not heavily utilized due to having created a workflow without it. Besides LLMs, the models were used to turn informal POCs into runnable scripts and second to swap-in classical ML components (k-means, logistic regression, tree ensembles, one-class SVMs) to stabilize noisy measurements. All experiments were run in Jupyter with standard Python tooling, letting us iterate quickly on parameters, feature extraction, and perform changes on the code itself.

E. Mitigation

Our approach on mitigations is based on a paper that discusses them [7]. We recorded the solutions discussed there, and for each challenge we evaluated which mitigations could be applicable.

III. Challenges

This section consists of three subsections, one per challenge set provided during the competition. Each of the subsections contains a summary box for each challenge, accompanied by a discussion for each of the methodology steps presented above.

A. Set - 1

Gatekeeper 1/2 - 100/150

```
Algorithm: Password Verification
Attack type: Timing Attack
Challenge solution:
gk1{l0g1npwn}
gk2{7rU3ncrYkIND}
Queries: 142/376 or 2881/14k(ML)
```

`Analysis:`

This was a timing attack challenge. The concept is that we are given two different passwords, the first of length 8 and the second of length 12, excluding the flag format. For the first password, when providing the correct character we get a delay of 5000-i (where i is the char's index) and for the second one 2500-i.

`Solution:`

Our solution is that for every possible alphanumeric character 'x' we send a query with flag 'gk1{x!!!!!!!}' and measure the executed time. We used the character '!' as padding since it does not belong in the password character set. Iterating through our alphabet, we will reach the correct character, which will cause a delay we can measure. Therefore we can append the correct character to the known flag, and proceed to the next index.

The correct method of measuring the time it takes for code sandwiched between `trigger_high()` and `trigger_low()` using a ChipWhisperer would be using `scope.adc.trig_count` in Python. However we soon realized that the CW nano does not have that capability (`trig_count` can be used by CW lite and above). We had to rely on measuring the entire execution with Python's `perf_counter()` which worked for Gatekeeper 1, but gave us inconsistent results on Gatekeeper 2. Eventually our solution script produced the first 6 characters of the flag on a 2024 CW nano and the rest of the flag on a 2025 CW nano.

Optimization:

The proposed optimization terminates iteration for a given index upon detecting a significant time differential, which indicates the correct character. This method reduced the queries required to retrieve the first password from 288 to 142.

AI/ML Integration:

To improve stability, we wrapped the timing step in a lightweight ML layer instead of hard-coding a single threshold. For each index we queried every candidate multiple times (10 samples per candidate) and collected a 1-D timing vector. Because the firmware's delay increases only when the current prefix is correct, we expect timings to form two natural groups: 'fast' (rejected early) and 'slow' (correct prefix, extra loop). We ran a 2-cluster k-means on these timings and automatically labeled the cluster with the higher centroid as the "slow" one. Then, among all samples assigned to that slow cluster, we picked the character that appeared most often, i.e. the character that consistently produced the longer execution time. Repeating this procedure per index gave us the full password without manually tuning per-board thresholds. This was especially useful because as mentioned the raw timings contained host-side jitter. Clustering absorbed that jitter and still exposed the one character that caused the extra delay.

For Gatekeeper 2 traces were noticeably noisier than the first Gatekeeper, so instead of a single loop where we measure once per candidate, we used an adaptive, Bayesian-style solver. For every character position we first measured a fresh baseline (all padding) to remove long-term drift, and then we sampled every character in the alphabet a few times to get initial means and variances. We modeled each candidate's time as a normal distribution and used Monte-Carlo draws to estimate the probability that this candidate is the slowest one, i.e. the one that made the device advance to the next comparison. If no character was clearly dominant, we only re-sampled the top few contenders, updating their posteriors until one reached a confidence threshold or at least appeared as 'top' often enough. This made the attack robust to jitter, spurious delays, and occasional timeouts, while still converging to the correct 12-byte flag.

Mitigation:

The mitigation for such an attack [2] is to make sure that the same amount of time is taken on any given character check, whether it is correct or not. We can take this a step further and say that exactly the same number of instructions must be executed whether the character is correct or not.

Sorters Song 1/2 - 150/200

```
Algorithm: Insertion Sort
Attack type: SPA, DPA
Challenge solution:
ss1{y0u_g0t_it_br0!}
ss2{!AEGILOPS_chimps
Queries: 259/517 or 259/517(ML)
```

Analysis:

The code implements an insertion sort algorithm with a power analysis vulnerability in the `sort_data` function that introduces data-dependent delays during comparisons. We can submit values to be inserted into copies of the secret array and trigger the sorting operation while measuring power consumption. By analyzing the power traces from multiple sorting operations with different input values, attackers can infer which comparison branches are taken during the merge process, gradually revealing the contents and ordering of the original hidden array. An example of this is shown in Fig. 3.

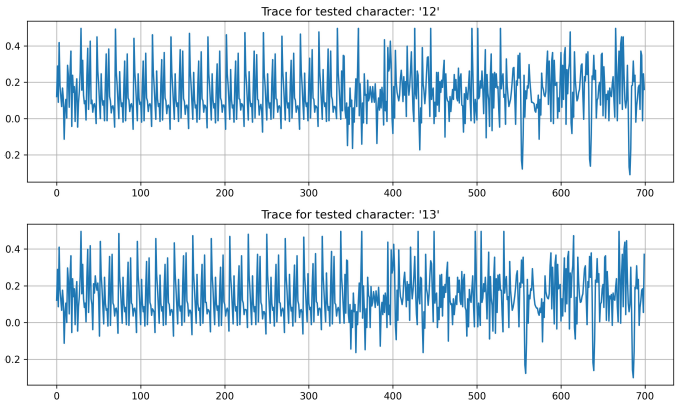

Fig. 3: Power trace for numbers 12–13 — shows a bit-shift occurring at $\tilde{5}0$ samples

Solution:

To solve this challenge we insert an element of our choosing into the array and measure the power it takes for the chip to sort this modified array. Our initial approach was to first call `reset` to reset the array, then replace an element by using `get_pt` and after that take the traces of an 8-bit number by calling the `sort_data` function. We repeated this process for all 256 possibilities and observed the traces. Each time the element we were testing was greater by 1 than the one

checked before, that meant that we found the secret element at that position.

For the second version however, brute forcing and performing the same attack would require around 2 * 2**16 queries, so we started thinking about optimization techniques, which are discussed below. So we implemented a binary search for every offset.

Optimization:

As mentioned, the second version of this challenge needed improvement in our approach for the first version. Our solution was a binary search for each offset. We made sure that our binary search was performed up until the last element was found and not on the entire initial search space, since the array had been sorted up to that point. After that we also added a cache, a dictionary in Python which stores the power trace we got for each value-index pair so that no queries should be performed on the same index more than once.

AI/ML Integration:

The side channel strategy for this challenge was to perform a binary search on each array element by comparing three traces. One for a "low" guess, one for a "mid" guess and one for a "high" guess, and then decide whether the true value lies in the lower or upper half based on which pair of traces looked similar. This works, but it is inconsistent. On some traces the L1/L2 difference between `low→mid` and `mid→high` is small, and a single noisy decision can push the search in the wrong direction, forcing extra queries or a complete restart. To make the branching step adaptive, we added a lightweight ML layer. For every search step we built a feature vector from the three traces (pairwise L1 and L2 distances, per-trace mean/std/range) and recorded which way we actually branched. After collecting a minimum number of such labeled steps, we trained (and periodically retrained) a logistic regression classifier to predict 'go left' vs. 'go right'. The solver first falls back to the original heuristic when there is not enough data, but once the model has seen enough samples from the real device it starts making the branch decisions itself. This has two benefits:

1) it smooths over measurement noise and small timing/power drifts across iterations.
2) it reuses information gathered while solving earlier indices to solve later indices faster, since the array is strictly decreasing, and later searches operate in a smaller value range.

Mitigation:

One mitigation to this issue is adding random delays inside the critical areas of the code so that the power trace is essentially random when measured.

Critical Calculation - 100

```
Algorithm: Calculation and
Comparison
Attack type: Voltage Glitching
Challenge solution:
cc1{C0RRUPT3D_C4LCUL4T10N}
Queries: 430 or 1822(AI)
```

Analysis:

The concept of this challenge is fault injection, specifically a glitch attack. We are tasked with intercepting a nested for loop responsible for a counter calculation. After the for loop executes, 'cnt' should be 8000. If not, we will get the flag. So we need to perform a glitch attack on the loop.

Solution:

To perform this attack, we start by performing glitch attack attempts using the CW nano glitch API, with random offsets and repeat values. We got consistent results, but they varied among the CW devices. We generalized the range of the offsets to 1400-1800 and the range of the repeats to 9-12, which have been shown to consistently perform the attack on all devices.

Optimization:

By performing several tests and calculating the average of the successful repeat and offset values, we can shorten the ranges that we use for these values, thus minimizing the number of attempts. However, this approach seems to be device-specific.

AI/ML Integration:

Glitching this target came down to finding a narrow combination of two parameters, the external glitch offset and the number of repeats that causes the nested loop to misexecute and produce a counter value different than 8000. A random search over this 2-D space works, but it is slow and very device dependent. The ideal time window shifts slightly between CW nanos and even between reboots. To speed up this process, we wrapped the search in an adaptive ML loop. Every glitch attempt produced a tuple (offset, repetition) and a binary label that says whether the response looked 'interesting' (i.e. not the normal diagnostic string) or not. We fed these samples to a small MLP classifier (16×16 hidden layers) that approximates P(success offset, repeat). After an initial warm-up with purely random attempts, subsequent attempts were chosen by sampling many candidate parameter pairs and scoring them with the model, while still keeping a fixed fraction of random exploration to avoid local optima. In practice this quickly concentrated attempts around the device specific range (e.g. offsets in the 1400–1800 range with repeats 9–12), reducing the number of trials needed to trigger a fault and making the attack portable across slightly different boards.

Mitigation:

One valid mitigation for fault injections and glitch attacks specifically is to add software side security checks. For example in our case, calculating the 'cnt' variable more than once and comparing the results would make for a viable patch and make this attack significantly harder.

B. Set - 2

Dark Gatekeeper - 200

```
Algorithm: Password Verification
Attack type: SPA Timing attack
Challenge solution:
ESC{J0lt_Th3_G473}
Queries: 291 or 1610(ML)
```

Analysis:

Our approach to this was similar to Gatekeeper 1, 2. We did not have any stalling computations to identify delays between checks, but we could see how the power traces looked like. In Fig. 4 we see the power trace of 'a'. With some minor differences due to noise, every trace looked the same except one. The difference between them is evident, providing us with a distinctive pattern to capture the correct characters.

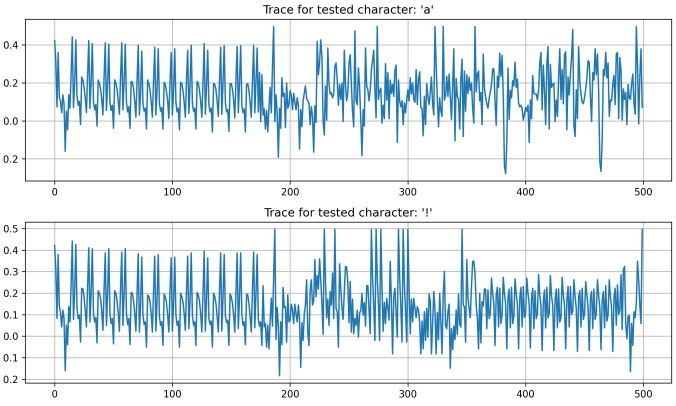

Fig. 4: Comparison of traces: incorrect final character ('a') vs correct character ('!').

Solution:

First, we recorded a reference trace using 12 space characters. Then, recovering the key from left to right, we fixed the known prefix, brute-forced the next byte over a printable character set, and padded the remaining positions with spaces to keep later iterations consistent. For each candidate, we captured the loop's power trace and computed the Pearson correlation with the current template. A correct byte changes iteration i from a mismatch to a match, altering the trace and thus *de*-correlating it from the template. We accepted the candidate once the correlation dropped below an empirical threshold (approximately 0.7), updated the template to that new trace, and repeated this process for all 12 indexes.

Optimization:

Just like Gatekeeper 1 and 2, we calculated a threshold to provide us with an early exit on success (specifically 0.7) so that we do not have to keep searching through the charset if we have already identified the correct character.

AI/ML Integration:

For the AI integration, we converted the side-channel search into an unsupervised anomaly-detection problem. For each password position, we built a background model of wrong guesses. We collected traces from random (almost surely incorrect) characters, standardized them, reduced dimensionality with PCA, and fit a One-Class SVM on the PCA space. Given this model, we tested every candidate character, scored its trace with the SVM decision function (more negative -¿ more anomalous relative to the wrong-guess distribution), and picked the most anomalous candidate as the correct one. We then repeated the process for the next position. This replaces a single weak correlation threshold with a noise-tolerant, data-driven detector.

Mitigation:

To mitigate this power analysis attack [2], we can introduce constant-time, data-independent processing so power correlations no longer reveal key bytes. Specifically, replace the early-varying branchless comparison with a single constant-time memcmp-style routine that accumulates differences (e.g. XOR all bytes and OR into a flag) and perform it while masking or adding random noise/dummy operations to equalize power consumption.

Hyperspace Jump Drive - 200

```
Algorithm: XOR
Attack type: DPA
Challenge solution:
b64{c3RhcmR1c3Q=}
Queries: 1280(Stable) or 1280(ML)
```

Analysis:

The source provides several operations, including one that checks if a submitted sequence matches the secret. The vulnerability exists in the polarity inversion function that XORs a user-supplied mask against each byte of the secret sequence while power measurement triggers are active. The XOR operation processes each secret byte individually with deliberate delays between iterations, creating distinct power consumption patterns that leak information about the secret values through differential power analysis. The spikes in Fig. 5 signal encryption.

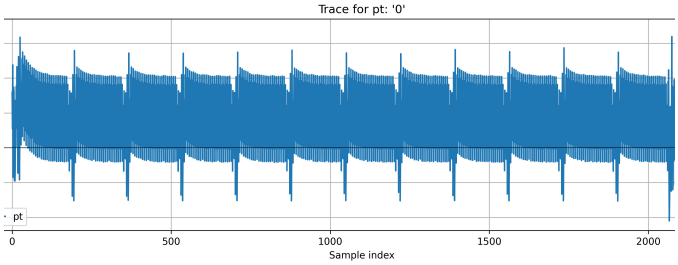

Fig. 5: Trace showing inverted polarity with plaintext 0.

Solution:

Our solution collects power traces from the `invert_polarity` command across all 256 single byte masks. Inspired by the common AES attack [4] [6], we

average repeated captures per mask to reduce noise, compute differential traces per bit by grouping traces where a given bit of $mask \oplus secret$ is 0 or 1, and then sample those DPA peaks at offsets corresponding to each secret byte's lsb to reconstruct the 12 secret bytes bit-by-bit. The lsb's can be shown visually in Fig. 6.

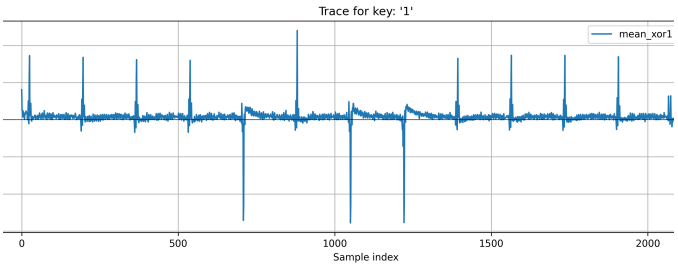

Fig. 6: DPA plot revealing the least-significant bits (LSBs) across each byte.

Optimization:

We optimize noise and capture time by averaging a small number of repeated traces per mask (around five displayed consistency) to get a stable template. Then we collapse the 256 averaged traces into per-bit differential traces so that we only need to analyze eight DPA traces instead of 256 full traces. Finally, we sample fixed offsets for each byte rather than performing expensive per-sample correlations.

AI/ML Integration:

Even after signal-based snapping, some acquisitions still produced peaks that were one sample too early or too late, which is enough to ruin the reconstructed 12-byte sequence. To make the peak selection robust, we trained a lightweight classifier (`ExtraTreesClassifier`) on the trace itself to tell real peaks from background noise. We synthesized the training data as follows:

- **positives:** for every coarse peak we took the feature vectors of its neighbors $(p-1, p, p+1)$ and labeled them as peak=1.
- **negatives:** we sampled random indices far from any coarse peak and labeled them as peak=0.

Each feature vector contained the DPA 'energy' in a small window around the candidate index and the per-bit absolute DPA values in the same window. At inference time, for each of the 12 coarse peaks we searched only a very small neighborhood (± 3 samples), scored candidates with the trained model, and applied an energy guardrail. The ML refinement was allowed to move the peak only if the new position had equal or higher signal energy. This design could noy replace a classic DPA approach, but rather support it. It could correct jitter-induced off-by-ones, but it could not invent arbitrary new peaks. The final ML-refined 12-peak set was then fed to the original bit-extraction function producing a valid ignition sequence.

Mitigation:

A straightforward mitigation would be to add random masking to the secret before the XOR operation, such that instead of computing mask XOR `secret_byte` directly, the device would first generate a random value *r*, XOR the secret with r to create a masked secret, then perform mask XOR (`secret_byte` XOR *r*), and finally XOR the result with *r* again to recover the correct output. This ensures that the intermediate value being XORed varies randomly across different executions even with the same mask input, breaking the correlation between the mask value and the power consumption patterns.

Ghost Blood - 200

```
Algorithm: Salsa20
Attack type: SPA, DPA
Challenge solution:
ESC{Th*t'sT*eSp1*1t!}
Queries: 496
```

Analysis:

This challenge implements a 16-bit variant of the Salsa20 stream cipher [5] [1]. The core cryptographic operation is the block function, which iteratively mixes a 16-word state matrix over twenty rounds using a Quarter-Round function. The initial state is constructed from public constants, a fixed nonce, and the secret 8-word (16-byte) key.

The system exposes a shift function via a serial interface. This function allows a user to execute the block operation with custom rotation constants. The critical feature lies within the ROTL function, which is called numerous times during the cipher's execution. ROTL contains a data-dependent conditional branch `if (a >= (1<<(16-b)))`.

This breaks the challenge because it is easy to distinguish which branch every capture the codeflow based on if *a* overflows or not. In Fig. 7 we can see a slight shift around the start.

Solution:

To attack the scheme, we made a Python implementation of the algorithm to experiment locally and explore the best avenues of exploiting the leaks. Our initial approach used Z3, with which we modeled the key as an array of symbolic variables. Then we constrained those variables by matching the simulated oracle outputs to the values the symbolic variables should produce, and attempted to concretize the symbolic key by finding a satisfying model.

In our first attempt we attempted to model the entire encryption flow with Z3. However the cipher's ARX (add-rotate-XOR) design changed the key between rounds, making it unable to be modeled with SMT solvers. It should be noted however that our POC worked for the first few rounds.

That observation motivated a per-round strategy, where we extract traces for a single round, modify the cipher's shift values, and collect a new set of traces for those shifts. Repeating this process sufficiently many times will eventually constrain the symbolic variables to a single valid key. Essentially we

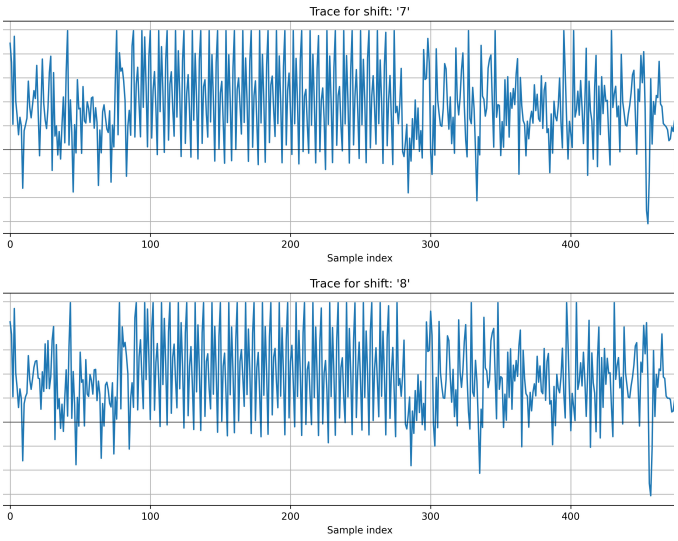

Fig. 7: Reference trace: non-overflow rotation vs. overflowed rotation

create an oracle that takes a certain shift array, and returns whether the ROTL is in the second or third branch for every threshold we put.

Optimization:

We tried both random and specially crafted shifts and were able to reduce the required number of distinct shifts to 30, which corresponds to 30*16 traces. However, this number is highly dependent on the specific key and may not be as effective at uniquely determining others.

Depending on the attacker's threat model and resources, there is a trade-off between trace collection and offline brute-force. An attacker can collect a set of traces that leaves the system under-constrained and then enumerate the models that satisfy those constraints until the correct keystream (and therefore key) is identified.

AI/ML Integration:

In this challenge, we aimed to replace the manual/brute-force step (selected shift) with a supervised machine-learning model. Instead of correlating every captured trace with 16 reference traces and applying a fixed threshold, we wanted to train a lightweight classifier that directly predicts the branch taken for a given shift configuration. However, time and complexity were obstacles in this implementation, leaving it as unfinished work for the future.

Mitigation:

The most direct mitigation would be to eliminate the conditional branch in the ROTL function that creates the timing side-channel. Instead of checking whether `a >= (1<<(16-b))` and executing different code paths, the function should always perform the full rotation operation `ret = ((a) << (b)) OR ((a) >> (16 - (b)))` regardless of the input value. This constant-time implementation ensures that all rotations take the same execution path and time, preventing attackers from distinguishing between different intermediate values through timing measurements.

C. Set - 3

Alchemist Infuser - 200

```
Algorithm: XOR
Attack type: DPA
Challenge solution:
a1c{Wh1teDragonT}
Queries: 1280
```

Analysis:

This challenge implements a cryptographic service using XXTEA encryption where the goal is to recover a hidden encryption key through differential power analysis. The program exposes three operations: encrypt, decrypt, and verify functions accessible through a serial interface. During encryption, input data is XORed with bytes from a secret key within a power-triggered window, followed by XXTEA encryption. The critical vulnerability lies in this XOR operation which leaks information through power consumption patterns that correlate with the secret key values. Fig. 8 showcases the critical offsets.

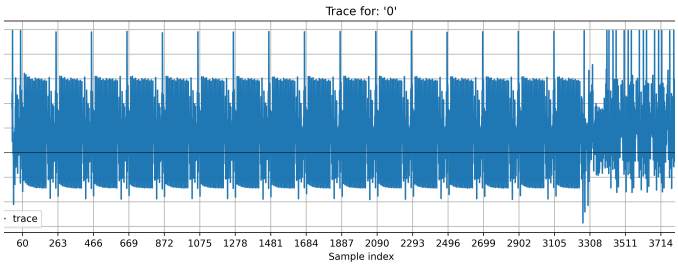

Fig. 8: Plaintext 0 trace analyzed to determine leakage offsets.

Solution:

Our solution performs a DPA attack [4] [6] to recover the 16-byte secret key by exploiting power consumption patterns during the XOR operation in the encryption function. The attack works in two phases. First, it collects power traces while encrypting controlled plaintexts (all bytes set to values 0-255), then it performs DPA by grouping traces based on hypothetical bit values of the XOR result and computing mean differences to identify correlations that reveal the actual key bits.

The first 8 bytes of the key are recovered by analyzing power consumption at specific time offsets corresponding to the first half of the XOR loop. This is similar to our approach on HyperSpace. Then these recovered bytes are used to adjust the analysis for the remaining 8 bytes. Fig. 9 displays the result from the first bit of the first byte which is zero.

Optimization:

We have identified two ways of solving this challenge. The extraction of the first 8 bytes of the key is the same for both methods. For the second part of the key however, we can either get new traces by sending $foundkey \oplus pt$ from the chip and

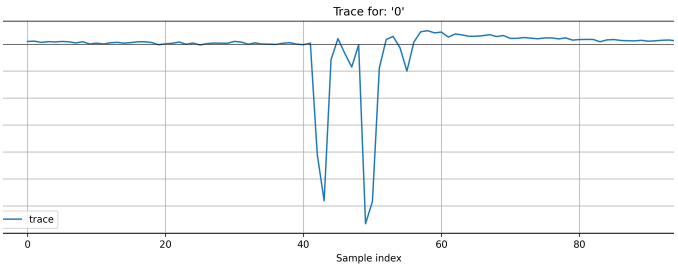

Fig. 9: DPA result exposing the LSB of the first byte.

perform the same attack, or we can try to perform an analysis on the existing traces and cut the queries in half. We proceeded with the second idea. We recover the first bytes, and then we developed a custom DPA algorithm which took as input the first half, and recovers the last 8 bytes using the already samples traces. Unfortunately we did not have time to script the extraction because the offsets where not constant so we plotted each critical area separately and extracted the bits by hand.

AI/ML Integration:

We built an automated side-channel solver that follows a light ML model. We capture 256 averaged traces (one per input mask), derive 8 differential traces (one per bit), automatically find the 12 repeated XOR points, and at each point classify the DPA sign to get one byte using the correct mapping. Our run did not recover every byte successfully, but it recovered enough to show potential, and with more tuning (better peak picking, wider classification window, ...), we expect a full key recovery.

Mitigation:

The most effective mitigation is Boolean masking, which breaks this correlation by introducing a random mask value. By masking both the plaintext and key bytes with the same random value before the XOR operation, then unmasking the result, the intermediate computation becomes statistically independent of the actual secret key. This forces attackers to either develop significantly more complex higher-order DPA attacks or collect orders of magnitude more traces, rendering practical key recovery infeasible with standard DPA techniques.

Echoes of Chaos - 200

```
Algorithm: Merge Sort
Attack type: SPA, DPA
Challenge solution:
eoc{th3yreC00ked}
Queries: 271 or 271(ML)
```

Analysis:

The code implements a merge sort algorithm with a power analysis vulnerability in the `clydeSort` function that introduces data-dependent differences during comparisons. We can submit values to be inserted into copies of the secret array and trigger the sorting operation while measuring power consumption. By analyzing the power traces from multiple sorting operations with different input values, attackers can infer which comparison branches are taken during the merge process, gradually revealing the contents and ordering of the original hidden array.

Solution:

Our solution exploits the merge operation by capturing power traces while inserting different values into the array and triggering the sort. We implemented a binary search strategy that compares power consumption patterns between different candidate values to identify boundaries in the secret array. By computing the difference between traces captured when sorting with different inserted values, we can detect when values land in different positions relative to the hidden elements, as this causes distinct comparison patterns during the merge. We maintain a list of discovered values and systematically search the gaps between them, using a correlation threshold to determine when trace differences are significant enough to indicate we have found an actual array element.

Optimization:

The optimizations implemented for Sorters Song 1 and 2 also apply here. Thus, we made sure that we performed our binary search on an ever smaller range and also cached the traces for each value so that it is not measured more than once.

AI/ML Integration:

We tried two learning-based strategies. First, for each array index we captured two anchor traces (value 0 as SMALL, value 65535 as BIG) and trained an `SGDClassifier` on these extremes. During the binary search we captured the midpoint trace, let the classifier decide SMALL/BIG, updated the bounds accordingly, and called `partial_fit` so the model could adapt to trace drift and per-index leakage. This ML-guided proved fast and stable. Second, we experimented with a tiny CNN (PyTorch) that only refined the final, ambiguous segment by re-training on normalized traces from a narrow window around the decision boundary. While the CNN worked, it required more traces and training on-the-fly, so the classical ML version remained the preferred solver.

Mitigation:

Since this challenge falls under the sorting side channel category along with previous challenges, the same mitigations apply here as well.

References

[1] Daniel J. Bernstein. Salsa20 security, 2005.
[2] Intel Corporation. Guidelines for mitigating timing side channels against cryptographic implementations, 2022.
[3] NewAE Technology Inc. Side-Channel analysis tool-chain. kernel description, 2021.
[4] Nathan Jones. Breaking 45rc3edcer3 with an oscilloscope, 2025.
[5] Paul Knutson. Vulnerability analysis of Salsa20: Differential analysis and deep learning analysis of Salsa20, 2020.
[6] Colin O'Flynn. Breaking a password with power analysis attacks. *Circuit Cellar*, 2017.
[7] Fayez Gebali Abdulrahman Alkandari Samer Moein, T. Aaron Gulliver. Hardware attack mitigation techniques analysis. *International Journal on Cryptography and Information Security*, 2017.