CSAW ESC 2021 Final Paper: Team Rackets for Georgia Tech VIP

Spencer Hua (Student)
Vertically Integrated Projects
Georgia Institute of Technology
spencerhua@gatech.edu

Ammar Ratnani (Student)
Vertically Integrated Projects
Georgia Institute of Technology
aratnani7@gatech.edu

Suhani Madarapu (Student)
Vertically Integrated Projects
Georgia Institute of Technology
smadarapu3@gatech.edu

Zelda Lipschutz (Student)
Vertically Integrated Projects
Georgia Institute of Technology
hlipschutz3@gatech.edu

Allen Stewart (Advisor)
Vertically Integrated Projects
Georgia Institute of Technology
allen.stewart@gtri.gatech.edu

***Abstract*—The authors participated in CSAW ESC 2021, conducting many side-channel attacks (SCAs) and fault-injection attacks (FIA) on the presented challenges. This paper serves to debrief those challenges, debriefing the team's analyses, though processes, and solution attempts.**

***Index Terms*—Cybersecurity, Embedded Systems, Hardware Exploitation, Side-Channel Attacks, Fault-Injection Attacks**

I. Introduction

Hardware exploitation is a powerful tool, allowing attackers to bypass higher levels of abstraction entirely and compromise the underlying hardware a program runs on. Two general techniques of hardware exploitation are side-channel attacks (SCAs) and fault-injection attacks (FIAs). SCAs observe a processor's characteristics, like power draw or cycles taken, to reverse-engineer the code running inside it. On the other hand, FIAs manipulate the environment a processor is in to elicit unintended behavior by introducing hardware faults, typically in the form of voltage glitching or clock glitching.

The authors of this paper participated in CSAW ESC 2021, a competition which focused on SCAs and FIAs. As participants, the team created and mounted many such exploits. This paper explains the authors' approach to overcoming the challenges, considering exploit analyses, exploit development, failed attempts, and final solutions.

II. recall

The `recall` challenge presents a vulnerable string comparison function. In this challenge, the `verify` function checks a user-provided input against a string in memory, breaking out of the loop upon finding a mismatching character.

A. Solution

The team assumed the solution would be ASCII text, as had been the case for `err0r` and `fizzy`. Based on that assumption, they followed the approach recommended in ChipWhisperer's SCA 101 course [1].

To guess the character at index $0 \leq i < 16$, the team constructed a reference string S_{ref}, where the first i characters are the correct ones determined in previous rounds, and where the character at index i is known to be incorrect. For this purpose, the team used a null byte, assuming the solution would be printable ASCII. The team finally captured a reference trace T_{ref} of length T_{len} of the target performing the string comparison on S_{ref}.

With that setup done, the team proceeded with attacking the i-th character. They brute-forced all possibilities for that character, capturing the trace $T(c)$ for character c. The correct character would run `verify` for longer due to the loop breaking one character after the other guesses, creating a different power trace. Meanwhile, all of the incorrect characters would have traces very similar to the reference trace. Thus, for each character c the team computed

$$E(c) = \sum_{i=1}^{T_{\text{len}}} \left| T(c)[i] - T_{\text{ref}}[i] \right|,$$

and concluded that the character with the highest error E was the correct one. Using this method, they obtained the flag:

`p0w3R1skn0wl3dg3`

B. Efficiency

This solution reaches the stated asymptotic efficiency of this method. For a string of length n over an alphabet Σ, the team was able to extract it in $O(n \cdot |\Sigma|)$ time instead of the usual exponential complexity.

Practically, this method runs quickly, taking about ten seconds to find the target string of length $n = 16$ over $|\Sigma| = 96$. It was also very reliable, not relying on magic numbers to process traces. As such, the team often used the solution code for this challenge to debug issues with the ChipWhisperer Nano boards.

III. err0r

The `err0r` challenge computes the CRC32 checksum of a user-provided value twice, then compares the results. To successfully complete the challenge, the resulting checksums must be different.

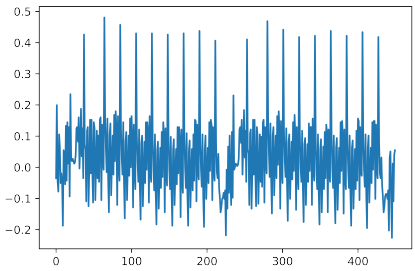

Fig. 1. Power traces obtained from the `crc32` functions in the `err0r` challenge.

A. Solution

Since the `crc32` method was what calculated the CRC32 checksums, the authors decided to use a FIA during one of the method calls in order to modify the output of one checksum, successfully completing the challenge. First, the authors ran a reference implementation in order to capture and save the expected output of a failed attack. Then, they proceeded to run a power trace to determine when the `crc32` method was called.

From the graph in Figure 1, the authors surmised that the first `crc32` method was run from 0-200 glitch clock cycles, and decided to brute-force voltage glitch attacks in the range from 30-70 glitch cycles in order to ensure a more efficient run time. For each attack, they checked if no output was returned (indicating the attack bricked the device), and reprogrammed the device if necessary. All other output was compared with the expected result and saved only if the output was different, showing that the attack succeeded. After the attacks were run, the only output saved was the flag, which were the ASCII characters `[0x57, 0x49, 0x4e, 0x21]` and translate to "WIN!".

B. Efficiency

Although the team's solution was a brute-force attack, it was fairly efficient as it was run over a relatively short period of time. It took thirty seconds to complete the attack on the authors' systems.

C. Failed Attacks

When the authors first attempted the attack, they didn't realize that the glitch clock cycle runs 16 times faster than the board clock cycle, so the resulting power trace indicated `crc32()` was run from 500-3000 glitch clock cycles, a much broader range that resulted in a less efficient brute-force attack. Once the authors realized their error, they were able to run another power trace and greatly narrow their range to the final version presented above, vastly speeding up result collection.

IV. CRT

The `CRT` challenge implements the RSA-CRT encryption algorithm and runs signature generation on a user-provided input (that is, $M^d \bmod N$), providing the intermediate values M_P and M_Q as output. The challenge also provides a Python program encrypted with AppArmor that takes these intermediate values and provides a full RSA signature as a result. Finally, the end goal is to identify the RSA private key used by the board.

A. Bellcore Attack

1) Solution: In 2000, Boneh, DeMillo, and Lipton presented an attack known as the Bellcore attack on RSA-CRT, described as follows [2]. Let S be the correct RSA signature for a given input. Without loss of generality, assume that a hardware fault corrupts only one of the intermediate outputs M_P, leaving M_Q untouched, and call the resulting signature $\hat{S}$. By the properties of the Chinese Remainder Theorem, it is known that $S = \hat{S} \pmod q$ and $S \neq \hat{S} \pmod p$. Then,

$$gcd(S - \hat{S}, N) = q \tag{1}$$

and the other factor of N can be easily found.

The authors implemented this attack on the ChipWhisperer Nano by first using the provided `gen_signature` function to calculate a known good signature of an input parameter. Then, the authors conducted voltage glitch attacks from 21000-35000 glitch clock cycles, checking whether Equation 1 contained a valid factor of N. Once a valid factor was found, the authors reconstructed the RSA private key, verifying that a self-constructed signature matched the known good signature. The recovered private key parameters were:

$$p = 962476599190059883, q = 1084024262488859977$$

$$d = 74093197121930911128075719999234449$$

2) Efficiency: This attack relies on brute-forcing good parameters in order to cause a fault in one component of the resulting signature. However, even given reflashing delays and other invalid keys, the attack typically succeeded within four minutes of starting, which represents a tiny fraction of the amount of time it would typically take to brute-force the key.

B. Failed Attacks

1) Leaking Modular Exponentiation: The team's original method relied on exploiting the leakiness of modular exponentiation using the multiply-and-square method, as described by LiveOverflow. This attack relies on the fact that different operations are conducted when performing modular exponentiation, depending on the parity of the key bit. For an odd bit, a multiply and square operation are performed, while an even bit only has a square operation performed. Thus, given an array like `[multiply, square, multiply, square, square]`, the team can deduce that the bottom 3 bits of the key must be `011`, since the algorithm operates from the least significant to the most significant bit.

Because the `modular_exp` function was broken up into repeated calls of the `modular_mul` (modular multiplication) function, power analysis was only possible on the multiplication function. After several hours of analysis, the team deduced that one medium spike was an addition operation, a medium

and large spike was a multiplication operation, and two large spikes signaled the end of multiplication, the equivalent of a square operation for exponentiation. Finally, the team recognized that if a multiply and square operation were both performed, the underlying additions and multiplications were repeated. Thus, the plan of attack was as follows:

1) Split up the power trace into additions, multiplications, and terminating squares.
2) For each square, see whether the underlying add-mult array could be evenly split into two identical subarrays. If so, mark key bit as 1; else, mark key bit as 0.
3) Repeat steps 1-2 until the key was recovered.

Given enough time, this approach would have managed to leak both dp and dq. Unfortunately, this approach was fundamentally flawed. Given only dp and dq, there is no known polynomial time algorithm to calculate p and q. While Jochemsz and May did introduce an attack, it is restricted to $dp, dq < N^{0.073}$, which does not hold here. Finally, the ChipWhisperer Nano has a hardware limitation that limits the sample collection capabilities to 100,000 samples maximum. In the team's testing, the team found that this was only sufficient to leak the bottom 6 bits of dp before running out of memory, rendering the attack a failure.

V. FIZZY

The `fizzy` challenge runs bubble sort on an array and outputs the final sorted result. However, every time a swap executes, the board loops for some time doing meaningless multiplication. Thus, the intent of the challenge is for the team to measure when the swaps occur, and to identify the original array from that.

A. Fault Injection

1) Solution: FIAs are known for being able to skip critical instructions, so the team conjectured the following shortcut to leak the original array. Instead of conducting power analysis, they would simply inject a fault as the code was branching to the subroutine, causing the call instruction to be skipped. Since no sorting had occurred, the code would then output the original array. This method ended up working, allowing the team to recover the flag:

`TIMINGSIDECHANNELSARESOCOOL`

2) Efficiency: Aside from not being the intended solution, this method has many drawbacks. First, the attack requires a pre-requisite brute-force to determine the exact timing for the fault. As well, the timing is very tight, making the exploit incredibly inconsistent and thus difficult to find the flag. Nonetheless, the fact that it worked once provided the team the flag and made the task of crafting the intended exploit much easier.

B. Power Analysis

1) Solution: As noted before, the `swap` function spends time performing multiplications that do not affect the final output. They exist merely to provide a distinctive power trace, shown in Figure 2. The team wrote a program to isolate the trace's relatively high peaks and find the time difference between them.

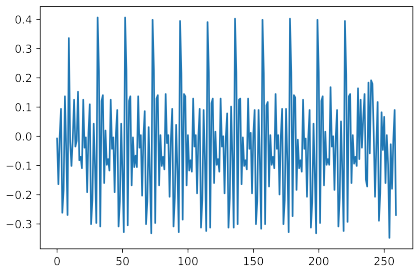

Fig. 2. Distinctive power trace of the multiplications within a call to `swap` for `fizzy`. The trace has relatively high peaks at regular intervals.

Manually looking at the data, the team came to the following conclusions. Individual peaks within a call to `swap` are separated by at most 22 clock cycles. Gaps of that length or less can thus safely be dropped from consideration. More importantly, the `swap` function has a "cooldown," and adjacent calls to it are separated by either 44 or 66 cycles. If some pairs of elements are not swapped, the delay between the calls will increase by 17 cycles per pair. Finally, starting a new pass of bubble sort will further increase the delay by 17 cycles.

These rules can be applied to find how many pairs of elements were considered and not swapped between two pairs that were. If c cycles elapsed between two sets of multiplications, the number of intervening operations can be computed as either $\frac{1}{17}(c-66)$ or $\frac{1}{17}(c-44)$, whichever comes out to a whole number. Each operation is either a pair that was not swapped or the code moving back to the start of the array for another pass. These cases can be differentiated by keeping track of where in the array the swaps are happening.

Ultimately, these rules are used to construct the sequence of swaps $\sigma_1, \cdots, \sigma_k$ applied to the original array A_0. The team was given the final array

$$A = \sigma_k \cdots \sigma_1 \cdot A_0,$$

so they could reconstruct A_0 as

$$\begin{aligned} A_0 &= \sigma_1^{-1} \cdots \sigma_k^{-1} \cdot A \\ &= \sigma_1 \cdots \sigma_k \cdot A, \end{aligned}$$

and obtain the flag:

`TIMINGSIDECHANNELSARESOCOOL`

2) Efficiency: The method presented here is as efficient as it can reasonably be. It runs in linear time on the number of swaps, and thus quadratic time on the size of the array. Practically, the analysis runs fast, and the time it takes is dwarfed by the time taken just getting a trace.

It is also dwarfed by the amount of time it took to create the algorithm itself. The rules given above are complicated and surprisingly tricky to implement in code. For example,

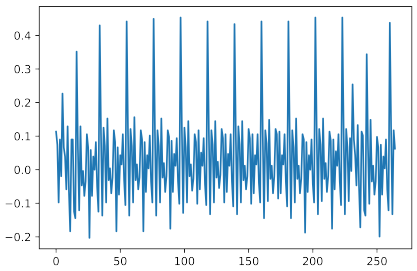

Fig. 3. Distinctive power trace of the multiplications within a call for `casino`.

the rules only give information about what happens between swaps, making it easy to run into fencepost errors. In fact, the team originally didn't even implement their approach in code, opting to do everything manually. It took much longer, but it was enough to verify the approach's correctness before proceeding further.

VI. CASINO

The challenge `casino` provides an array with the correct values but asks the authors to find the correct order. For each element in the array, it performs a number of multiplication equations equal to the value of the element of the array itself. The authors learned from `fizzy` that repeated multiplication has a distinct power trace, so if the authors could count the number of power spikes for each element in the array, they could determine its correct order.

A. Solution

The authors first ran a power trace on the challenge to check if the resulting graph supported their theory that the solution could be solved by counting the power spikes associated with each element of the array. As the graph supported the authors' theory, the authors were able to determine that the graph did seem to show regular spikes of power that corresponded to the multiplication operations. They saw that the spikes of power associated with multiplication were consistently over 0.4 volts. The authors knew from `fizzy` that the distance between power spikes associated with multiplication operations occurring in a loop were about 21 processor glitch cycles apart, so a longer gap between the spikes indicated that the process had restarted on the next element of the array. They then used a run-length encoding algorithm to count the number of spikes associated with each element in the array. If the spikes were roughly 21 glitch cycles apart, they were considered multiples of the same element. A longer gap signalled the start of the next element to be counted. In order to account for the fence post principle, 1 needed to be added the each count, which was implemented by rounding the final count of power spikes to the nearest 10. When put together, this data became an array with its elements in the correct order, revealing the order as:

```
120 90 80 60 110 10 50 20
30 40 150 140 70 100 130
```

B. Efficiency

The presented solution is fairly efficient given that it simply iterates through the power trace while counting. The solution took only a few seconds to run on the author's computers. If L represented the magnitude of the largest element in the array and n represented the length of the array, then the solution could be described to have a time complexity of $O(L * n)$. It would be difficult to make this solution more efficient as the only way to get an accurate count of the power spikes present is to iterate through the entire power trace.

VII. FIASCO

The `FIAsco` challenge contains an implementation of AES, and the team was challenged to find the key. Typically, the techniques of correlation power analysis (CPA) or differential power analysis (DPA) are well suited to attacking these types of problems.

A. Solution

While the team could have implemented the logic for implementing a CPA attack themselves, they realized that the challenge binary originated from a ChipWhisperer tutorial [3]. As such, the team simply used the ChipWhisperer's built-in library to perform CPA, captured some traces to feed into it, and obtained the flag:

```
0x 2b 7e 15 16 28 ae d2 a6
   ab f7 15 88 09 cf 4f 3c
```

The team then verified that they had the correct key by encrypting some data with it and ensuring the self-generated output was equal to the board's output.

B. Efficiency

Given most of the work is done by the ChipWhisperer Analyzer library, the team does not have many parameters to optimize. The main variable they control is the number of traces used N_{traces}. Given the correlation step is at least linear in N_{traces}, it is best to capture as few traces as possible. The team was able to get the correct answer with as low as $N_{\text{traces}} = 50$. Lower numbers were not tested.

VIII. SEARCH

The `search` challenge begins with an array of 257 integers, from 0 to 255, followed by a terminating 0. Then, the program removes 6 numbers from the array, leaving the team to figure out which numbers were removed.

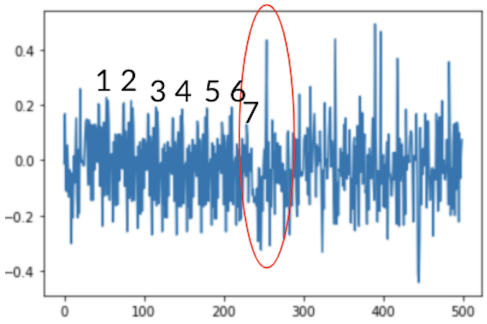

Fig. 4. Power trace of a call to `binarySearch` with an input of 0 (which was not removed) for `search`. The trace has seven distinct peaks before a "big one".

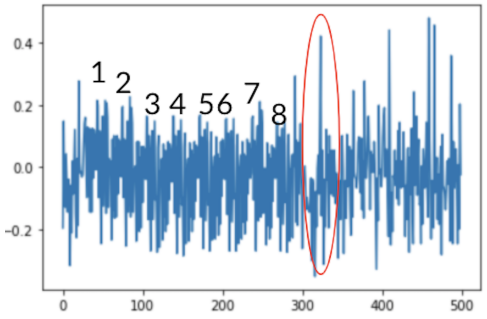

Fig. 5. Power trace of a call to `binarySearch` with an input of 59 (which was removed) for `search`. The trace has eight distinct peaks before a "big one".

A. Solution

The worst-case runtime of a binary search function in a given array is $O(\log_2 n)$, where n represents the array's length. Thus, if a number does not exist in the array, the search function should run $\log_2(251) \approx 8$ times before returning. These missing numbers can be identified by counting the iterations of the search value in a power trace, then identifying which numbers contain 8 peaks.

First, the team created a power trace for the `binarySearch` function with an input of 0, as seen in Figure 4. Upon looking at the traces, the team noticed that after several iterations, there is a large spike (typically larger than 0.35) in the power trace, which the team colloquially referred to as a "big one", which they assumed represented the end of the search function.

Initially, the team's scope range was between 0 and 250 clock cycles, as the initial large spikes occurred around time 250. However, the team noticed that 59's large spike's time would not print. As a result, the search range was extended to 0-500 clock cycles, resulting in enough data capture to identify which numbers had their large spike after 300 clock cycles, resulting in a flag of:

```
0x 27 5b 76 98 cf e8
```

B. Efficiency

The script checks 257 power traces with a time complexity of $O(n)$ where n is the length of the array. While this is a good runtime, the program could have been further optimized by checking for the highest non-removed number.

If a number before the middle value ($\frac{257}{2}$ in the challenge) was removed, then it would take more than one iteration to find the middle value, indicating that the array has been downshifted. To find the first removed integer, an algorithm could disregard the upper half of this list, checking if the middle values takes:

- one iteration, in which case no value before the middle has been removed, allowing the team to disregard that section,
- longer than one but less than $\log_2(n)$ iterations, in which case a value higher in the list has been removed,
- or $\log_2(n)$ iterations, signifying that the middle number has been removed.

In total, the time complexity of the described algorithm would be $O(k \log(n))$ where k is the number of integers removed and n is the length of the array, a significant time savings over $O(n)$ when k is small compared to n.

IX. CALC

In the `calc` challenge, the authors were tasked with attempting to reconstruct the original array of numbers in a calculator program using leaky operations.

A. Solution

The authors began their attack on the least significant bit of each integer element in the array. First, the $mult()$ method was run 31 times with a scalar of 2 in order to push the last bit to the first position. Next, the $divisor()$ method was run 31 times with a divisor of 2 in order to shift the least significant bit back to the last position while ensuring that every bit in the integer was the same as the least significant bit. From here, the authors could run the $dividend()$ method on the altered integer.

If all of the bits in the integer were $zero$ (indicating the least significant bit was 0), the method would not execute and would have a different power trace than if it did execute (indicating the least significant bit was 1). In order to ensure this method worked on subsequent bits, the authors would first left shift the current bit to the last position before beginning the process to isolate the bit. This allowed the authors to reconstruct the original array in reverse order bit by bit, resulting in the following output:

```
0x 67 d3 3a cd b4 1e
   9c bf e7 05 21 77
```

B. Efficiency

If n represents the length of the array and each element is b bits, then the efficiency of this solution is roughly $O(n * b^2)$. The actual process of isolating bits is a fixed number of

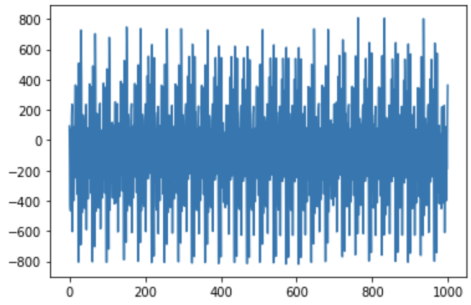

Fig. 6. Distinctive power trace of `homebrew`'s stream cipher encrypting random data. The trace has only two styles of peaks: wide and skinny.

operations. Although later bits require an increasing number of left shifts to put them in the last position before they can actually be isolated, since the maximum number of these increased operations will always be less than or equal to a constant, they can be disregarded in consideration of the overall efficiency. On the authors' systems, the solution took about a minute to run per element in the array.

X. HOMEBREW

The `homebrew` challenge is a custom stream cipher that runs different code paths depending on the parity of a specific key bit, with the challenge being to find the private key.

A. Solution

First, the team took a power trace of the program (see Figure 6) and noticed that there were only two types of peaks: wide and skinny. They assumed the wide peaks were the "else" implementations, indicating a 0 for the private key's bit, while skinny peaks implied a 1. This approach led to 128 distinct key bits, validating the team's approach as the key was 128 bits long. However, this process needed to be automated.

The team split the power trace into intervals, then noticed that the wide peaks were low time values grouped in two or more (for example, `[600, 660, 600]`), and separated with a time value greater than 15. Meanwhile, the skinny peaks were only zero or one spikes long. The team implemented code that would scan through the power trace and differentiate between wide and skinny peaks, substituting a 0 or 1 for the private key bit as desired. Finally, the team verified their output by encrypting the data with their generated key and comparing it to the board's output.

At first, the team had actually inverted their assumption on the mapping between wide and skinny peaks and bit parity. After adjusting the program such that a wide peak corresponded to a 1 and vice versa, the team's verification passed, resulting in a key of:

```
0x 80 88 25 dc ad 52 c9 71
   01 6f f2 64 7f 26 95 a8
```

B. Efficiency

As the script needs to scan through the entire power trace, the time complexity is $O(n)$ time, where n is the length of the power trace.

XI. NOTSOACCESSIBLE

The `NotSoAccessible` challenge implements the SIMON cipher, originally developed by the National Security Agency in 2013 [4]. The challenge provided access to read or inject a fault in the 26th round of the cipher, and asks the authors to identify the private key given a leaked half.

A. Proposed Attack

After some research, the team discovered an algebraic differential fault attack proposed by Le, Yeo, and Khoo [5]. In the paper, Le, Yeo, and Khoo describe an attack that only requires a single fault injection in the $T-6$th round, where T represents the total number of rounds. Algorithm 4, the proposed attack, can be summarized as follows:

1) Pick a plaintext P and obtain the encrypted ciphertext C.
2) Construct a set A, which consists of several subsets:
   - L, the set of linear equations to solve for the key schedule,
   - D, the set of differential faults that arise from computing $\delta_l^j(i)$ for key bit i and the (j,l)th entry in the differential fault table,
   - S, the original set of SIMON cipher equations.
3) Since the first half of the private key is known, substitute these values into the set A and pre-solve as many equations as possible.
4) Feed the remaining equations in A into a modern SAT solver, returning the key variables found.

Since the challenges in set 3 were released rather late, the authors attempted to conduct some rudimentary fault injection, but were unable to re-construct the set A as described by Le, Yeo, and Khoo in time [5]. However, the authors have full confidence that this attack is the correct attack, and given more time would lead to a successful leak of the private key.

B. Efficiency

Because the team is given 64 bits of the 128-bit private key, the authors suspect that the runtime to solve the set A would be faster than Le, Yeo, and Khoo's original findings, where fixing 22 key bits led to a successful solve in only 47.4 seconds. Given even more key bits, the runtime would likely be much smaller.

XII. CONCLUSION

In this paper, the team details their approach to analyzing and solving the challenges presented to them as part of the CSAW ESC 2021 competition. It covers their initial impressions about the code, as well as any patterns they could find in power traces. It also traces their attempts at exploit development, considering both failures and successes. Finally, the paper speculates on how the team's attacks can be made to run more efficiently and more robustly in the future.

REFERENCES

[1] ChipWhisperer, "SCA101: Part 2, Topic 1, Lab B: Power analysis for password bypass." [Online]. Available: https://chipwhisperer.readthedocs.io/en/latest/tutorials/courses_sca101_soln_lab%202_1b%20-cwnano-cwnano.html#tutorial-courses-sca101-soln-lab-2-1b-cwnano-cwnano

[2] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of eliminating errors in cryptographic computations," *J. Cryptology*, vol. 14, pp. 101–119, 2001.

[3] ChipWhisperer, "SCA101: Part 4, Topic 3: ChipWhisperer analyzer CPA attack." [Online]. Available: https://github.com/newaetech/chipwhisperer-jupyter/blob/8b4d4cfa7cc4851488bb479583fb5a0dec3a5ab3/courses/sca101/Lab%204_3%20-%20ChipWhisperer%20Analyzer%20CPA%20Attack%20(MAIN).ipynb

[4] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The simon and speck lightweight block ciphers," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2744769.2747946

[5] D.-P. Le, S. L. Yeo, and K. Khoo, "Algebraic differential fault analysis on simon block cipher," Cryptology ePrint Archive, Report 2021/436, 2021, https://ia.cr/2021/436.