CSAW ESC 2025 Final Report

TheRomanXpl0it - Sapienza

Francesco Vertucci
Sapienza University of Rome
francesco.vertucci22@gmail.com

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

Federico Angelilli
Sapienza University of Rome
fedeangelilli@gmail.com

Lorenzo Colombini
Sapienza University of Rome
lorenzinco23@gmail.com

Emilio Coppa
Luiss University of Rome
ecoppa@luiss.it

***Abstract*—We present our end-to-end exploitation and mitigation study for the CSAW ESC 2025 hardware challenges using the ChipWhisperer Nano platform. Our workflow combines classical physical-attacks (timing/power side channels and fault injection) with targeted automation and machine learning to accelerate data collection, analysis, and verification. To scale experimentation remotely, we built a lightweight module to drive ChipWhisperer over SSH/RPyC from personal laptops via a Raspberry Pi relay, enabling reproducible headless capture, centralized storage, and multi-user collaboration.**

Beyond exploitation, we distill concise, evidence-backed countermeasures and map them to the precise leak points we abused. Our report emphasizes *correctness* (all flags recovered and validated), *AI/ML augmentation* (ML models and LLM-assisted tooling to automate acquisition and post-processing), and *operational efficiency* (reduced queries via side-channel guided search, automated glitch sweeps, and remote orchestration). The result is a practical blueprint for attacking and defending small MCUs under realistic, resource-constrained conditions.

***Index Terms*—Side-channel analysis, fault injection, ChipWhisperer, correlation power analysis, hardware security, machine learning for SCA.**

I. Introduction

Side-channel analysis (SCA) and fault injection attacks (FIA) exploit the physical behavior of embedded systems to extract or alter sensitive information. Even when cryptographic or authentication algorithms are mathematically secure, their physical implementation often leaks unintended information through timing, power consumption, or electromagnetic emissions [1]–[3]. Measuring and correlating such signals can reveal secret keys, passwords, or internal control-flow decisions. Complementary to passive observation, active fault injection manipulates voltage, clock, or electromagnetic conditions to corrupt execution, skip security checks, or induce logic faults that expose hidden data [4], [5].

These techniques have evolved from theoretical models to practical attacks feasible on low-cost platforms. Modern SCA combines precise synchronization, high-resolution data capture, and advanced statistical or learning-based analysis to isolate information-bearing variations across noisy traces [6], [7]. FIA, on the other hand, increasingly relies on automation to find timing offsets that deterministically disturb the target, turning reliability issues into exploitable vulnerabilities [8], [9].

The CSAW ESC 2025 hardware track challenged participants to apply such physical attacks on real embedded targets using the ChipWhisperer Nano board. Each task required recovering a secret flag by leveraging power analysis, timing side channels, or fault injection. Our team approached the competition with two primary objectives. First, to demonstrate how classical side-channel and fault-injection analyses can be improved and scaled through lightweight automation and machine learning — i.e., propose new SCA/FIA strategies using ML, accelerate per-byte/trace decisions with compact models, and build efficient, reusable scripts that make established attacks both faster and more robust. Second, to provide an open, collaborative infrastructure that enables reproducible experimentation and remote hardware access: the `ChipWhispererRemote` Python module (discussed below) exposes the ChipWhisperer API over RPC, supports headless captures via a Raspberry Pi relay, and streamlines the workflow for geographically distributed teams.

Throughout the report, we document how this infrastructure supported the exploitation of several real-world leakages — from correlation-based power analysis and trace classification to timing and fault-induced data corruption — and how the same framework can serve as a foundation for secure firmware testing and educational research.

II. Our Approach

We attempted and solved all three sets of challenges. Table I summarizes the recovered flags, ordered by week.

For this year's challenge, we adopted a fully remote and automated setup to maximize collaboration and reproducibility. The provided ChipWhisperer Nano was connected to a Raspberry Pi 4, which acted as a relay on our internal network. This configuration allowed every team member to perform captures, parameter sweeps, and data downloads from anywhere without direct physical access to the hardware.

Challenge Name	Recovered Flag
Gatekeeper 1	gp1{l0g1npwn}
Gatekeeper 2	gp2{7rU3ncrYkIND}
Critical Calculation	cc1{C0RRUPT3D_C4LCUL4T10N}
Sorter Song 1	ss1{y0u_g0t_it_br0!}
Sorter Song 2	ss2{!AEGILOPS_chimps}
Dark Gatekeeper	ESC{J0lt_Th3_G473}
Hyperspace	ESC{21hYP35TrEEt}
Ghostblood	ESC{Th*t'sT*eSp1*1t!}
AlchemistInfuser	a1c{Wh1teDragonT}
Echoes of Chaos	eoc{th3yreC00ked}

TABLE I
RECOVERED FLAGS FOR ALL CSAW ESC 2025 CHALLENGES.

To achieve this, we developed `ChipWhispererRemote`, a lightweight Python module that provides remote access to the complete ChipWhisperer API through secure SSH and RPyC communication. The module allows seamless invocation of standard ChipWhisperer functions, including programming, arming, triggering, and trace acquisition, while transparently handling data transfer between the remote device and the local analysis environment.

This tool was crucial for coordinating long-running experiments such as power-trace collection and glitch sweeps, where the Raspberry Pi's limited processing power and thermal constraints previously caused frequent interruptions. Offloading the computationally expensive steps (e.g., correlation analysis, CNN inference, or trace alignment) to local machines allowed us to run automated scripts overnight, dramatically improving throughput and experiment consistency.

By prioritizing modularity and ease of use, `ChipWhispererRemote` became a central component of our workflow. It facilitated asynchronous contributions from all team members, streamlined trace management, and reduced the iteration time required to test new ideas or mitigation hypotheses. We expect the framework to remain useful beyond this competition as a general-purpose remote control and automation layer for the ChipWhisperer ecosystem.

An example of how to interact with the ChipWhisperer through our module is shown below:

```python
from remote_cw import remote_cw, RemoteConfig
from helper_cv import setup_cw, cap_pass_trace
from rpyc.utils.classic import obtain

cfg = RemoteConfig(
    host="www.remotecw.example",
    user="pi",
    key_filename="path/to/priv_key",
    port=18812,  # RPyC port
    connect_host="127.0.0.1",
    remote_host="127.0.0.1",
)

with remote_cw(cfg) as cw:
    # Runs on the remote machine (Raspberry Pi)
    scope, target, prog = setup_cw(cw, cw.scope())
    trace_data = cap_pass_trace(...)
    local_data = obtain(trace_data)

    # Runs locally on the analyst's computer
    expensive_computation(local_data)
```

This hybrid approach enabled controlled, reproducible experiments across multiple devices while maintaining full flexibility for data processing and analysis.

III. WEEK 1

In the first week we solved three challenges and developed the remote interaction module.

A. GateKeeper

In this challenge, the password checker exhibited distinct power traces depending on whether the character currently tested was correct. Our first goal was to ensure that the entire password-checking routine could fit within a single acquisition, since the provided ChipWhisperer Nano did not support capturing multiple traces per run or streaming data directly to the Raspberry Pi.

To achieve this, we overclocked the target to reduce the delay between individual character checks, and underclocked the ChipWhisperer to capture fewer samples while maintaining temporal resolution. We then set the capture size to 100,000 samples, which worked reliably even though the official documentation suggests otherwise.

First Approach. The first challenge came pre-solved (the flag was provided within the Jupyter notebook `gk1{l0g1npwn}`), allowing us to use that data leak to train a convolutional neural network to classify whether a single character check was correct or incorrect. We trained the model in this way because we knew that the second challenge would feature shorter delays and a different password length, making approaches based on counting wrong characters ineffective. The task turned out to be straightforward for the network, which achieved 99% test accuracy. The architecture of our convolutional neural network is in Fig 1.

Using this model on the segments of each captured trace corresponding to the password checks, we could accurately determine whether each character was correct or not. By iteratively reconstructing the password one character at a time, we successfully recovered the full 12-character flag in just a few minutes.

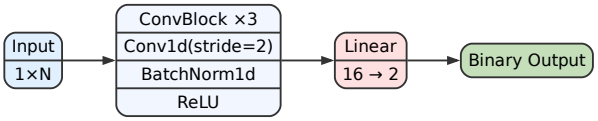

Fig. 1. Architecture of the 1D CNN for trace classification.

Second Approach. Besides the CNN, we also developed a simpler, fully analytical method based on comparing the frequency spectra of traces. A naive sum of absolute differences (SAD) in the time domain failed to reveal meaningful leakage: although correct and incorrect characters triggered visibly different behaviors, the corresponding signals were often misaligned in time due to the variable-length delay loop

in the firmware. These phase shifts caused SAD to fluctuate unpredictably even for identical traces.

Transforming the traces using the discrete Fourier transform (DFT) solved this problem. By comparing the magnitudes of the spectra, we effectively ignored temporal misalignment while capturing the characteristic energy peaks introduced by the delay loop whenever a correct character was processed. This made the difference between correct and incorrect guesses immediately apparent, producing highly stable results across all traces. Despite its simplicity, the DFT-based approach achieved time performance comparable to the CNN while requiring no training data and no manual split of the trace.

Both methods successfully retrieved the other flag: `gk2{7rU3ncrYkIND}`.

B. Critical calculation

The challenge runs a tiny built-in diagnostic: when you send the simpleserial command "`d`" the firmware raises a trigger, executes a deterministic double loop that increments a counter ($100 \times 40 \times 2 = 8000$), then lowers the trigger and checks the counter. If the counter equals 8000 the device replies with a padded "`DIAGNOSTIC_OK...`" message; if the loop is disturbed (iteration skipped, counter corrupted, early exit, etc.) the firmware takes the failure branch and leaks the secret flag via simpleserial_put.

We solved it by fault-injecting that loop. Using our remote setup we boosted the target clock and tuned the scope, then performed a systematic sweep over glitch timing (ext_offset) and width (repeat)[1] while issuing the "`d`" command and reading the response. Most attempts returned with positive response or caused resets, but at a specific timing —width = 2 and offset = 16— the glitch corrupted the loop (so cnt != 8000) and the device emitted the flag `cc1{C0RRUPT3D_C4LCUL4T10N}`.

Mitigations. A simple and effective mitigation for this particular diagnostic is redundancy: run the loop twice and compare results, or perform a consistency check with an independent counter/CRC. In addition, deploying glitch sensors or active voltage monitoring (which reset or halt execution on abnormal supply disturbances) would prevent simple MOSFET-based shorting attacks from causing information leakage.

C. Sorter Songs

This challenge exposes two secret arrays that are randomly generated and sorted at boot: a 15-element 8-bit array (verified by command 'a') and a 15-element 16-bit array (verified by command 'b'). The firmware also implements a "prepare" command ('p') that lets the user prepend a chosen value to a partial copy of one of the secret arrays; the device then sorts this modified buffer using commands 'c' (for 8-bit) or 'd' (for 16-bit). Because the sorting operation's timing and power profile change depending on whether the inserted element is smaller or larger than the next element in the hidden array, each capture effectively leaks one comparison result — turning the sorting process into a timing/side-channel oracle. By exploiting these subtle differences, it becomes possible to reconstruct the entire secret array element by element.

We solved the first challenge (the 8-bit array) by scripting the ChipWhisperer remotely, capturing two reference traces for adjacent guesses, and then sweeping through possible byte values. For each position, we used 'p' to build an array starting with our guess and then triggered a sort with 'c', comparing the resulting trace with the references. When the trace diverged significantly, it indicated that we had just crossed the correct boundary, allowing us to fix the previous byte as correct and move to the next position. Repeating this process recovered all 15 elements, and submitting them with 'a' revealed the flag `ss1{y0u_g0t_it_br0!}`.

For the second challenge (the 16-bit array), we employed the same idea but replaced the linear scan with a binary search over the 16-bit range, which reduced the number of traces required per element. Each comparison indicated whether our guess should be higher or lower, allowing us to converge quickly to the true value. After reconstructing all 15 elements, sending them with command 'b' produced the second flag `ss2{!AEGILOPS_chimps}`.

IV. Week 2

A. Dark Gatekeeper

This challenge implements a 12-byte password verification routine invoked with command `'a'`. The firmware compares each byte of the input with a hidden `master_key` inside a trigger-delimited region and finally returns either "`Access Denied...`" or the flag. Although the loop does not break early on mismatches, each comparison introduces a subtle variation in power consumption, leaking information about which bytes match.

First Approach. We exploited this leakage using a simple side-channel analysis (SCA) based on the *maximum sum of absolute differences* between each captured trace and a reference. For each position, we fixed the known prefix, iterated one character across printable bytes, and scored traces against the reference window. The correct byte consistently produced the highest deviation, as matching comparisons shift the signal amplitude after a few dozen samples. Figure 2 shows an example overlay between the reference trace and different guesses, where the correct byte exhibits a clear offset. Repeating this process for all positions revealed the full password `7N4>qp14c70!`, which successfully unlocked the flag `ESC{J0lt_Th3_G473}`.

Second Approach. Separately, we attempted fault injection to force the device into a state that would print the flag. Instead of toggling the final `access_granted` bit, one particular glitch produced an unexpected partial memory leak: with `repeat=4` and `ext_offset=38` the device printed the 12-byte password itself (followed by padding/adjacent bytes): `b'7N4>qp14c70!\x00\x00\x00\x00Ac'`. This behavior is consistent with a transient memory or control-flow corruption (for example, a corrupted return address, stack

[1]Official reference: https://chipwhisperer.readthedocs.io/en/v6.0.0b/Capture/ChipWhisperer-Nano.html#glitch

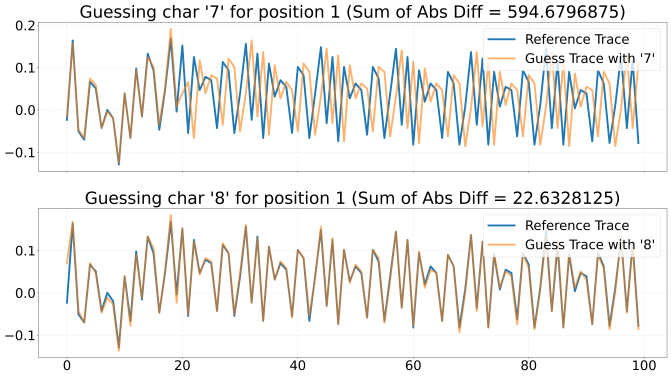

Fig. 2. Overlayed traces for one password position: the correct guess shows a consistent post-comparison offset.

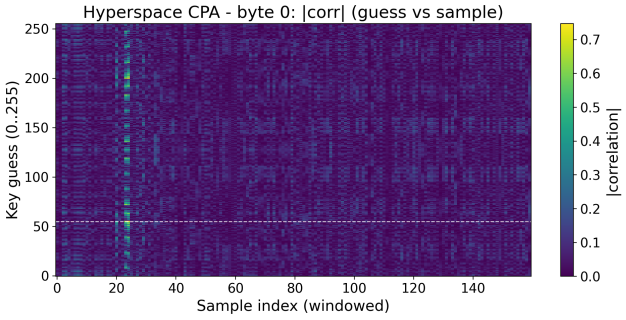

Fig. 3. CPA correlation heatmap for one key byte (guesses 0–255 on the vertical axis, sample index on the horizontal axis). Bright vertical structure indicates the sample region where certain guesses correlate strongly with the measured power; the brightest row is the winning key guess.

pointer, or a write/read to an in-RAM buffer) that caused the program to copy or transmit secret data accidentally.

Mitigations. The SCA leak is closed by making comparisons constant-time and data-oblivious (e.g., accumulate differences and branch only once after the full loop). The FIA result highlights the need for redundancy and memory-safety practices: avoid printing sensitive buffers on error paths, add redundant consistency checks for authentication results, and consider simple hardware sensors to detect/abort on abnormal supply events.

B. Hyperspace Jump Drive

This challenge hides a 12-byte "ignition sequence" and exposes two useful commands. The `'a'` (arm) command compares a 12-byte input against the hidden sequence and returns either a fake "FailureToLaunch!" message or — on a specific branch — the real flag. The `'p'` (polarity) command XORs a one-byte mask with each secret byte inside a trigger-delimited loop; that loop generates regular, byte-aligned power activity that is ideal for side-channel analysis.

We performed a classic CPA (Correlation Power Analysis) on the `'p'` handler. We captured 256 traces with inputs 0..255; each trace contains twelve distinct windows (one per secret byte). For every window and every key guess (0..255) we predicted the Hamming weight and computed the sample-wise correlation with the measured traces. The correct guess produces a pronounced correlation peak in the time window corresponding to that byte.

Applying this procedure across all twelve windows recovered the full ignition sequence: `37 45 4c 16 6e 1c 77 2d 5b 5a 22 7b`. Submitting those bytes to `'a'` triggered the leak branch and returned the flag: `ESC{21hYP35TrEEt}`.

C. GhostBlood

This challenge implements a custom stream cipher, based on ROTL operations arranged in a ChaCha-like quarter-round structure. It has an internal state of 16 words, of which 8 words are the unknown key.

With the provided `shift` function, we can run the cipher with custom shifts (four 8-bit values) and a threshold. Using the ChipWhisperer we can capture a power trace when the count of ROTL operations reaches the threshold.

This allows us to probe each individual ROTL behavior. Using a simple correlation analysis between traces (shown in Fig 4), we can distinguish between the branch taken by the if statement, which depends on the condition $a \geq 2^{16-b}$. By varying only the last shift for the column (e.g. `shift[2]` for the third ROTL operation), we can find the minimum shift that triggers the true branch. This is a reliable way to constrain the ROTL input to a range of $[2^{16-b}, 2^{16-b+1})$.

For each threshold value, we can provide many combinations of shifts, excluding the shift at index `(thresh-1)%4`, which will be linearly scanned to find the minimum branching shift. To keep nonlinearity down, we focused only on the first four quarter round operations. With enough samples, we can recover information for all bits in the key.

Using our Python library, we collected a large sample set. Then, we recreated the cipher algorithm with Z3 and added all the constraints to symbolic 16-bit bit-vectors.

Solving the system yielded a unique key assignment: `[0xea96, 0xf735, 0x95b5, 0xba52, 0xd896, 0x1a96, 0xb689, 0x5f9]`.

We submitted it to the target program via the decrypt function and obtained the flag `ESC{Th*t'sT*eSp1*1t!}`.

Mitigations. This challenge was a perfect example of how seemingly sound cryptographic algorithms can be undermined by physical attacks. As usual, we want to minimize differences between different branches, so that an attacker will have an harder time identifying discrepancies between traces.

A mitigation is rewriting the ROTL function to be completely branchless. By removing the if statement and calculating always $(a \ll b)|(a \gg (16-b))$, we obtain a mathematically equivalent result with no information leakage.

V. Week 3

A. AlchemistInfuser

This firmware implements a two-stage transformation around a 16-byte secret key. The device exposes three simpleserial commands: `'e'` (encrypt), which first performs a

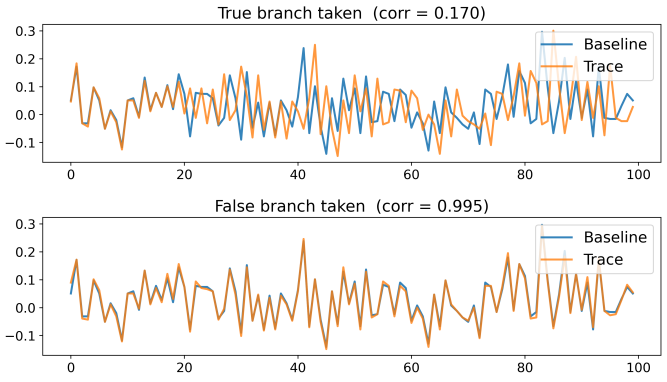

Fig. 4. The baseline trace has $b = 0$, so that $a \geq 2^{16}$ is always false.

trigger-delimited bytewise XOR of the input with the key (with a small software delay between bytes) and then runs an in-place XXTEA encryption; `'d'` (decrypt), which runs the inverse operations; and `'c'` (verify), which returns a dummy string unless the supplied 16 bytes exactly match the internal key, in which case it returns the flag.

Crucially, the initial XOR loop runs inside a timed region and produces very regular, byte-aligned power activity — a classic side-channel leakage source. The subsequent XXTEA routine mixes words, producing a second, distinguishable leakage region later in the capture. Because of this two-stage structure we obtain complementary leakage: one window that reveals information about individual key bytes directly via the XOR, and another window that reflects the cipher's mixing (useful to disambiguate remaining uncertainty).

Our exploit is a pragmatic CPA-driven workflow adapted to that structure:

1) Capture several hundred traces of `'e'` using random 8-byte inputs, producing long recordings that contain both the pre-XXTEA and post-XXTEA leakage windows.
2) Segment the traces into per-byte windows corresponding to the initial XOR loop (eight windows) and the later region (another eight windows). Compute per-window means and standard deviations.
3) For each window and each candidate byte (0..255) compute the Hamming-weight prediction and the sample-wise correlation with the recorded traces (standard CPA). Record the peak absolute correlation per candidate.
4) Take the top candidate per byte and also a small shortlist of near-ties (candidates within ≈95% of the peak) to account for statistical ambiguity.
5) Use the second leakage region and the partial key information from phase one to further refine guesses for the remaining bytes, again via CPA over the appropriate modeled intermediate.
6) Finally, submit combinations from the Cartesian product of the per-byte candidate lists to the `'c'` (verify) command until the device returns a non-dummy response.

This approach is efficient and robust: it avoids exhaustive search over the full 128-bit key while tolerating small statistical noise. Applying it recovered the full 16-byte key (printed by our solver) and produced the flag: `a1c{Wh1teDragonT}`.

B. *Echoes of Chaos*

This challenge extends the "Sorter Song" concept into a more complex setting: instead of simple insertion or bubble sorts, the firmware uses a recursive merge sort routine called `clydeSort`, which merges subarrays using nested loops and data-dependent delays. Each merge step compares two half-arrays element by element, applying small software delays only in one branch. This creates distinct, measurable variations in power consumption and timing — effectively leaking information about the structure and relative order of the secret array elements.

At startup, the device initializes a hidden 15-element array of 16-bit integers. When queried with the command `'p'`, it builds a modified array by inserting one guessed element at the start of a shifted copy of the secret array and then performs a full merge sort inside a trigger-delimited region. The resulting trace depends on whether the guessed element belongs before or after the next unseen value in the hidden array. The command `'a'` verifies the guessed array and returns the flag if it matches the original.

We exploited this by performing a side-channel guided binary search on each position in reverse order. For every element, we captured two reference traces with nearby guesses to establish a baseline difference, then iteratively adjusted the guess until the measured trace difference exceeded that reference threshold — a clear sign that the correct insertion boundary had been crossed. Each byte-pair (16-bit value) was thus located with minimal traces, converging logarithmically rather than linearly.

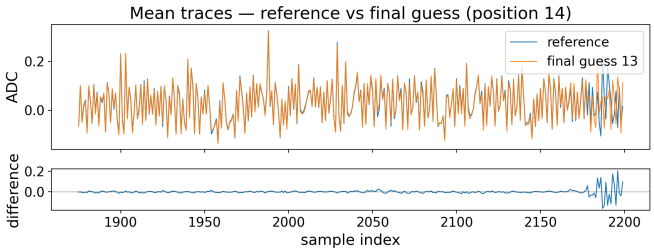

Fig. 5. Overlay of mean traces for a single guessed insertion at position X (top) and the difference trace (bottom). The highlighted region (samples ...) shows the merge-step leakage used by the attack.

After recovering all 15 values, we sorted the reconstructed array and sent it back to the device with `'a'`, which verified it as correct and returned the flag: `eoc{th3yreC00ked}`.

VI. Mitigations and Secure Implementations

All the challenges we attacked share the same root weakness: the implementation lets an internal decision (comparison result, array position, loop progress, key byte) influence something we can measure from the outside — timing, power, or the device's reaction to a fault. When the device is accessible to an active adversary (exactly the CSAW ESC setup), the only

robust defence is to remove, randomize, or continuously check these correlations. This is the same observation that underlies most side-channel guidance from NIST and national agencies [10]–[12].

A. Constant-Time and Data-Oblivious Code

Several of our solves depended on tiny data-dependent branches (e.g., "insert from left array" vs "take from right array" in the merge, or "this byte matches" vs "does not match" in password checks). The simplest mitigation is to make these routines *constant time*:

- do not break early on the first mismatch (compare the whole array);
- keep loop bounds independent of secret data;
- avoid branches that execute extra instructions only on one side.

For a password check, this means: walk every byte, OR the differences into a single accumulator, and branch once at the end. For array checks, this means: verify all 15 elements and only then decide whether to return the flag. These patterns are exactly the ones suggested in practice-oriented SCA write-ups and in FIPS-style guidance on leakage-resistant implementations [11], [12].

B. Masking, Hiding, and Randomization

When the code *must* process a secret (key bytes, hidden arrays, authentication strings), it can be protected with classical side-channel countermeasures:

- **Masking**: split the sensitive value into random shares and process the shares instead of the raw value, so that a single trace does not directly leak the secret. Masking, even at low order, is a standard defence in the SCA literature [13].
- **Hiding**: add noise or random delays, or randomize the processing order (e.g., shuffle merge subcalls, or shuffle the order of bytes being compared). This does not remove leakage but makes alignment harder — which is exactly what we exploited when we aligned traces by trigger.
- **Domain separation**: keep sensitive routines in a dedicated, non-interruptible region, so an attacker cannot isolate "just the compare" and correlate it.

Even on tiny boards, lightweight masking/hiding is practical if it is planned from the beginning rather than bolted on later [12], [13].

C. Software Fault-Injection Countermeasures

Our "Critical Calculation" glitch worked because the firmware checked the loop counter only once and trusted it. A basic way to harden this is to add *redundancy*, as suggested in experimental FI evaluations [14]:

- compute the loop twice and compare the results;
- check the condition both in positive and in negative form and ensure the two results are consistent;
- if an inconsistency is detected, enter a safe state and never output the flag.

Compiler and code-layout awareness also matters: on small MCUs a single well-timed glitch can skip two adjacent instructions if the compiler packed them in the same fetch word. Spreading critical checks, or using diversified encodings, makes this kind of single-glitch attack harder [15].

D. Hardware-Assisted Protections

Firmware alone cannot fully protect a device that the attacker can touch. Practical products therefore add:

- **Glitch / clock / voltage sensors** that reset or discard a computation if the clock or supply looks abnormal;
- **Active shields** or external monitors to detect probing or EM injection;
- **Secure update / secure boot** so that a vulnerable constant-time routine can be replaced later.

Surveys on microcontroller hardening point out that many devices ship with such monitors disabled for cost or power reasons, which leaves them open to exactly the kind of clock/voltage attacks we used [12], [15].

E. Testing and Certification

Finally, good defences must be *measured*. A simple TVLA-style "fixed vs random" leakage test would have immediately shown the data-dependent merge branches and the data-dependent compares we abused. Likewise, running an automated fault-injection campaign (even just sweeping clock offsets) during development would have exposed the single-point-of-failure diagnostic check. Adding these tests to CI/CD for embedded firmware is realistic: traces can be captured headless (exactly like we did, via the Raspberry Pi), and regressions are easy to detect. This mirrors the recommendation in industrial/agency reports on MCU attack surfaces [12].

VII. Conclusion

Throughout the CSAW ESC 2025 hardware challenges, we explored a broad range of physical attack vectors on embedded systems — from side-channel analysis and correlation-based key recovery to precise fault injection and signal pattern recognition. Each challenge emphasized how even simple hardware designs can unintentionally expose sensitive information through timing, power, or electromagnetic emissions. By combining rigorous data collection, statistical analysis, and automation, we demonstrated how such vulnerabilities can be systematically exploited without relying on firmware disassembly or direct software introspection.

Working with the ChipWhisperer Nano platform provided a concrete understanding of the relationship between hardware behavior and security, highlighting the importance of physical-layer protections such as noise injection, randomization, and fault countermeasures. Equally important was the collaboration and remote workflow we built, which allowed our team to share resources, scripts, and captured data efficiently while working across different environments.

Overall, this competition reaffirmed that true hardware security requires a holistic approach: algorithms must be paired with robust implementation defenses, and developers must

assume that adversaries can both observe and perturb their systems. Our participation strengthened our expertise in practical hardware exploitation, signal analysis, and coordinated teamwork — experience that will continue to guide our future work in embedded and system-level security.

REFERENCES

[1] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Annual international cryptology conference*. Springer, 1999, pp. 388–397.

[2] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Annual international cryptology conference*. Springer, 1996, pp. 104–113.

[3] S. Mangard, "Power analysis attacks: Revealing the secrets of smart cards," 2007.

[4] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1997, pp. 37–51.

[5] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[6] L. Lerman, G. Bontempi, and O. Markowitch, "A machine learning approach against a masked aes: Reaching the limit of side-channel attacks with a learning model," *Journal of Cryptographic Engineering*, vol. 5, no. 2, pp. 123–139, 2015.

[7] H. Kim, S. Lim, Y. Kang, W. Kim, D. Kim, S. Yoon, and H. Seo, "Deep-learning-based cryptanalysis of lightweight block ciphers," *Entropy*, vol. 25, no. 7, p. 986, 2023.

[8] A. Gangolli, Q. H. Mahmoud, and A. Azim, "A systematic review of fault injection attacks on iot systems," *Electronics*, vol. 11, no. 13, p. 2023, 2022.

[9] M. Dumont, M. Lisart, and P. Maurine, "Electromagnetic fault injection: How faults occur," in *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2019, pp. 9–16.

[10] Y. B. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and impacts on cryptographic module security testing," in *NIST Physical Security Testing Workshop*, 2005. [Online]. Available: https://csrc.nist.gov/csrc/media/events/physical-security-testing-workshop/documents/papers/physecpaper19.pdf

[11] National Institute of Standards and Technology, "Fips 140-3: Security requirements for cryptographic modules," https://doi.org/10.6028/NIST.FIPS.140-3, 2019.

[12] Federal Office for Information Security (BSI), "Study on hardware attacks against microcontrollers," https://www.bsi.bund.de/EN/Service-Navi/Publications/Studies/Hardware-Attacks-Microcontrollers/hardware-attacks-microcontrollers_node.html, 2023.

[13] G. Agosta, A. Barenghi, and S. Pelosi, "Securing cryptographic embedded software against side-channel attacks: the meet approach," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 8, 2015. [Online]. Available: https://re.public.polimi

[14] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Experimental evaluation of two software countermeasures against fault injection attacks," in *FDTC*, 2014. [Online]. Available: https://arxiv.org/pdf/1407.6019.pdf

[15] T. Trouchkine, A. Heuser, J.-M. Bruel, and P. Maistri, "Electromagnetic fault injection against a system-on-chip, toward new micro-architectural fault models," in *FDTC*, 2019. [Online]. Available: https://doi.org/10.1109/FDTC.2019.00013