The Uncs of Th3Os: Deep Dive into CSAW ESC 2025

Ilias Fiotakis, Christos Papadopoulos, Meletis Michail, Vasileios Ananiadis

Gatekeeper 1 & 2 (100 pts & 150 pts)

Algorithm: Password verification.
Attack: Timing. (host-side timing due to CW Nano limitations)
Key idea: Measure per-byte latency differences caused by early-exit comparisons to recover each character.
Detailed explanation: We can call a `verify()` function on the target board that checks each byte of the password we send. First, we send a known-wrong character to build a timing **template** — i.e., measure how long the board takes to execute the check for an incorrect byte. Then we brute-force possible characters one by one and record their timings. If a candidate's timing shows a significant deviation (large de-correlation) from the wrong-character template, we infer that it's the correct byte.

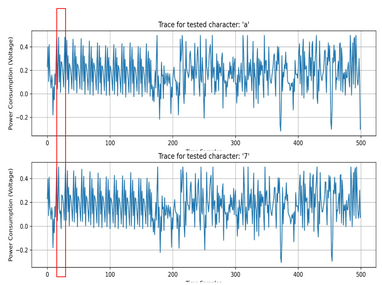
gk1{……..} -> 1ms
gk1{a……} -> 1ms
gk1{b……} -> 1ms
gk1{l……} -> 5 ms (hit)
gk1{la……} -> 6ms
gk1{lb……} -> 6ms
gk1{lo……} -> 10 ms (hit)

gk2{……..} -> 1ms
gk2{a……} -> 1ms
gk2{b……} -> 1ms
gk2{7……} -> 5ms (hit)
gk2{7a……} -> 6ms
gk2{7b……} -> 6ms
gk2{7r……} -> 10ms (hit)

Result: Recovered both flags;
AI/ML: Used k-means on timings to classify fast vs slow guesses
Queries: 142 / 376 (threshold) or ~2881 / 14k (ML path).
Mitigation: Constant-time, branchless comparison for all bytes; equalize instruction count.

Dark Gatekeeper (200 pts)

Algorithm: Password verification. (no deliberate stalls)
Attack: SPA; correlation against a template.
Key idea: Correct byte flips one iteration from mismatch→match, decorrelating the trace from a "wrong-byte" template.
Detailed explanation: We can call a function called `verify()` in the target board that checks if each byte of the password we send is correct. First we test a character that we know to be wrong and we use it for a template trace like the first Gatekeeper challenges. After that we start brute forcing all the possible characters and get their power trace. If the trace that we get has a big de-correlation compared to the template then we know we found the correct character.



On the left, we can see a key difference between characters 'a' and '7'. A timing shift towards the right of the graph. We can detect this difference with np.corrcoef().

Result: Full key via correlation drop (~0.7 threshold).
AI/ML: anomaly detection (PCA + SVM) to detect the correct trace as an outlier.
Queries: 291 (threshold) or ~1610 (ML).
Mitigation: Constant-time, memcmp-style accumulator with masking/noise to flatten power profile.

Critical Calculation (100 pts)

Algorithm: Counter computation with nested loops.
Attack: Voltage glitch (offset × repeats search).
Key idea: Fault the loop to avoid reaching the expected terminal count (8000) and trigger success path.
Detailed explanation: Scan (glitch_offset, repeats) space; adapt with a simple classifier to focus attempts where faults cluster. The above values were configured like so:
scope.glitch.ext_offset = offset
scope.glitch.repeat = repeat
The results were dependent on the ChipWhisperer board we used, and so we tried to deduct the valid ranges for a solver that worked on all the boards.

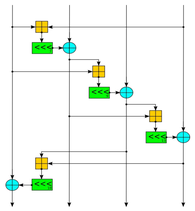
Result: Consistent faults across devices in narrowed bands (e.g., offsets ~1400–1800, repeats ~9–12).
AI/ML: Used MLP to learn high-success glitch parameters.
Queries: ~430 (average) or ~1822 (ML).
Mitigation: Redundant computation and consistency checks; temporal diversity; brown-out detection.

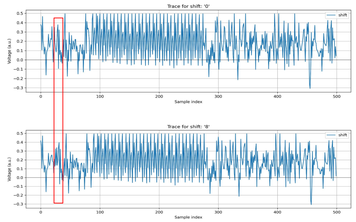
Ghost Blood (200 pts)

Algorithm: Salsa20-style ARX. (modified, 16-bit)
Attack: SPA/DPA via conditional in ROTL. (data-dependent branch)
Key idea: Branch taken vs not taken shifts trace alignment; by probing with selected rotation constants, infer constraints on keywords.
Detailed explanation: This variation of the Salsa20 stream cipher takes a 16 bytes secret key and expands it into 32 bytes. After that multiple operations are performed on the key, specifically Additions, Rotations and XORs.

On the right we can see the ROTL function where we can control the shifts (green) and take their traces.

Our attack starts by making the first shift of the first round zero, and taking its power trace. We use this as a reference point because we know the shift will not overflow the current value. Thus, we know the branch we have taken.





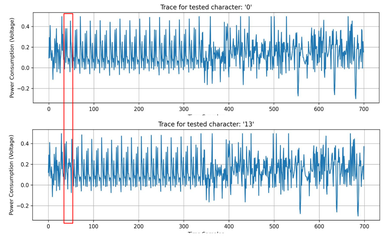
If we try all the possible shifts we will see that at some point the power trace for '8' will be different. This means that we are in the second branch of our if else statement and the byte has overflown.

We can try different shifts and use the power traces to build an oracle that we then plug into z3 to make enough constraints to find us the flag.

Result: Flag recovered with ~30 distinct shift sets (key-dependent).
AI/ML: Tried to use ml to predict the shifts.
Queries: 496.
Mitigation: Constant-time rotate: always compute (a << b) | (a >> (16-b)) without branching.

Sorter Song 1 & 2 (150 pts & 200 pts)

Algorithm: Insertion sort (data-dependent control flow)
Attack: SPA/DPA on comparisons while inserting chosen values
Key idea: Insert test elements; trace differences reveal when comparisons shift—thus recovering the secret array.
Detailed explanation: The board has a secret list that when we connect to the board it copies to another list called data. We can manipulate that data list. More specifically we can do 3 operations. First we can sort the list. Second by calling get_pt we can switch the first index of that array with an arbitrary controllable number. Lastly we can reset the list, essentially copy the original list to the data list. Since we know that the first number of the list is positive, we can call get_pt with zero skips and the number zero as arguments. We can then call the sorting function and get the powertrace. This power trace will be our reference base since we know that sort did not do anything to the array. We can then try all 256 numbers and if we find one that the de-correlation is big from our reference power trace, we know we found the secret number.

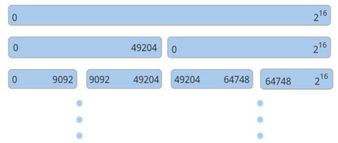


Instead of searching all the numbers we used binary search. We captured three power traces and compared deviations. Depending on the comparison, we repeated on the narrowed range until we found the correct number, reusing previously cached traces.

Result: Recovered arrays; ML logistic regression stabilized noisy branch decisions.
AI/ML: Used Logistic regression on trace features to choose left/right branches.
Queries: 259 / 517 (both)
Mitigation: Add random delays/dummy ops or make comparisons data-independent.

Echoes of Chaos (200 pts)

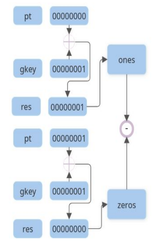
Algorithm: Merge sort (data-dependent comparisons)
Attack: SPA/DPA on merge decisions.
Key idea: Binary search over candidate values; trace differences show on which side of hidden elements the probe lands.
Detailed explanation: Like the Sorter Song challenge here we also have the same functions. The key difference is that for this challenge the sorting algorithm is merge sort. Because of this, we will follow a similar attack path. Instead of doing a simple binary search on each position we will do a recursive binary search on the whole key space. The value checking part of the algorithm remains intact, but instead of doing whole skips we split the search area into [0, found_value) and (found_value, 2^16]. We will recursively apply binary search to those ranges, and we will repeat this process until all secret elements are found.



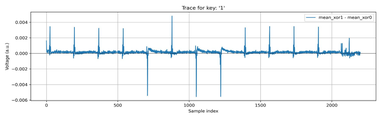
Result: Full array with stable, low query count; caching avoids redundant captures.
AI/ML: Used SGD-based classifier to decide comparisons from power traces.
Queries: ~271
Mitigation: Data-independent compare/move or randomized dummy operations.
evolution)

Hyperspace Jump Drive (200 pts)

Algorithm: XOR against secret sequence
Attack: DPA on per-byte XOR inside trigger window
Key idea: Group traces by hypothetical bit of (mask ⊕ secret) and take mean differences to expose LSB→MSB per byte.
Detailed explanation: In this challenge we have the ability to ⊕ our plaintext with a mask. This mask is secret to us and we need to find a way to retrieve it. Since the source code has convenient nop between each ⊕ operation it's easy to determine the critical areas. To analyze this attack lets simplify the plaintext and the key to one byte. As seen on the right we will send a null byte to get encrypted, get the power trace and we will guess (gkey) that the lsb of the key is 1. We will perform plaintext ⊕ gkey, and observe the result. If the result is 1 we will put the power trace into a list called ones. If the result is 0 then we will append the power trace into the list called zeros. We will repeat this process for some plaintexts.



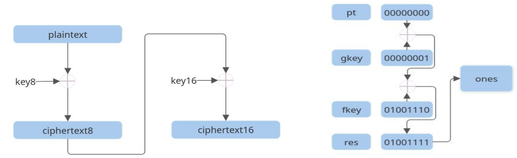
Then we will take the average(ones) - average(zeros) and plot the result. Our concept is to perform a DPA attack similar to the one for AES. If the spike is positive then we now that the lsb of that byte is 1. If the spike is negative then we know that the lsb of that byte is 0. We will repeat the process for every bit of every byte and recover the secret mask.



Result: Reconstructed 12-byte sequence.
AI/ML: Used a classifier to separate real leakage peaks from noise.
Queries: ~1280 (stable)
Mitigation: First-order Boolean masking of intermediates; randomized operation timing.

Alchemist Infuser (200 pts)

Algorithm: XXTEA; leakage in pre-XOR of key bytes
Attack: DPA on XOR step during encryption
Key idea: Recover first 8 key bytes at fixed XOR offsets; use them to align and extract the remaining 8 with tailored DPA.
Detailed explanation: For the first part of the attack (the first 8 bytes of the key) we will follow the methodology presented on Hyper. For the second part, we have two options. Either send the same plaintexts but ⊕ with the first 8 bytes of the key, get new traces, and analyze the new traces (the offsets might change) or create an algorithm to perform DPA on the existing traces taking into consideration the first half of the key. We chose the second due to optimization gains as this approach reduces the queries for the second round in half. To perform it we modified the algorithm of Hyper. Instead of talking the result between the plaintext ⊕ key, we got the result of the plaintext ⊕ gkey ⊕ found_key. The attack was exactly the same, but the offsets were dynamic so it wasn't easy to automate. This is why the attack was performed manually.



Result: Majority of key bytes recovered; manual bit readout confirmed feasibility.
AI/ML: ML model on differential traces to classify DPA bit signs at XOR points for key bytes.
Queries: ~1280
Mitigation: Boolean masking of key/plaintext prior to XOR; randomized schedule.