Final Report on Side Channel & Fault Injection Attacks on CSAW'21 ESC Challenge Sets

CSAW 2021 Embedded Security Challenge

Md Toufiq Hasan Anik, Mohammad Ebrahimabadi, Javad Bahrami, Suhee Sanjana Mehjabin and Naghmeh Karimi
Dept. of Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, Maryland
(toufiqhanik, ebrahimabadi, jbahram1, suheesm1, nkarimi)@umbc.edu

***Abstract*—This manuscript deals with the vulnerability of the problems provided by the CSAW 2021 Embedded Security Challenge as a red team, concentrating on side-channel and fault injection attacks. We address the presented challenges and their pitfalls in this paper. The technical approaches that can be utilized to leak confidential information were also explored, as well as potential mitigation for such vulnerabilities. Our UMBC-SECRETS team was able to crack all 10 given challenges using computational approaches. This paper breaks down technical specifics of the attacks into simple stages.**

***Index Terms*—CSAW, Embedded Security Challenge, Side Channel Attacks (SCA), Fault Injection Attacks (FIA)**

I. Introduction

The 2021 Cyber Security Awareness Week (CSAW) Embedded Systems Challenge [1], [2] focuses on vulnerability of computer programs written in C programming language running on an micro-controller architecture. 3 sets of problems with 10 challenges in total are provided for the red team to analyze the vulnerability and perform potential attacks to extract the secret information from the given operating programs. The main assumption is that if these programs are running on a system, how an attacker can manipulate the system through side channel analysis or fault injection to leak the hidden messages.

As a red team, different attacks have been launched on all the challenges provided by our UMBC-SECRETS team. We realized that some of running operations are clearly visible from the power traces. The amplitudes of the power traces may represent how many bits are being processed inside, or which particular instruction is running. Also, some information about the execution time can be leaked from power traces. Such information can help in uncovering the secrets. Finally, although the fault injection function of the given board does not have much flexibility, it is still quite effective in attacking some of the given challenges.

The rest of this report is structured as follows. The preliminary backgrounds are discussed in Section II. Then in Section III we discuss all the challenges one by one including the problem statements, possible vulnerabilities, attacking methodologies and any possible mitigation scheme. Section IV gives a summary of the works done and concludes the paper.

II. Preliminary Backgrounds

A. Side Channel Attacks (SCAs)

While in operation, electronic devices leave traces such as power, heat, time, sound, electromagnetic radiation (EM), and so on. These features can be utilized to undertake statistical analysis in order to carry out side-channel attacks [3]. Figure 1 illustrates the side-channel leakage that may be extracted from a cryptographic device while operating.

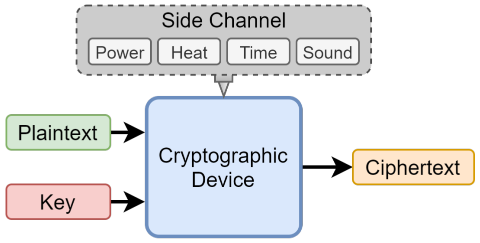


Figure 1. Illustration for side channel leakage from cryptographic devices.

Side-Channel Analysis Attacks (SCA) focus on the three main key properties of a functional device: timing, power, and electromagnetism, and realize the following attacks respectively: a) Power Analysis Attacks, b) Timing Attacks and c) Electromagnetic Attacks. In this study, we will mostly emphasize on power and timing analysis attacks.

B. Fault Injection Attacks (FIAs)

The Fault Injection Attack (FIA) focuses on extracting secret keys from cryptosystems by injecting faults during their run time and analysing the impact of such faults on the device outcome. One type of FIA is Differential Fault Analysis (DFA) attack. The basic idea behind this attack is to analyze the relationship between the faulty and fault-free ciphers to uncover the secret key. By analyzing the relation between faulty and fault-free ciphers, we try to reduce the search space of the key [4]. Fault attacks also can be realized as Fault Sensitivity attacks and Fault Intensity attacks, but considering the nature of the given problems they were not applicable to the challenges given in this contest. So we do not discuss them here.

C. Chipwhisperer Nano

ChipWhisperer is an open-source toolchain, which makes it simple and inexpensive to learn about side-channel attacks [5]. The platform also provides a well-documented and cost-effective means of doing side channel research in a repeatable fashion [6]. ChipWhisperer-Nano is the most cost-effective platform for executing side-channel power analysis attacks.

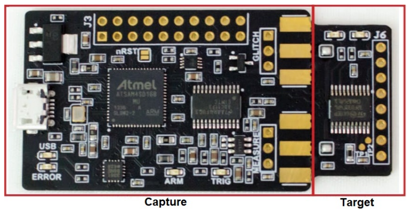


Figure 2. ChipWhisperer Nano device [7].

Figure 2 shows the Chipwhisperer Nano board [7]. As shown, it has two sections refereed as Capture and Target. The desired algorithm such as a crypto algorithm is running in target board, and the capture board is just to communicate with target board and also monitoring power or injecting fault.

III. Attacks Launched on the Given Challenge Set

Note: Please note that the power traces from ChipWhisperer is not always stable. For example, some times when extract the power traces, the power samples are saturated to 0.5 or it has some offset. Normally, iot should have value between -0.5 to 0.5. This makes our attacks to be unsuccessful in that case please reprogram or record again until you get good traces.

A. Set 1: CRT

Problem Statement: The attacker team is supposed to find two large prime numbers (P and Q) used for generation of N (used for generating public and private keys of the RSA algorithm). RSA is an asymmetric encryption algorithm that includes public and private keys where private key is used for decryption and public key for encryption. Here $N = P{\times}Q$ where P and Q are 2 large prime numbers. This algorithm can be used as a one way function as follows:

Encryption:
C = f(M) = M^e mod N where (N,e) build public key

Decryption:
M = f^{-1}(C) = C^d mod N where d builds the private key.

Considering the required operations in this algorithm (Exponentiation operation, modular multiplication, and etc.), different optimized algorithms are introduced in the literature to reduce the execution time of the RSA. For instance, the exponentiation algorithm can be sped up using left-to-right and right-to-left algorithms. Apart from that, RSA encryption is significantly faster than the decryption due to the shorter public exponent's length. So, other algorithms such as Chinese Remainder Theorem (CRT) are used to speed up the decryption algorithm. Although the number of modulo exponentiation increases from one to two in the CRT, they use half size modulus and exponents. So, each of them are around 8 times faster, and the whole CRT-based decryption is around 4 times faster than the baseline implementation.

Vulnerabilities: Based on the "SLON_Lab2_1-Fault Attack on RSA"'s report of the ChipWhisperer documents, one option for extracting secret information (P and Q) is using the fault attack, as the board is capable of the fault injection on the power supply (VCC). The second option is using both trigger_high() and trigger_low() functions for collecting power traces, and launching the power/timing side-channel attacks. These two functions are added while the rsa() function is called.

Methodology: As CRT is used inside the RSA algorithm, d is replaced by two smaller new exponents d_P and d_Q. So, we have:

s_1 = m^{d_P} mod P, and s_2 = m^{d_Q} mod Q

where s is:

i_Q = Q^{-1} mod P, and s = s_2 + Q×(i_Q(s_1 - s_2) mod P)

Based on our experiment, the injected fault affected the s_1, and as a consequence, using equation $gcd(m - {s'}^e, n)$ (s' is the faulty cipher) results in P(Q). Then we can find Q(P) with $Q(P) = N/P(Q)$ equation.

Extracted Information: The value of P and Q extracted by launching our attack were:

$P = 10840242624888599777, Q = 962476599190059883$

Other Possible Attacks: Timing side channel attack can be launched since by analyzing the .c code we realized if the value of b and exp are odd both in $modular\_mul$ and mod_exp, the execution time gets longer as one more operation is executed.

Mitigation: We can mitigate the fault injection attack by decrypting the ciphertext right after generating it. In this case, we should get the original plain text, otherwise the output of encryption is faulty. Therefore, we can block the output cipher if a fault is injected into it.

B. Set 1: Recall

Problem Statement: The first challenge set also includes the Recall challenge, which provides us with a hex file with the concealed memory of 16 elements, each of which having one byte of secret information. It also contains a C programming language representing the hex file, which compares a message provided to the platform with the memory specified in the hex file but does not contain the concealed memory. If the message verifies the concealed memory, a "one" is returned; otherwise, a "zero" is returned. The objective of this challenge is to find the C code's vulnerability and use a side-channel attack to disclose the concealed memory.

Vulnerabilities: The verify function in the supplied code tries to match the user provided memory with the concealed memory. If it does not match, the operation is interrupted; however, if it does match, the procedure is continued. The vulnerability can be represented as variation in the power trace's execution time, which might lead to a timing attack relying on the loop's execution time.

Methodology: Since each memory element is one byte, there are 256 possible values for each of the 16 memory locations. We can take one index at a time, derive the power trace for each of the possible elements of the memory. Execution time will be similar for all the values except the correct one. Finding the memory value for which the execution time will be longer reveals the correct value in memory. There are many possible ways to find the execution time. However, we find that the correlation between the power traces is an efficient and reliable solution. The wrong values power traces will be highly correlated with each other. However, the ones that have less correlation will be the correct memory element. Since there is a single correct value in the memory, its power trace is different than all others wrong memory value and thus the value of the correlation function is lower. We can take the index value having the lowest correlation function to be the value associated with that memory location.

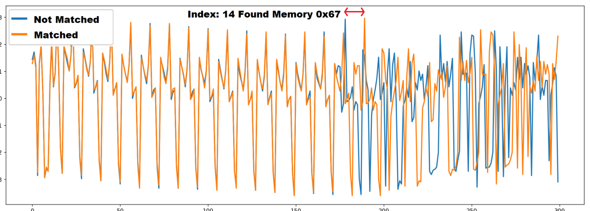


Figure 3. Visible change in execution time between an mach and non matched memory element from the power traces.

Figure 3 shows the visible change in execution time between a match and unmatched memory element from the power traces. We can see on the red arrows that the matched element has shifted traces than the unmatched one.

To reduce the complexity, we can correlate the power trace of all the possible memory values (0 to 255) for a byte through iterations and whenever we get the low correlation, we can conclude that the value associated with that low correlation is our desired memory value. With this approach, there will be a maximum of 256 iterations to be considered for each memory location but we break the operation as soon as we get a low correlated power trace or the correct value which decreases the worst-case time complexity. Thus the number of iterations for each memory location will depend on the memory value where we find a match that decreases the average time complexity of our program. One drawback of this approach is that if the value of the hidden memory is 0, then correlating with all the traces will give a low correlation function. So to proceed with this algorithm, we need to check before iterations if the memory value is 0 or 1 for that particular memory location. We do that by taking the power trace of memory values 0, 1, and 2 and correlating each of those in pairs to check whether all the values of correlation are comparable. If they are not comparable, by finding the lowest correlation function, we find the memory value which can be either 0 or 1. If they are comparable, we proceed through with the iterations.

After all the iterations are over the deduced array is then sent to the ChipWhisperer which then compares it with the hidden memory and if all the elements match then returns "one", otherwise returns "zero".

Found Information: Memory: CWbytearray(b'70 30 77 33 52 31 73 6b 6e 30 77 6c 33 64 67 33')

Other Possible Attacks: Another possible attack might be a power attack. It seems that when there is a match, the $mem\_different$ becomes 1 from 0. This will generate a differential power trace.

Mitigation: We might consider three possible mitigation. The first one is removing the 'break' function such that most power trace has similar execution timing. The second one, instead of verifying in a loop 0 to 15, verifying with a random manner will have difficulties for an attacker to find the correct indexing. The last one is balancing the power related to $mem\_different$ with flipping another bit at the same time on opposite value.

C. Set 1: Fizzy

Problem Statement: The objective of Fizzy is finding the order of the elements in a predefined array. The attacker should find the array and use it as an input to the main program, and if attacker's guess matches with the secret array, the program returns 1, 0 otherwise. Among different functions inside the main program, the attacker can increase the success rate of his attack by analysing a few of them, such as **sort** and **swap** functions. Please note that by calling sort function the board returns the sorted array. It means that we know what numbers are inside the array, but we do not know – the attacker should find out – the order of numbers before it got sorted.

Vulnerabilities: As mentioned, the attacker requires to have a closer look at some functions to be able to extract the secret information. In case of *Fizzy*, a dummy loop is located inside the swap function which affects its timing in a way that makes the power signature distinguishable from where there is no swap operation. It is worthwhile mentioning that finding this vulnerability point could be more challenging in case that the attacker could only see the power traces, not the source code.

Methodology: The followings are the points that we took into account for launching our attack:

- **Sort function**: This function sorts the array, and the attacker can learn which elements exist in the secret array. (Again, the objective is finding the correct order of elements in the secret array)
- **Trigger high/low**: The code inside the sort function is sandwiched between trigger_high() and trigger_low() functions, meaning that power traces can be collected while this portion of the code is being executed.
- **Dummy multiplier**: Since there is a dummy multiplier inside a loop within the swap function, the power traces are substantially different because of the distinguished power traces as a result of the multiplication operation. During the execution of the swap function, we could observe 12 power peaks, 2 as a result of the $*xp = *yp$ and $*yp = temp$ operations, and the remaining 10 power peaks come from the dummy loop with 10 iterations.

Also, between each power peaks mentioned in the very last item, we could find a same repetitive pattern of power signature elements as following: 24 power traces between $*xp = *yp$ and $*yp = temp$ operation, 22 power traces between $*yp = temp$ and the beginning of the dummy loop, and finally, 19 power traces between each iteration of the dummy loop. For finding the correct order of elements in the secret array, we should learn when the swap function is executed, and since we have the sorted array, we can start the reverse of sorting operation backward from the sorted array all the way to the starting point to find the original secret array. For extracting swapping points, one Template array is filled up with 1 when there is a peak power element, and with 0 otherwise (the Template array only contains 1's and 0's). The only required operation which helps the attacker to find the location of the swap operation is the correlation of the Template array with the with all power samples. Figure 4 shows a part of power traces when we called the sort function. It is clear that when we have back to back swap there is not any gap between power peaks. However, when we do not have swap, there is a gap between power peaks, for example the gap between the third and fourth swap. By analyzing the gaps between power peaks, we can uncover when the swap is happened or not. Also the Template array is depicted on the bottom of the figure with orange color. The distance between each bar in the Template array as we mentioned earlier is the same as the power pattern when swapping operation is happened.

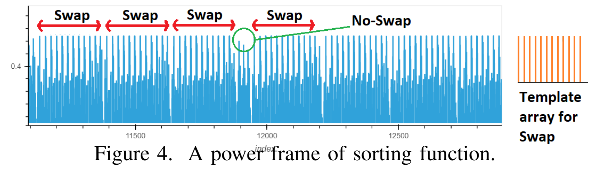


Figure 4. A power frame of sorting function.

The correlation between Template array and power traces must return a maximum value where there is a swap function. Although the attack scenario seems to be straightforward, a couple of exception cases make the analysis more challenging, for instance, in case of a back to back swap operation, we realized that there are 255 power samples. Second, if we have one non-swap operation between two swaps, the number of power sample increases to 272. By taking all these information into account, the analysis of the secret array started from the its sorted state to the beginning until the secret information get extracted.

Found Information: This the initialized memory before sort function: [84, 73, 77, 73, 78, 71, 83, 73, 68, 69, 67, 72, 65, 78, 78, 69, 76, 83, 65, 82, 69, 83, 79, 67, 79, 79, 76].

Other Possible Attacks: Fault injection attack is also another possible attack on this algorithm. For example, if we could apply fault at the time that the comparison for the swapping is happening, we can stop the swapping. In this case the order of the array will not change. Here we could uncover the position of 0x33 by fault attack. However as the board is unable to apply a repetitive fault, we cannot inject fault with this version of ChipsWhisperer. Thus we stick with aforementioned attack.

Mitigation: One effective solution against our attack is letting the dummy multiplier work when we do not have swap (or removing the dummy multiplier). Also we should have a dummy swap on two dummy variable when we do not have swap. In this case power pattern of swap and not-swap would be similar.

D. Set 1: Error

Problem Statement: The challenge "Error" included in the first challenge set opts to find a 4 Byte secret (called "win_code"). A function and its duplicated have been implemented and fed with same input. If the outputs do not match, the system returns an error message which is the win_code. The goal of this challenge is to perform a side-channel attack to extract the win_code.

Vulnerabilities: The C-code shows that it is possible to inject faults to corrupt the output of one of the two function calls without corrupting the other one. Such error eventually provides us the win_code. In this case, voltage glitches have been introduced to inject faults and reveal the secret by such side-channel.

Methodology: We have introduced voltage glitches to inject faults into the system to get the win_code. However, the problem that occurs is that the resolution of the output for matched function outputs and output for unmatched function outputs that provides win_code is the same, which makes it difficult to separate the two. To solve this issue, we take an arbitrary input value. Since the input values are identical, the matched function outputs are identical as well which enables us to distinguish between the matched output and win_code. Another issue with fault injection is to introduce the voltage glitch - at the time when the program performs a function call to generate a output. it does not matter whether the function call is to generate the first function output or the second function output.

If we inject the fault at regular intervals while the program is in operation. Three scenarios may occur in this situation:
i) the fault might be injected at a random position not concerned with which function call but functions outputs matched.
ii) the fault might be injected at one of the function calls.
iii) the fault might be injected at both function calls.
Since we introduce faults multiple times (300 in our case) through iterations, all three cases occur multiple times. Case (i) gives us the desired output for a match, case (ii) gives us the win_code, and case (iii) gives us a output that is different than both output for a match and win_code because both functions outputs become corrupted but matched each other. We can safely ignore the output for matched. In order to distinguish between cases (ii) and (iii), we can exploit the fact that the probability of a fault corrupting both function calls simultaneously is less than that of corruption in either of the two function calls. Output we get through the iterations more frequent would be the win_code. We then ensured that our observation was not based on the input signal that we used for computation. To do that, we repeated the process for 4 more random input signals. Each time the faulty output were found to be different for case (iii). Only the win_code of the case (ii) was the same in each cases. So we took the intersection of the set of all the codes for the five iterations to get the win_code with certainty, and thus successfully performing the attack and cracking the error challenge.

Found Information: Correct Winner code in decimal [87, 73, 78, 33]

Mitigation: Either instead of duplication, triplicating or in general N-replication is used. Or another solution can be making the checker secure by duplicating the checker itself. Or we can prevent erroneous output from coming out and in case of mismatch the system is rerun again till a match occurs and we send that value to the output.

E. Set 2: Casino

Problem Statement: Casino is the challenge from the second set, which contains a set of 15 elements already given, but not in the correct order. The goal of this challenge is to determine the correct order and verify it using the C program, where the program returns "one" if the sequence matches, else returns "zero".

Vulnerabilities: From the given code it is evident that there is a function called "draw" which performs as many iterations as the number of elements of the array whose sequence we need to determine. During each iteration, there is a loop that runs exactly as many times as the value of the respective elements of the array and performs a multiplication. For example, if the first element of the array is 30, it will loop through 30 times and perform multiplication in each loop. Multiplication is a computation-intensive operation and we can observe a peak in power trace whenever multiplication is performed. A power peak is observed in the transition of iterations as well but the peak is considerably smaller in magnitude than that of the multiplication operation. Thus by counting the number of peaks in the power trace, we can count the number of times multiplication operation has taken place in an iteration, and thus the value of the respective elements of the array. Thus we can exploit the power side channel and perform timing analysis to determine the sequence.

Methodology: Since we want to exploit the power trace, we first run the casino function and compute the power trace. We then distinguish the peaks that resulted from multiplication to those that resulted from the change of iteration by analyzing their magnitude. Then by counting the number of peaks between two iterations, we can calculate the value of the element of the array corresponding to that iteration and repeat the process for all 15 iterations to get the correct sequence. This sequence is then sent for verification for checking whether it is the correct sequence. Figure 5 show a power frame of draw function. The three red circle shows a drop in the power traces that hint us in the draw function the next iteration (the next value of the array) is initiated. The number of pick between two drop is exactly equal with the value of array element. As we told we count these peak till we observe a drop in the power traces.

Found Information: This the correct order of the memory: [120, 90, 80, 60, 110, 10, 50, 20, 30, 40, 150, 140, 70, 100, 130].

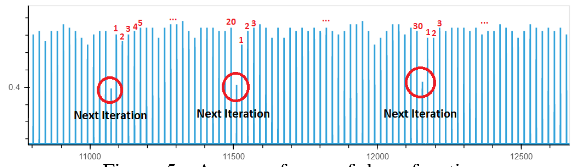


Figure 5. A power frame of draw function.

Other Possible Attacks: Analyzing the power traces during the verify function can be another way of attacking this algorithm. Although here we do not have access to the power traces when we call verify function, in a real case we always have the power traces during the run time.

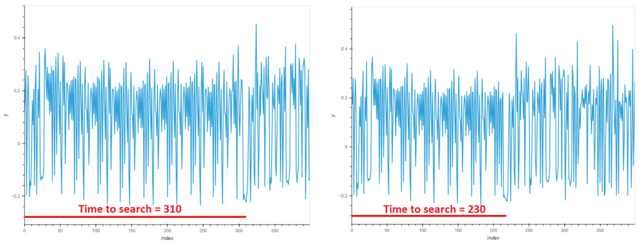
Mitigation: Based on the number of peaks in the power traces during multiplication, we could break this algorithm. One possible mitigation scheme is to remove the dependency of the number of multiplications to the values stored in the array. In this case it will be difficult to find the value that is loaded in the draw function. Another scheme can be randomizing the order of multiplications (not following sequential indexing).

F. Set 2: Search

Problem Statement: The "Search" challenge, first, initializes an array of 257 elements, then, it calls the "Remove" function to move 6 random elements of the array to the last 6 location of it (This process is repeated each time the C code starts). Each time a new elements is moved to the end of the array, the entire array is shifted to the left to be able to fit in the new element. By doing so, the elements in location 256 goes to location 255, the elements in location 255 goes to location 254 and so on. The attacker's task is finding those 6 elements **regardless** of their orders. Also, the attacker can verify the correctness of his/her guesses using the "Verify" function returning 1 if both arrays match, 0 otherwise.

Vulnerabilities: Both trigger_high() and trigger_low() functions are used when the "binarySearch" is utilized, meaning that the power traces can be collected and analysed later by the attacker when this function is called. In the binarySearch function, it first looks for the elements selected by the attacker by dividing the array into two chunks. If the selected number is not found in the first chunk, the algorithm look for it in the second one. It continuously divides the array by two until it finds the element. This search algorithm does not return any value showing that whether the elements is inside the array or not. However, the attacker can analyse the duration of the execution of the function to extract the number of times that the array's length is halved for the search operation. When the length of the power trace is longer, it shows that the selected element is located in the end of the secret array and search function cannot find it.

In Figure 6 we demonstrate the power traces for two cases when we called search function. Based on the figure, 310 power samples exist when it searches number 91 while 230 power samples exist when it searches number 123. The takeaway point here is that for those 6 numbers that are moved



(a) search(91) (b) search(123)

Figure 6. The power traces of querying the binarysearch function with 91 and 123.

to the end of array, the processing time is higher as the search function needs to go through its all iterations to find them. We observed that the number of power samples for that 6 values is around 310, meanwhile 91 is one of those 6 values. The power samples for all the values other than those 6 (the ones not moved) are lower than 310. Please note that when the process of search function is finished, the program stops collecting power traces, but ADC's memory still contains some garbage power traces. That power drop shown in figure 6 is when the program stops collecting power traces.

Methodology: The attacker inputs numbers between 0 and 2^8-1, and for each case, the respective power traces are collected and stored in a separate array. It is important to highlight that although the power traces are collected and observed, the attacker can distinguish desired power traces by looking the the duration of the power trace. It clearly states that the attack is a timing attack rather than a power side-channel. The arrays should be sorted either in an ascending or descending order, and finally, the attacker picks the first (ascending) or the last (descending) 6 elements.

Found Information: These are 6 values which are moved to the end of array: [207, 232, 152, 91, 59, 11].

Other Possible Attacks: Analyzing the power traces when the verify function is being executed is an alternative way of attacking this algorithm. Here we do not have access to the power traces during the execution of the verify function, but in general to attack this algorithm, we can consider this attack as well.

Mitigation: If the search function goes through all iterations regardless of the order of its numbers, the power signature and execution time of the algorithm is independent of the location of the array's elements. Doing so makes the attacker's life much harder if the timing side channel is of interest.

G. Set 2: FIAsco

Problem Statement: The goal of this challenge is to extract the secret key of the AES cipher using power/timing side-channels, or fault injection. AES is a block cipher with different byte-wise operations including substitution, XOR, shiftrow, and mixcolumn operating on different bytes of the intermediate state. For this challenge, we launched Correlation Power Analysis (CPA) analysis on the first round of the encryption.Hypothetical power model is generated using either Hamming Weight (HW) and finally, this model is used with the collected power traces to perform CPA.

Vulnerabilities: Though different implementations of AES are available in the c code, only a standard normal AES is used. Normal AES has 10 rounds. Those were completely visible from the picks in the power traces (Fig. 8). Several

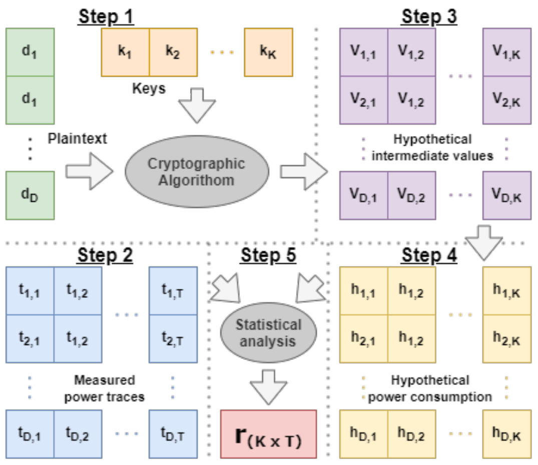


Figure 7. Generalized steps for power analysis attacks [3]

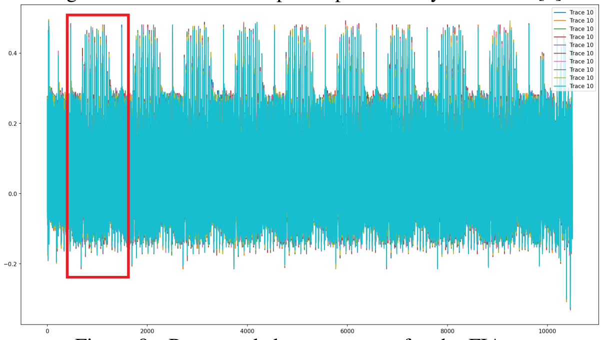


Figure 8. Pre-recorded power traces for the FIAsco.

attacks including DPA, CPA, MC DPA, Fault injections, etc can be applied to AES. We only used CPA attack in the first round of the AES. The main vulnerability for considering a power analysis attack is the changes in the power traces for variation of inputs.

Methodology: Based on our experiences, the total number of power samples for one pass of the AES encryption is around 10500. Since we targeted the first round of the algorithm, even with first 1500 samples with 100 (increase if needed) power traces the secret can be uncovered. Considering the steps shown in figure 7, followings are the required steps in the CPA attack:

Step 1: Selection of the Intermediate Result: This intermediate value should be a function of non-constant data (d) and a small part of the secret key (k). In our experiment, $D = 100$ (less is also possible) input plaintexts are used each having 16 bytes (AES-128). So, the required array has D rows each containing 1 byte of the plaintext. For the secret key (k) array, all combinations of 8-bit data is generated for each plaintext, meaning that the k vector has D rows, with values of 0 to 255 in each row (256 columns: as 256 combination possible for a byte key). The plaintext and the master key are XORed in the first round of the algorithm, and before applying the S-Box to the state.

Step 2: Power Measurement: In this step, power consumption is measured during the encryption, and power traces are gathered. Only first 1500 power samples are enough to attack the first round of the given AES-128. So, the dimension of the the matrix is D by 1500 in this case.

Step 3: Calculation of the Hypothetical Values: For every possible value of the secret key (k_1, k_2, ..., k_K), hypothetical values are calculated and referred as key hypotheses. Taking into account the dimension of the D and k arrays, this matrix has D rows and 256 columns (plaintext XORed with master key). Apart from that, h matrix which is the result of applying S-Box operation on the v vector is available, and is used in the next step for calculation of the correlation between h and t matrices.

Step 4: Mapping of Intermediate Values to Power Traces: Power models are used to map hypothetical intermediate values v to a matrix h of hypothetical power consumption. Each column of the h matrix is compared with all columns of the t matrix, and final correlation value is stored in the corresponding element of the r matrix. Hence, r matrix has 256×1500 elements in total.

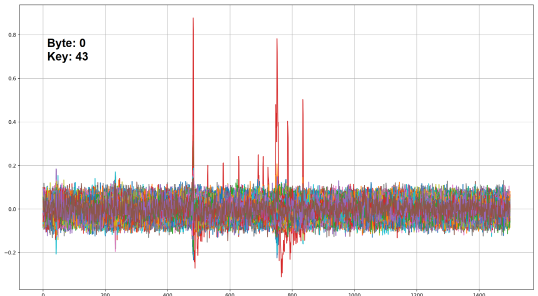


Figure 9. Correlation coefficients for byte 0.

Step 5: Comparison of the Hypothetical Values with Power Traces: The secret information will be extracted based on the analysis of the simulation vs. real measurement results. After the r matrix is generated, all values are plotted, and we realized that only one plot is substantially different from others with high correlation, corresponding to the correct master key of the algorithm.

Figure 11 shows the correction matrix for the power with the hypothetical value. We can see that correct key byte has a very high correlation value than other possible keys. This tells us that, the CPA model is perfect for attacking this implantation.

It is important to note that all mentioned steps are only for one byte of the algorithm, and since AES-128 has 16 bytes, the entire process should be repeated 16 times to be able to extract the entire secret key.

Found Information: Key in hex: ['2b', '7e','15', '16','28', 'Ae','D2','A6','Ab','F7','15','88', '9','Cf','4f','3c'] or in decimal = [43,126,21,22,40,174,210,166,171,247,21,136,9,207, 79,60]

Other Possible Attacks: Standard AES implementation has many security flaws in terms of side channel and fault injection. To mention some: DPA [8], CPA [9] on other rounds like 9th round, MC-DPA [10], Template attack [3]. Fault injection on round 9, etc.

There are tons of security analyses in the literature investigating the resiliency of this block cipher against different attacks[11], [12], [13].

Mitigation: There are many proposed solutions available. This C code has Masking which is a good solution. A higher order masking is more resilient. Moreover, parallel operations to increase the noise will bring difficulties for the attacker. Also power balanced operations will make the attack harder.

H. Set 3: Homebrew

Problem Statement: This challenge includes an encrypt function which operates on the received plaintext using different round keys. Also, two different sboxes (sb_1 and sb_2) are used for the substitution implementation. Each bit of different bytes of the master key is checked to see whether it is 1 or 0, and based on this value, input gets updated using different operations including XOR, multiplication, addition, and subtraction. The main objective of this challenge is finding all different 16 round keys.

Vulnerabilities: The encrypt function starts with trigger_high() and ends with trigger_low() functions, meaning that the power traces can be collected when the input plaintext is being encrypted. However, there is an interesting venerability in the code which refrains the attacker from launching power side channel. By analyzing the code, we found out that each block of the plaintext is independent from other bytes after applying the sbox during the encryption. By exploiting this independency, brute-force attack's order becomes $2^8 \cdot 2^4$ from 2^{128}, meaning that the attack can successfully extract the secret information in a much shorter time.

Methodology: The following steps have been taken:

- An array of random plaintexts is generated as the input candidates.
- for all 16 bytes of the secret key, all 2^8 combinations are generated.
- The output of the real hardware (ChipWhisperer) is stored in an array using target.simpleserial_read ('r',16) function.
- The hardware's output is compared with the output of the script (Python) that was written for the attack to find the key candidates
- Finally, in case that these two outputs match, new key will be appended to the key candidate array. After each two iterations, two different key candidates arrays are filed up with a new candidate key. Finally, for making the key space smaller after each iteration, the intersection of these two key arrays is used as the starting point key array for the next plaintext. Based on our experiment's results, on average, using only 4-5 plaintext resulted in breaking the cipher. It is worthwhile mentioning that due to the simple nature of the cipher, the attack's time always converge rapidly.

Found Information: Key in Dec.: [1, 17, 164, 59, 181, 74, 147, 142, 128, 246, 79, 38, 254, 100, 169, 21]

Other Possible Attacks: Timing analysis attack is quite simple in this encryption algorithm. When the key bit is 0 the algorithm uses two multiplication which is more time consuming than when the key bit is 1 (Xor and Plus). In this way we can uncover the key bit by bit, and finally the whole key rounds.

Mitigation: As we told this encryption method is quite vulnerable to brute forcing attack. Here we should make a dependency between keys and different cipher bytes; the same as other resilient crypto such as AES.

I. Set 3: Calc

Problem Statement: One of the challenges in the third challenge set is the "calc", which focuses on the array of three elements each being 4 bytes. The goal is to find the array called the "org_arr" which is generated using a pseudo-random number generator. The org_arr is also stored in another variable called "arr", which can be modified using basic arithmetic operations of addition, subtraction, multiplication, and division.

Vulnerabilities: There are two possible attacks that can be applied in this challenge. i) CPA attack - However, in this attack, the power trace variation is not satisfactory. Also, the attack is not strong enough to find all the possible bytes and comparatively has more execution time. ii)Buffer overflow - As there are addition and subtraction functions available in the C code there is a possibility of buffer overflow, which causes a significant power drop that is distinguishable from other traces. The side-channel attack, in this case, is also the power analysis.

Methodology: A buffer overflow has been exploited in order to determine the org_arr. Basically, if 1 is added to 1111 1111, we get 0000 0000. The hamming weight drops from 8 to 0 and there is a distinguishable impact on power trace. A similar impact is seen when we subtract 1 from 0000 0000 and get 1111 1111. Since we have the option to manipulate the memory held by the calculator through performing arithmetic operations, we can manipulate the data inside. But the array has three elements each of which has a length of 4 bytes. Thus a brute-force-based attack has the complexity of $2^{32*32*32}$ which is difficult in terms of time and resources required. To simplify the attack and minimize the complexity, we attack one byte at a time and loop through the four bytes in four different iterations. Our attack is summarized by the following steps:

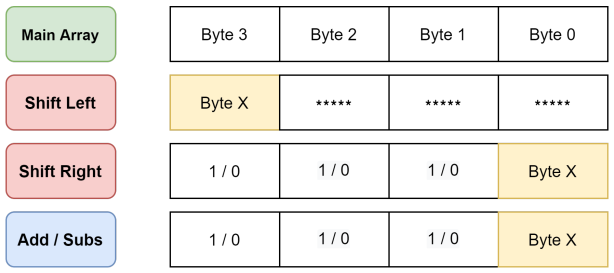


Figure 10. Operations for attacking a byte in Calc.

i) Getting rid of all bytes except the one we are attacking: we exploit the multiplication and division functionality of the code to achieve this goal. Since multiplication by 2 virtually means one bit left shift and division by 2 virtually means one-bit right shift, by successive multiplication and division by a power of two, we can shift the byte under attack enough so that all the other bytes are padded with 0 in case of positive numbers or 1 in case of negative numbers. The byte under attack is kept as the least significant byte. This can be done by shifting Byte X to MSB byte then shifting back to LSB. This operation is depicted in figure 10. This is an easy solution that we found. However, other calculation functions can be used only to keep the byte under attack to LSB byte position,

ii) Since we need buffer overflow, we collect power traces for addition from 0 to 255 and also subtraction from 0 to 255. This will generate a 32-bit flip operation while hitting the correct value in the loop. As the LSB byte will be changed due to the addition of 1 with 255 or subs traction of 1 from 0 will have change all the bits in the array, this will have a visible power impact with Hamming weight (HW).

iii) We repeat the first two steps for each byte and collect the power traces.

In order to remove noise from the power traces, we take multiple power traces for each operation and get their arithmetic mean. We have used 20 repeated trace collections.

iv) For each byte, we then correlate the power recorded traces (for 0 to 255) attractively. For example correlate power traces for 0 addition with 1 addition. High correlation signifies the absence of buffer overflow and low correlation signifies buffer overflow. As soon as we get a high drop of correlation, we determine the index, which is our memory value. We repeat the same attack for every byte to get the 12-byte arrays.

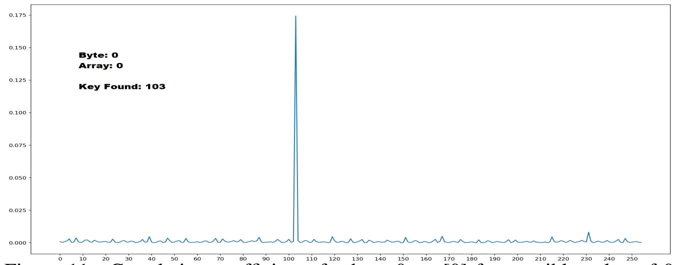


Figure 11. Correlation coefficients for byte 0, arr[0] for possible value of 0 to 255.

vi) Since we don't know the sign of the numbers, we get two different 12-byte arrays. We choose the element with a higher correlation for each memory space and verify. However, noise might result in a mismatch in which case, we check all the combinations of the two 12 byte arrays until we get a match. Our algorithm gives us confidence that each element is present in either of the two arrays. The complexity of this operation is $2^{12} = 4096$ which is reasonable. Then we finally apply brute forcing to figure out the final original array named org_arr.

Found Information: Array: [103, 211, 58, 205, 180, 30, 156, 191, 231, 5, 33, 119]

Other Possible Attacks: As mentioned before CPA with HW model has similar potential leakages.

Mitigation: There might be a couple of solutions for this. One, random array operation not serially one after another. Balanced operation Like running while bit flipping or random operation to remove the HW drop/rise. Though it is good for ASIC structure.

J. Set 3: NotSoAccessible

Problem Statement: The goal of this challenge is finding the secret key of last round of SIMON algorithm. It is a block cipher with n-bit word where n is required to be 16, 24, 32, 48, and 64 [14].The notation for representing the SIMON cipher with an m-word key is SIMON2n/mn. The version that is given to us has the word size of 16 and contains 4 chunks of words (key size = 64 bits, plaintext and ciphertext size = 32 bits). Each encryption round contains different operations such as XOR, AND, and left circular operation (left rotation operation), and the entire algorithm has 32 rounds. Each round of the key scheduler has an XOR operation along with the right circular operation (right rotation operation).

Vulnerabilities: Here is our naive idea that might lead to extraction of the secret data. By selecting all 0's as the plaintext, and as the right side of the state in the next round is the same as the left side of the current round, the right side of the state is all 0's. By applying the same analysis, the left side of the state is equal to the round key 0 (k_0). Afterwards, the right side of the state becomes k_0, and the left side becomes $k_0 \bigoplus \mathrm{k}_1$. One possible attack scenario might be the analysis of the intermediate state (using the same methodology) until it gets to the round 25. Since the power trace is available in that round, the attacker might be able to find a correlation between the power traces and the state value. Another possible attack is injecting fault in round 25 and analyzing the faulty cipher with fault free cipher to extracting key.

Methodology: Here We used the brute forcing and uncovering the key of round 32 as the number of brute forcing is not high. **Found Information**: The key of the last round: 0x8d14 and the master key: 0x1918111009080100.

Other Possible Attacks: Power analysis attack in earlier round of the Simon can recover key easily.

Mitigation: In case of implementing any concurrent error detecting algorithm along with the encryption, we can prevent the faulty cipher propagate to the output.

IV. Conclusions

In response to the CSAW 2021 Embedded Security Challenge, this study explores potential side-channel and fault injection attacks on the given challenges [2]. With our investigation, we identified that there are several potential attacks that may be conducted in order to get secret information from the provided programs. The attacks are detailed in-depth with explicit procedures, in the preceding sections. **All the attacks are successfully performed and automated codes with associated files are shared with the organizers.** We also provided ideas on how to mitigate the vulnerabilities. According to the observation, it is believed that all the given challenges have security flaws, therefore attacks can be carried out potentially through a side channel in order to retrieve confidential information. This shows that security sign-off, and in particular ensuring the resiliency of chips against side-channel and fault injection attacks, is a highly important stage for designing secure structures.

References

[1] "Csaw: Embedded security challenge," CSAW. [Online]. Available: https://www.csaw.io/esc
[2] "Csaw 2021 embedded security challenge (esc)," CSAW. [Online]. Available: https://https://github.com/TrustworthyComputing/csaw_esc_2021
[3] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards.* Springer, December 2006.
[4] D. Mukhopadhyay and R. S. Chakraborty, *Hardware security: design, threats, and safeguards.* CRC Press, 2014.
[5] C. O'Flynn and Z. D. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in *COSADE.* Springer, 2014, pp. 243–260.
[6] "NewAE Hardware Product Documentation." [Online]. Available: https://rtfm.newae.com/index.html
[7] "Perform Power Analysis Side-Channel Attacks with the ChipWhisperer-Nano." [Online]. Available: https://www.hackster.io/
[8] P. C. Paul C. Kocher et al., "Differential Power Analysis," in *CRYPTO*, ser. LNCS, vol. 1666. Springer, 1999, pp. pp 388–397.
[9] E. e. a. B. Brier, "Correlation power analysis with a leakage model," in *CHES.* Springer, 2004, pp. 16–29.
[10] A. Moradi and F.-X. Standaert, "Moments-correlating dpa," in *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*, 2016, pp. 5–15.
[11] F. Wegener and A. Moradi, "A first-order sca resistant aes without fresh randomness," in *COSADE.* Springer, 2018, pp. 245–262.
[12] C. Kison, J. Frinken, and C. Paar, "Finding the aes bits in the haystack: Reverse engineering and sca using voltage contrast," in *CHES.* Springer, 2015, pp. 641–660.
[13] B. Bilgin et. al, "A more efficient aes threshold implementation," in *Int'l Conf. on Cryptology in Africa.* Springer, 2014, pp. 267–284.
[14] R. Beaulieu et. al, "The SIMON and SPECK Families of Lightweight Block Ciphers," Cryptology ePrint Archive, Report 2013/404, 2013.