CSAW ESC 2025 REPORT

HACKCESS TEAM

BESSON Jérémy
iDeaan
Roanne, France
jeremy.besson@hackcess.org

MBENGUE Guy
gmx.0x1
Roanne, France
guy.mbengue@hackcess.org

MILLOT Elisa
Tenshi.eli
Roanne, France
elisa.millot@hackcess.org

DUMAS Mathieu
Hypno
Roanne, France
mathieu.dumas@hackcess.org

I. Introduction

This electronic document serves as a report on our participation in the CSAW Embedded Security Challenge (ESC) 2025 edition. It includes detailed descriptions of our resolution methods for the challenges we successfully addressed, and our proposed ideas and approaches for tackling the unsolved challenges. The final phase of CSAW ESC is structured into three sets of challenges, with the setup utilizing a "ChipWhisperer Nano" device and provided source codes distributed by the organizing committee.

II. Context

A. Description of the 2nd parts

CSAW 2025 calls for using AI (deep learning and LLMs) to automate hardware attacks and build smart defenses. Challenges center on classic hardware threats such as SCAs and FIAs, our team actively participated in the CSAW 2025 Embedded Security Challenge, focusing on hardware attacks across three distinct sets: **Set 1**, **Set 2**, and **Set 3**. Each set provided progressively complex scenarios that allowed us to experiment with side-channel and fault-injection attack and implement defensive strategies.

B. Setup

For the hardware setup, we simply connected the ChipWhisperer Nano to the host PC via USB and executed the provided Jupyter notebook Setup_Generic.ipynb (from the CSAW challenge repo). The notebook automatically initializes the scope and target, verifies the USB connection, applies the default capture and trigger settings, and runs a quick test capture allowing us to start trace acquisition immediately without manual low-level configuration

Figure 1: ChipWhisperer Nano

For each challenge (Set1, Set2, Set3) we followed the same procedure: connect the board and run the challenge-specific notebook (for example, challenges/set1/gatekeeper.ipynb), which applies the challenge's wiring and capture parameters, performs automated sequence runs, and saves labeled traces and metadata.

III. Resolving challenges

We are going to explain how we resolve challenges set by set.

A. *Set 1*

This set is composed of 3 challenges: Gatekeeper (1 & 2), Sorters Song (1 & 2) and Critical Calculation.

GateKeeper1: flag = gk1{l0g1npwn}

We tackled **GateKeeper** 1 by exploiting a timing side-channel in the password verification routine. The binary compares the input password byte-by-byte against the correct flag, formatted as gk1{xxxxxxxx}, and returns a success byte (0x01) only when the entire string matches. By sending crafted inputs with correct prefixes and padding the unknown suffix with dummy characters (like !), we measured the response time for each guess at the current position. Correct characters cause the comparison loop to run one extra iteration before failing, resulting in consistently longer execution times; typically, a few microseconds more than incorrect ones.

```
Testing: gk1{l0g1npwl}.6f
Testing: gk1{l0g1npwm}.6f
Testing: gk1{l0g1npwn}
SUCCESS! Password found: gk1{l0g1npwnn}
```

Figure 2: Flag gatekeper1

To ensure reliable measurements, we reset the "*ChipWhisperer Nano*" target before every trial, ran 20 repetitions per character, and used the median time to filter out noise. We iterated through lowercase letters and digits (a-z0-9) for each of the *8 unknown positions*, appending the longest-timing character to our recovered flag. This systematic approach recovered the full password gk1{l0g1npwn} without needing fault injection, as shown in the console output where the final success message appears after brute-forcing the last character.

GateKeeper2: No flag

We attempted to solve GateKeeper 2 using a timing side-channel attack on its password verification. The binary compares input byte-by-byte to the correct flag gk2{xxxxxxxxxxxx} and returns success only when all bytes matches. By sending inputs with the known prefix, a guessed character, and dummy padding, we aimed to detect longer response times for correct guesses due to an extra comparison iteration and the delay loop (2500 – i*125 cycles). Using a ChipWhisperer Nano, we reset the target before each trial, ran three repetitions per character, and used median timings to reduce noise while testing digits and lowercase letters for all 12 unknown positions. In theory, this would recover the flag without fault injection, with glitching as a fallback. However, we failed to extract it; likely because timing differences were too small or communication noise masked the signal.

Countermeasure: A simple countermeasure would be to implement a constant time comparison, ensuring the verification routine always executes the same number of operations regardless of where the mismatch occurs.

SorterSong: no flag

We attempted to recover the flag but were not successful, and here is our deduction on how to do it:

Attack type: Timing attack

The *Sorters Song* challenge exploits a timing leakage in an insertion-sort routine. The `'c'` (for `arr1`) and `'d'` (for `arr2`) commands sort modified arrays, and the execution time depends on the number of inversions or shifts required to insert a new element. This behavior allows an attacker to infer the relative position of elements in the sorted array.

B. *Set 2*

This set is composed of 3 challenges: Dark Gatekeeper, Ghost Blood and HyperSpace Jump Drive.

Dark GateKeeper: flag = **ESC{J0lt_Th3_G473}**

For the ***Dark Gatekeeper*** challenge, we performed a straightforward, iterative power-analysis attack to recover a 12-character password. The script tests each password position by sending chosen inputs to the device over SimpleSerial, capturing power traces with the **ChipWhisperer Nano**, and comparing each trace to a reference trace. The selection metric is the sum of absolute differences between traces and the character that maximizes this difference is chosen for the current position. We initially tested the alphanumeric set (`0-9, A-Z, a-z`) without success; after reverse-engineering:

```
>> import re
>> import binascii
>>
>> def parse_intel_hex(filename):
>>     data = {}
>>     with open(filename, 'r') as f:
>>         for line in f:
>>             line = line.strip()
>>             if line.startswith(':'):
>>                 length = int(line[1:3], 16)
>>                 address = int(line[3:7], 16)
>>                 type_ = int(line[7:9], 16)
>>                 if type_ == 0:  # data
>>                     hex_data = line[9:9+length*2]
>>                     data[address] = binascii.unhexlify(hex_data)
>>     if not data:
>>         return bytearray()
>>     max_addr = max(data.keys()) + len(data[max(data.keys())])
>>     binary = bytearray(max_addr)
>>     for addr, d in data.items():
>>         binary[addr:addr+len(d)] = d
>>     return binary
>>
>> binary = parse_intel_hex('d:\\CSAW2025\\DarkGatekeeper\\darkGatekeeper-CWNANO.hex')
>> # Find printable strings of 12+ chars
>> strings = []
>> i = 0
>> while i < len(binary):
>>     if 32 <= binary[i] <= 126:
>>         start = i
>>         while i < len(binary) and 32 <= binary[i] <= 126:
>>             i += 1
>>         if i - start >= 12:
>>             strings.append(binary[start:i].decode('ascii'))
>>     else:
>>         i += 1
>>
>> for s in strings[:10]:
>>     print(repr(s))
>>
'2-byexpate k'
'0123456789ABCDEF7N4>qp14c70!'
'Access Denied.....'
```

Figure 3: Flag Dark gatekeeper

We discovered two additional characters (> and !) and added them to the candidate set. Using the expanded alphabet the attack succeeded, and we recovered the flag who was: **ESC{J0lt_Th3_G473}**.

```
Position 6: 'p' (diff: 681.80859375)
Clé actuelle: 7N4>qp
Position 7: '1' (diff: 680.94140625)
Clé actuelle: 7N4>qp1
Position 7: '1' (diff: 680.94140625)
Clé actuelle: 7N4>qp1
Position 8: '4' (diff: 677.46484375)
Clé actuelle: 7N4>qp14
Position 8: '4' (diff: 677.46484375)
Clé actuelle: 7N4>qp14
Position 9: 'c' (diff: 675.95703125)
Clé actuelle: 7N4>qp14c
Position 9: 'c' (diff: 675.95703125)
Clé actuelle: 7N4>qp14c
Position 10: '7' (diff: 674.5)
Clé actuelle: 7N4>qp14c7
Position 10: '7' (diff: 674.5)
Clé actuelle: 7N4>qp14c7
Position 11: '0' (diff: 670.33203125)
Clé actuelle: 7N4>qp14c70
Position 11: '0' (diff: 670.33203125)
Clé actuelle: 7N4>qp14c70
Position 12: '!' (diff: 655.2421875)
Clé actuelle: 7N4>qp14c70!
Mot de passe trouvé: 7N4>qp14c70!
Réponse finale: ESC{J0lt_Th3_G473}
Position 12: '!' (diff: 655.2421875)
Clé actuelle: 7N4>qp14c70!
Mot de passe trouvé: 7N4>qp14c70!
Réponse finale: ESC{J0lt_Th3_G473}
```

Figure 4:Test of keys positions

Countermeasure: The **shuffling** (random reordering of internal operations) would have significantly disrupted our position-by-position attack. By randomly changing the order in which each byte of the password is processed, the power consumption points

associated with a given position no longer consistently appear at the same time in the traces. Concretely, shuffling introduces desynchronization that forces the attacker either to precisely realize every trace or to collect a much larger number of recordings to recover correlations.

HyperSpace Jump Drive: flag = ESC{21hYP35TrEEt}

We tackled the ***HyperspaceJumpDrive*** challenge, which involved performing a **Differential Power Analysis (DPA)** attack to recover a 12-byte secret from a cryptographic device. The challenge provided access to a target device that responded to commands and leaked power consumption traces during internal operations.

We began by sending the `'p'` command with all possible 1-byte inputs (0–255), capturing the corresponding power traces. These traces reflect the internal computation influenced by the secret. Our hypothesis was that the device performed a XOR operation between the input and the secret, and that the resulting **Hamming weight** affected power consumption.

Using this, we divided each trace into 12 segments—one for each byte of the secret—and computed the correlation between the Hamming weights of guessed intermediate values and the actual power traces. For each byte position, we selected the guess with the highest correlation, effectively revealing the secret byte-by-byte.

```
Capturé 256 traces
Octet 0: 200 (corr: 0.878)
Octet 0: 200 (corr: 0.878)
Octet 1: 22 (corr: 0.848)
Octet 1: 22 (corr: 0.848)
Octet 2: 227 (corr: 0.861)
Octet 2: 227 (corr: 0.861)
Octet 3: 165 (corr: 0.846)
Octet 3: 165 (corr: 0.846)
Octet 4: 34 (corr: 0.865)
Octet 4: 34 (corr: 0.865)
Octet 5: 255 (corr: 0.748)
Octet 5: 255 (corr: 0.748)
Octet 6: 239 (corr: 0.264)
Octet 6: 239 (corr: 0.264)
Octet 7: 199 (corr: 0.257)
Octet 7: 199 (corr: 0.257)
Octet 8: 99 (corr: 0.256)
Octet 8: 99 (corr: 0.256)
Octet 9: 98 (corr: 0.245)
Octet 9: 98 (corr: 0.245)
Octet 10: 123 (corr: 0.596)
Octet 10: 123 (corr: 0.596)
Octet 11: 118 (corr: 0.278)
Octets du secret: [200, 22, 227, 165, 34, 255, 239, 199, 99, 98, 123, 118]
...
Octet 11: 118 (corr: 0.278)
Octets du secret: [200, 22, 227, 165, 34, 255, 239, 199, 99, 98, 123, 118]
Entiers secrets: [2783123144, 3354394402, 1987797603]
Flag: ESC{21hYP35TrEEt}
```

Figure 5: Flag HyperSpace Jump Drive

After reconstructing the 12-byte secret into three 32-bit integers **(little-endian format)**, we sent them back to the device using the `'a'` command. The device responded with the flag: flag = ESC{21hYP35TrEEt}

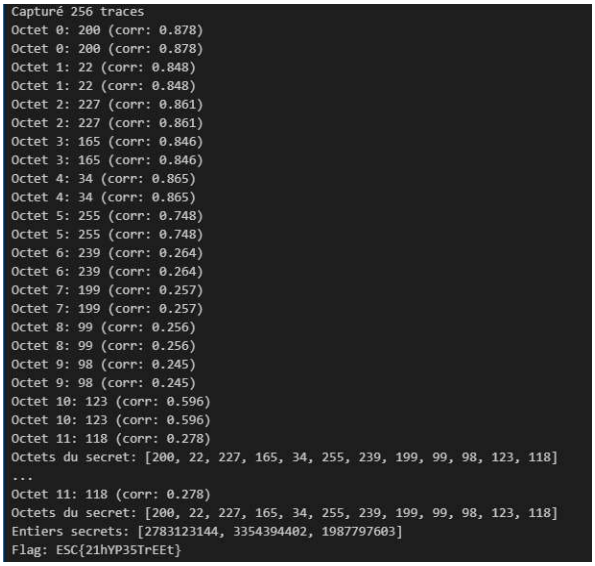

Figure 6: List of positions

C. Set 3

This is the last set of challenges of CSAW ESC 2025

- For the last challenge, we made several attempts to solve it, but we couldn't find any possible solution. We analyzed different approaches, yet none of them led to a successful result.

IV. CONCLUSION

In conclusion, throughout this month of challenges, we have significantly enhanced our skills in cybersecurity and logical thinking. This experience has been a valuable opportunity for growth and learning.