Patching of ESC Challenges via Applied Research and Experimental Verification: Team Mouseion Finalist Paper

Siddhant Singh (Student)
Vertically Integrated Projects
Georgia Institute of Technology
ssingh484@gatech.edu

Cameron Newman (Student)
Vertically Integrated Projects
Georgia Institute of Technology
cnewman35@gatech.edu

Siddharth Suman (Student)
Vertically Integrated Projects
Georgia Institute of Technology
siddharthsuman@gatech.edu

Sheel Shah (Student)
Vertically Integrated Projects
Georgia Institute of Technology
sshah23@gatech.edu

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

***Abstract*—This document details the vulnerability mitigation patches that team Mouseion from the Georgia Institute of Technology implemented for all challenges of the CSAW ESC 2021 event as part of the research track. These challenges involved various vulnerabilities to Fault Injection and Side Channel Analysis based attacks which were analysed, mitigated and verified by the team. This involved examining existing research as well as applying generalized mitigation approaches for such vulnerabilities to each challenge program. These mitigation implementations were tested via power traces and fault injection attacks to compare them with the original challenges and empirically verify their effectiveness and efficiency.**

***Index Terms*—Fault Injection, Simple Power Analysis, Statistical Power Analysis, Binary Rewriting, Redundancy, operation shuffling**

I. Introduction

Fault Injection Attacks (FIAs) are exploits on devices that interfere with the outcome of the device or algorithm by physically tampering with the device. Various attacks can be carried out by either meddling with the device's environment, like its voltage or the amount of heat the system is exposed to, or using instruments, such as lasers to target logic gates or other circuitry. Side Channel Attacks (SCAs) rely on gathering information by observing hardware or low-level software processes and effects. SCAs are attacks that utilize side-effects of a target computational load and derive information from it, leaking cryptographic secrets, compromising confidential data or provide information for usage in other attacks. Our write-up focuses on power based FIAs and SCAs and their respective mitigation techniques for each of the 10 challenges for this year's ESC challenge. Each challenge presents itself as a new environment to unravel and understand. During this competition, our team was exposed to everything from encryption algorithms to mathematical theorems, pushing us to look at aspects beyond just code. By examining each challenge from both the perspective of an attacker and defender, our team was able to implement counters to possible attacks that could be conducted, ultimately allowing us to solve every challenge.

II. General Solution Approaches

This section summarizes our general approach to analysing, patching and verifying each challenge.

A. Challenge Set 1

1) fizzy: The fizzy challenge revolves around the bubble sort algorithm found in between the trigger high and the trigger low. A fault injection attack could be performed, skipping over the super efficient sort method. As a result, the passed in array would not be sorted, and the original array would be returned. To remedy this, we added a check in the super efficient sort method to check if at least 1 swap had been performed. If this check occurred, this means that no attack had occurred. Otherwise, if the check did not occur, the super efficient sort would be called again to make sure a sort occurs.

Another attack would be to perform power analysis on the swaps within the sort function. Because a power spike occurs at every iteration of the swap, an attack could analyze the power trace to deduce how many swaps occur and at what times. A mitigation to this would be to have a constant swap happening at periodic intervals to mask the power spikes of relevant swaps, preventing attackers from detecting the swaps of values in the array. This concept was implemented by creating a dummy array and performing a swap, either on the data array or on the dummy array, on every iteration of the nested loop. This constant power use masks the actual power usage caused by the legitimate swaps and prevents power analysis from yielding useful information for this challenge.

2) err0r: The err0r challenge is a CRC32 calculation between trigger high and trigger low where the attacker wins if the two Error Correcting Code (ECC) values calculated from the same input do not equal each other. As such, we observed that introducing a fault into one of the two CRC32 calculations would result in successfully exploiting the challenge. A clock or power glitching attack could skip over an instruction in the calculation of one of the two ECC values, causing them to mismatch at the end.

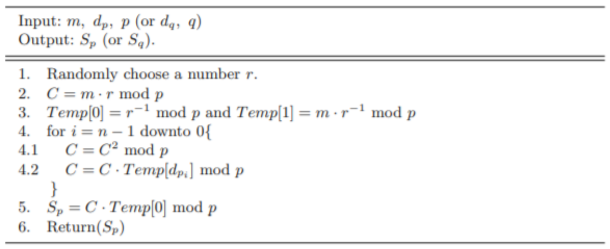
Hence, our goal was to make sure the ECC values always equal each other for the same input. To enable this, we calculated the ECC value a random additional number of times via the `rand()` function and stored them in a variable length array. We ensured that the total number of ECC values are odd to ensure that there is a mode among the values. By setting the final two ECC value variables equal to the mode of the array, we set the final ECC value variables to be the most likely accurate calculation and ensure their equality. The mode of the calculated values is used due to the assumption that a fault injection attack cannot occur during the calculation of every ECC value due to the random number of such calculations that occur at every execution of the code.

3) recall: The recall challenge revolves around a comparison loop found between the trigger high and trigger low. The comparison loop checks an unknown array with secret values to another array provided as an input. If each value at the same index between the two arrays matches then the flag is written to the simple serial output. However, if a single value between the two arrays is different then the loop breaks and a failure output is written to the simple serial output. As such, this program functions as a password checker comparing the input to a secret value in the `correct_mem` array.

As we observed from the power trace of the original source code, the check for each value between the two arrays has a different power trace when the values match and the loop continues as compared to a mismatch of values which immediately ends the loop. This difference can be used to conduct a power analysis based attack as it would leak which of the input array values did not match the correct value. Such an analysis would enable brute force enumeration of the correct input by working one value at a time over multiple executions of the program. As such, we mitigated this issue by leveraging the nature of boolean comparison in C where all positive values above 0 are considered true. As such, we introduced an unused `counter` variable and incremented either the `counter` or the `mem_different` variable for each comparison between the input array and the secret array. As such, the loop always runs for the same number of iterations and the `mem_different` variable serves the same function after the comparison loop. This prevents power analysis from revealing which iteration of the loop modifies the `mem_different` variable, preventing an attacker from deducing which input value matched or did not match the secret values in the `correct_mem` array.

4) CRT: The CRT challenge, as hinted at by the name, is an RSA signature calculation based on the Chinese Remainder Theorem (CRT). As such, the binary flashed onto the target board is used to calculate the two values that are used as inputs to the Chinese Remainder Theorem algorithm utilizing the target board as a co-processor in the cryptographic signing process. In doing so, the modular exponentiation function serves as a vector for power analysis in terms of how many times modular multiplication is carried out being dependent on the input values.

Research has shown that RSA optimized via CRT is vulnerable to both fault injection and power analysis attacks [1], [2]. As the recombination step of CRT is not present within the challenge binary, a fault injection attack would fall outside the scope of patching the binary as code changes will need to be implemented in the recombination step as well [1], [3], [4]. However, in order to mitigate the power analysis attack vector, message blinding exponentiation can be used [1]. To do so, we modified the modular exponentiation function to generate a random mask, perform message blinding exponentiation and finally removed the random mask to return the expected output. This modification was made by implementing the algorithm for exponentiation detailed in [3] and described in figure 1.



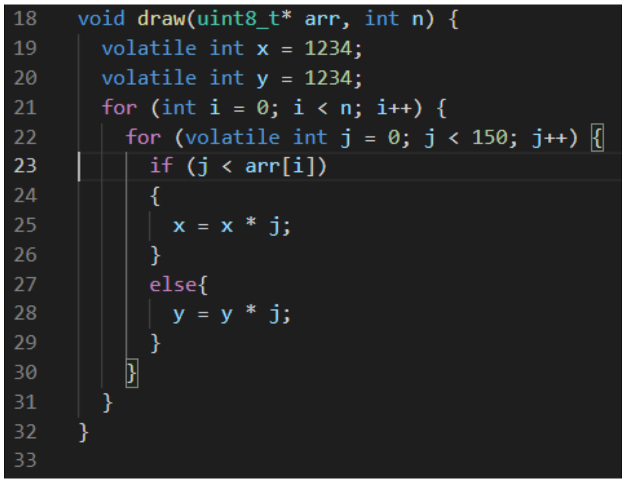
```
Input: $m,\ d_p,\ p$ (or $d_q,\ q$)
Output: $S_p$ (or $S_q$).

1.   Randomly choose a number $r$.
2.   $C = m \cdot r \bmod p$
3.   $Temp[0] = r^{-1} \bmod p$ and $Temp[1] = m \cdot r^{-1} \bmod p$
4.   for $i = n - 1$ downto 0{
4.1      $C = C^2 \bmod p$
4.2      $C = C \cdot Temp[d_{p_i}] \bmod p$
     }
5.   $S_p = C \cdot Temp[0] \bmod p$
6.   Return($S_p$)
```

Fig. 1. Exponentiation algorithm immune to DPA, SPA and Timing Attacks as detailed in [3].

B. Challenge Set 2

1) casino: Casino is a prime candidate for a power analysis attack. The draw function is vulnerable specifically, as it has varying iterations based on the number in the array. This would allow an attacker to determine the number at each index by the length of the related power spikes in the collected trace. Because the draw function's iterations are dependent on the number found in the array, the power draw will vary based on each number in the secret array. This would allow for a power analysis attack by reading the power draw graph to deduce the order of the known values within the secret array.

The mitigation technique implemented is found in the draw method which mitigates the vulnerability by finding the maximum number present in the array and setting that as the constant number of iterations for the nested inner loop. Because the max number is 150, the inner for-loop will always iterate 150 times, regardless of what is in the array. Additionally, a dummy variable is created, y, which is the same number as x. To replicate the $j < arr[i]$ functionality, an if is added in the inner for loop. If $j < arr[i]$, x is multiplied by j, like it originally was. However, once $j >= arr[i]$, y is multiplied by j, replicating the same power draw for the remaining $150 - arr[i]$ iterations. This hinders the attackers ability to analyze the power consumption because the draw



```c
void draw(uint8_t* arr, int n) {
  volatile int x = 1234;
  volatile int y = 1234;
  for (int i = 0; i < n; i++) {
    for (volatile int j = 0; j < 150; j++) {
      if (j < arr[i])
      {
        x = x * j;
      }
      else{
        y = y * j;
      }
    }
  }
}
```

Fig. 2. Mitigation technique implemented in draw method to generate a consistent power trace.

will be at a consistent level per iteration of the loop, masking the legitimate power consumption of the original functionality provided by the draw function.

Additionally, a slight vulnerability was found in the casino function. `simpleserial_put('r', n, arr)` was called after the draw function, which inadvertently leaks data. Taking the output of that function and providing it as input to the `verify` function gives you the flag allowing the attacker to retrieve the flag without performing any kind of SCA or FIA. This was patched by removing the line of code from the program.

2) FIAsco: The FIAsco challenge is an AES function that is vulnerable to Fault Injection Attacks (FIA) as hinted at by the name. Research has shown that a simple AES implementation is vulnerable to Fault injection as well as a variety of power analysis attacks [5] [6] [7]. In terms of FIA, inserting a fault within the AES encryption performed by any AES library included via the "independant.h" header file can yield faulty cipher texts for known plain texts [5], [6]. These faulty results can be used to derive the encryption key used for encryption. Power analysis can also be utilized independently as well as alongside Fault injection in order to compromise AES by narrowing the search space for bits of the AES key used to perform encryption [6].

As the AES encryption is implemented within any AES library that the challenge source code can be compiled with and due to the naming of the challenge hinting at Fault Injection, we concerned ourselves with mitigating Fault Injection Attacks. Such a mitigation can be done without modifying the underlying encryption implementation. However, mitigating power analysis would require significant modification or a re-implementation of the entire AES cryptosystem to implement masking as described in [8] and [5]. As such, the mitigation efforts for this challenge focused on implementing a randomness based fault detection and infection process before the encrypted cipher text is outputted [9]. To do so, between the trigger high and trigger low functions, each encryption function call was carried out twice. After two encryptions, the difference between the two cipher texts was calculated to isolate the injected fault and its effect on the encryption. This difference was then diffused via a diffusion function and magnified via a set of random values to infect the cipher text in a non-deterministic fashion. This diffusion yields a new cipher text that is the correct cipher text in the event of fault free execution and masks the effect of an injected fault in the case of an FIA [9]. This process is represented by figure 3 and implemented by us as described in figure 4. In figure 4 M represents a random number matrix of equivalent size to the AES state where $M \neq 0$. The resulting Γ is the diffusion result that is XORed with each of the two independent cipher texts of the same input, one of which is returned as the output of the existing encryption function from the original source.

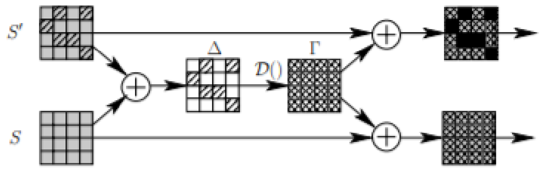


Fig. 3. Infection countermeasure against Fault Injection in AES as detailed in [5].

$$\begin{cases} \Delta'_{ij} = \bigoplus_{n=0}^{3} \Delta_{in} \oplus \bigoplus_{m=0}^{3} \Delta_{mj} \\ \Gamma = M \bullet \Delta'_{ij} \end{cases}$$

Fig. 4. Diffusion function for fault infection based on random values as detailed in [9].

3) search: The search challenge is about avoiding power analysis attacks during binary search. The program randomly deletes six values from the given array, and then uses binary search to find an input value within the remaining array. An attacker wins if they can successfully figure out which elements from the array are missing, which can be done by finding the difference in power spikes between binary search finding and not finding a value in the array.

The key to the patch is to mask the power trace of the binary search such that all instances of the binary search create the same number of power spikes regardless of input. To do this, we created a dummy copy of the array and the binary search function. The difference in binary search between a found and not found value is $log_2(n) - x$ power spikes, with n being the length of the array and x being the number of recursive calls to binary search. In this case, n is 249 to match that of the real binary search. Then we keep track of how many recursive calls binary search took to find or not find a value. If the value was not found, we call the dummy binary search that is equivalent to binary search in terms of power consumption, but only do so $log_2(n) - x$ times to add to the power spikes caused by the binary search function. This ensures that $log_2(n)$ power spikes are caused by every main call to the binary search function.

C. Challenge Set 3

1) NotSoAccessible: The NotSoAccessible challenge is a SIMON encryption function that is vulnerable to power analysis based reconstruction of the secret key due to partial knowledge of the key as found in the challenge source code. Research has shown that a simple SIMON implementation is vulnerable to power analysis attacks such as Differential Power Analysis [10], [11].

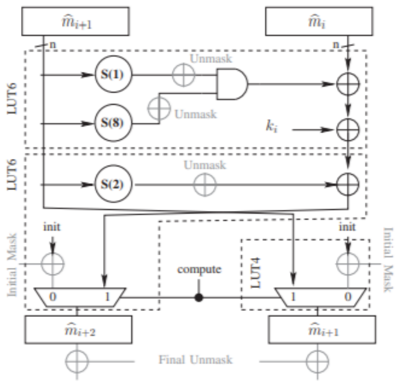


Fig. 5. First-order boolean masking based round function of SIMON based on random masks as detailed in [11]

As such, the mitigation efforts for this challenge focused on implementing a randomness based first-order boolean masking algorithm in the 25th round of the encryption. The 25th round of the encryption was used due to the `trigger_high` and `trigger_low` functions being present within this round only. The masking scheme utilized, which is empirically proven to be resistant to power analysis attacks with a high number of traces, is described in [11] is shown in figure 5.

2) calc: The calc challenge was a set of calculation functions that produce distinct power usage while computing based on the provided input on a copy of the secret array of values. As such, we observed that using multiple operations and collecting power traces for each of them may allow reverse engineering the values that the secret array started with. Hence, we found this challenge to be vulnerable to power analysis based deduction of secret values by modeling the functions and solving that model via power traces.

As such, to solve the challenge, we patched the source code such that every function performs the same computation while randomly shuffling the moment that the relevant operation for each function occurs within its execution. This was achieved by using multiple copies of the manipulated array and shuffling the order of independent operations via random number generation per function call.

3) homebrew: The homebrew challenge was a custom encryption function utilizing two S-boxes for substitution as well as a few key dependent operations for permutation. This was also hinted at by the title of the challenge referencing a "homebrewed" or custom solution to encryption. As such, by examining the implementation of this encryption system, we made a few observations. The S-boxes themselves may lead to possibly weak substitution values that would be relevant to a traditional statistical attack on the cryptosystem. However, the bits of the key and the conditional statement based on that bit are more relevant to side channel based attacks. As such, a 1 bit in the key leads to a more computationally intensive set of manipulations on the relevant byte of the plain text as compared to a 0 bit in the key. By observing this and by running a power trace via a compiled version of the source code using various dummy key values, we found that this challenge binary would be vulnerable to power analysis due to power spikes correlating to bits of the encryption key.

As such, to solve the challenge, we patched the source code such that both parts of the conditional are computed at every iteration of the loop and the plain text is only modified once via a simple value assignment. As this value assignment is conditional on the bit of the key, it is equivalent to the original source code in function. However, by executing the same set of computations at every iteration, regardless of the value of the key bits, we make the power traces of each iteration identical. As such, any differential or statistical power analysis on the encryption operation is prevented by masking the power usage and its dependency on the bits of the encryption key.

III. Experimental verification of Patches

A. Challenge Set 1

1) fizzy verification: In order to verify our patch for the fizzy challenge, we employed a Jupyter notebook that compiles, programs and interfaces with the original challenge source code as well as our patched source code on the Chipwhisperer Nano. The original source code is compiled and used to demonstrate the original power trace. By examining the original source code power trace, we can see varying spike and dips in power draw, caused by varying power usage. Such a power trace that enable identifying the bits of the sort function and the original, unsorted array. If we examine the power trace of our fizzy patch, we can see that the power trace is uniform as a function of the constantly swapping dummy array values used inside the sort function. The consistency of power draw is clearly visible in the rectangular, uniform shape of the graph. This effectively masks the legitimate power draw, deterring the attacker from performing power-analysis and retrieving the flag from the program. Additionally, we have shown that the functions produce identical outputs. The only difference is that our patch produces a power analysis resistant power trace.

As such, by comparing the power traces and outputs of both programs for the same input values, we have shown that the functions produce identical outputs but our patch produces a power analysis resistant power trace, verifying our solution.

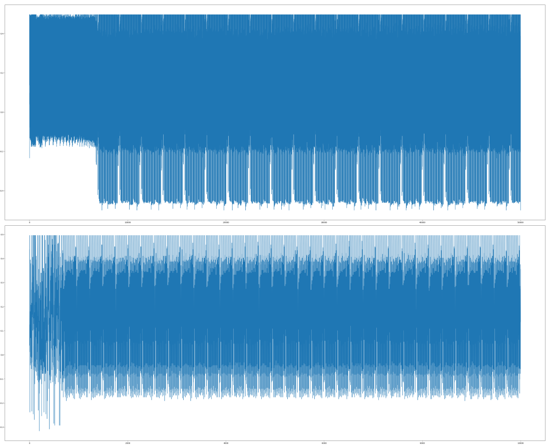


Fig. 6. **Top** - Power trace of original fizzy program. **Bottom** - Power trace of patched fizzy program.

2) err0r verification: The err0r challenge was verified with a Jupyter notebook by using it to compile and program the patched code in comparison to the original on the Chipwhisperer Nano. The code original code is first compiled with dummy values and a power trace was captured as a control to compare that of the patched code.

The goal was to make sure that faults that occur do not prevent the power traces of the original and patched code from matching, so the ECC values will always match despite any faults that occur. Even after forcing faults to occur with the patched code, the power trace still matches the original, showing the original, correct ECC values are output and any faults were unsuccessful.

3) recall verification: To verify the patch for the recall challenge, a Jupyter notebook was used to compile, program and interface with the original code compared to the patch on the Chipwhisperer Nano. The code is shown through a power trace graph. The varying power spikes and dips caused by the different usage of the code can be examined to identify when the comparison loop begins and ends.

Examining the power trace, it can be seen that in the original code source, there a portion where the usage is intensified and is uniformly being used. This clearly indicates that the comparison loop is currently in its process. The uniformity in the middle of the of the power trace easily enables the identify the bits of the private array. However, by examining the power trace of the patch, there no specific distinguish shown when the comparison loop is ran. There are large spikes and dips evenly spread, thus masking the execution and process of the function. As such, by comparing the power traces and outputs for the original, it produces identical outputs identical outputs. However, the patch produces a power analysis resistant power trace, thus verifying our solution.

4) CRT verification: In order to verify our patch for the CRT challenge, we employed a jupyter notebook that compiles, programs and interfaces with the original challenge source code as well as our patched source code on the Chipwhisperer Nano. This is done via two source code files, which both utilize the same private primes $p = 13$ and $q = 23$ as well as having the relevant dp and dq values. As such, for identical plain text inputs, running the programs should yield identical results via the value output on the simple serial interface.

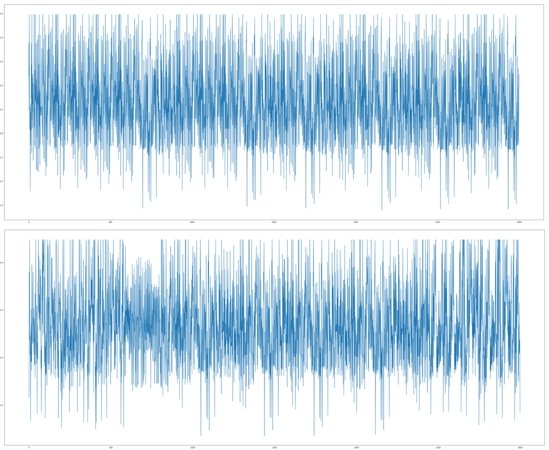


Fig. 7. **Top** - Power trace of original CRT program with $p = 13$, $q = 23$ and the relevant dp and dq values. **Bottom** - Power trace of patched CRT program with $p = 13$, $q = 23$ and the relevant dp and dq values.

The original source code was compiled and used to demonstrate how these specific private primes lead to a specific power trace for a specific input as shown in figure 7. The patched source code was also compiled and used to demonstrate how these specific private primes lead to a specific power trace for the same input as shown in figure 7 as well. By examining the original source code power trace, we can clearly distinguish that the original challenge code produced a power trace that enables identifying the bits of the private exponent used within the modular exponentiation function. However, by examining the power trace produced by our patch source code, we can observe that the power trace is uniform as a function of the random masking value used for both executions of the modular exponentiation function. As such, by comparing the power traces and outputs of both programs for the same private primes and the same input, we have shown that the functions produce identical outputs but our patch produces a power-analysis-resistant power trace. This observation coupled with the empirical evaluations done by the authors of [3] provide our verification for this challenge solution.

B. Challenge Set 2

1) casino verification: A Jupyter notebook on the Chipwhisperer Nano was used to verify the patch for the casino challenge by compiling, programming, and comparing the power traces of the original code to those of the patched code. We initially compile the code with known values. In the case that values are variable, we use dummy values. For this challenge, attacks would have to retrieve the flag through examining the power trace. The initial power trace graph shown is the power trace of the unpatched casino file, and upon analysis, the flag is retrieved.

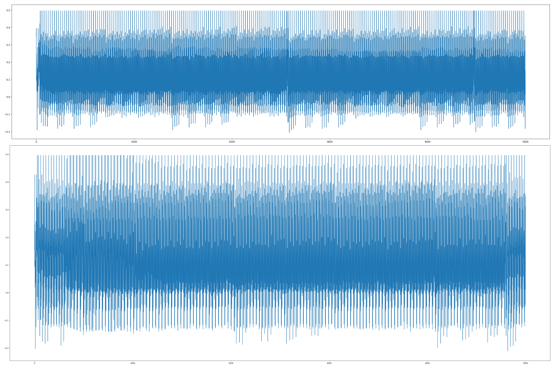


Fig. 8. **Top** - Power trace of original casino program. **Bottom** - Power trace of patched casino program.

We verify our patch by setting our casino patch as the testing board. The power trace shown from running the casino patch is shown above. There are significantly more peaks present in the power trace as well as more consistency, likely from the alternate implementation of the draw function, which runs 150 iterations for each element in the array. The consistency effectively masks the true power trace of the original unpatched code, making it significantly more difficult for attacks to do power analysis. Additionally, upon running the program to attempt to retrieve the flag, we are met with ”r00”, which is the incorrect flag, verifying that our patch works.

2) FIAsco verification: In order to verify our patch for the FIAsco challenge, we employed a jupyter notebook that compiles, programs and interfaces with the original challenge source code as well as our patched source code on the Chipwhisperer Nano. This is done via two source code files, which both utilize the same TinyAes128 library as well as the same AES key value of $0xdeadbeef$. As such, for identical plain text inputs, running the programs should yield identical cipher texts as well as using identical AES libraries for computation.

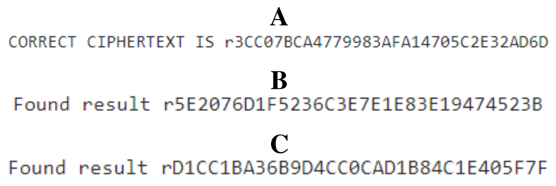

A
```
CORRECT CIPHERTEXT IS r3CC07BCA4779983AFA14705C2E32AD6D
```
B
```
Found result r5E2076D1F5236C3E7E1E83E19474523B
```
C
```
Found result rD1CC1BA36B9D4CC0CAD1B84C1E405F7F
```

Fig. 9. **A** - Fault Free output from both the original and patched FIAsco programs for a specific key and plain text. **B** - Fault injection based output from the original FIAsco program for a specific key and plain text. **C** - Fault injection based output from the patched FIAsco program for a specific key and plain text.

We collected power traces of both program executions to analyze them as well as collecting a fault free output of both programs for the same plain text proving them to be functionally identical as shown in figure 9. The original source code was compiled and used to demonstrate how the specific key value and plain text along with an injected fault during the encryption process leads to a specific kind of faulty output as shown in figure 9. The patched source code was also compiled and used to demonstrate how the same key value and plain text along with an injected fault during the encryption process leads to a specific kind of faulty output as shown in figure 9 as well. By comparing the original and patched program power traces over the same number of samples, we can observe that they provide identical power traces for execution of the same AES library functions. However, by examining the power trace produced by our patch source code during its entire execution, we can observe that the patched program executes the same encryption twice and then produces a variation in the power trace while executing the randomness based fault diffusion function. As such, by observing how a fault propagates within the same plain text during encryption between the two program as well as their respective power traces, we can see that the fault is diffused based on randomness in our patched program. This observation coupled with the empirical evaluations done by the authors of [9] provide our verification for this challenge solution.

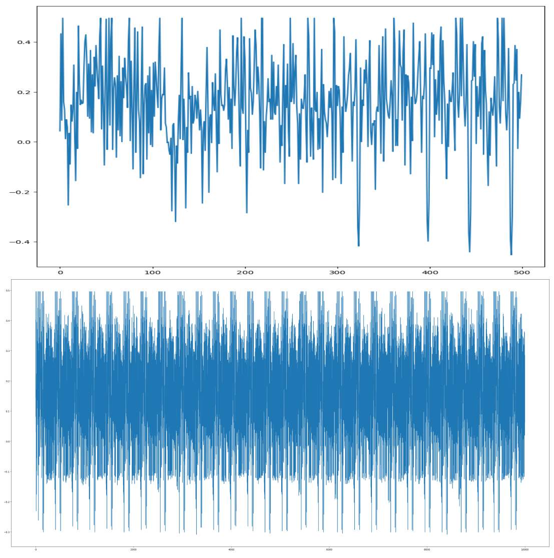


Fig. 10. **Top** - Power trace graph of original search program. **Bottom** - Power trace graph of patched search program.

3) search verification: A Jupyter notebook on the Chipwhisperer Nano was used to verify the patch for the search challenge by compiling, programming, and comparing the power traces of the original code to those of the patched code. Since the hackers win if they correctly guess which values were removed from the array, they can perform a power analysis attack and find the outlying dips and spikes in the power trace graph and figure out the values. Therefore, the goal was to remove the odd dips and spikes and make a uniform power trace graph no matter if the value was found or not.

As shown in figure 10, the power trace graph of the original code has distinct dips and spikes that give away the values when they are not found by the search function. The bottom power trace graph in figure 10 is of the patched code after various faults occurred. The power trace of the patched code is very uniform, especially when compared to the original code, demonstrating that there are no dips or spikes to give away a value that is not found in the array. Ultimately, figure 10 shows that the patched code is successful in hiding the abnormalities in the original code's power trace that reveals the removed values in the array.

C. Challenge Set 3

D. NotSoAccessible verification

In order to verify our patch for the NotSoAccessible challenge, we employed a Jupyter notebook that compiles, programs and interfaces with the original challenge source code as well as our patched source code on the Chipwhisperer Nano. This is done via two source code files, which both use the same dummy key to encrypt supplied plain text. As such, for identical plain text inputs, running the programs should yield identical cipher texts computed using the same key.

We collected power traces of both program executions to analyze them by running them with the same input plain text to get the same output cipher text. By comparing the obtained power traces, we observed a noticeable change in the power trace that showcases the 25th round of the encryption using the boolean masking scheme. As such, by comparing the power traces and outputs of both programs for the same encryption key and the same input, we have shown that the functions produce identical outputs but our patch produces a power-analysis-resistant power trace. This observation coupled with the empirical evaluations done by the authors of [11] provide our verification for this challenge solution.

E. calc verification

In order to verify our patch for the calc challenge, we employed a Jupyter notebook that compiles, programs and interfaces with the original challenge source code as well as our patched source code on the Chipwhisperer Nano. This is done via two source code files, which both use the same dummy key to seed the Random Number Generator via `srand()`. As such, for identical input scalars, running the challenge program should yield different power traces per operation. However for our patched program, for identical input scalars, the power traces for all operations should be equal and distinct from the challenge program traces due to the injected randomness.

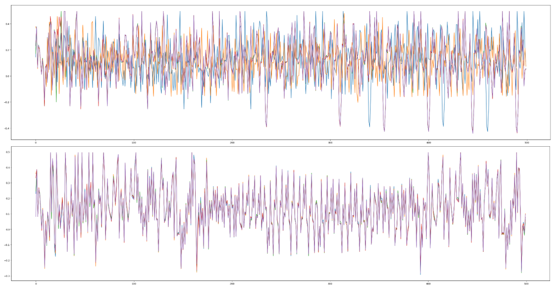
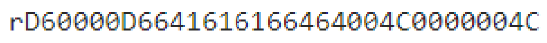


Fig. 11. **Top** - Power trace graphs of original calc program for all commands except verify. **Bottom** - Power trace graphs of patched calc program for all commands except verify.

We collected power traces of both program executions to analyze them by running them with the same input scalar from the initial program state for all operations. We then overlaid these power traces for each operation on the same graph to demonstrate the differences in power trace by operation executed as shown in figure 11. By comparing the obtained power traces, we observed that while the challenge program had distinct power traces, the patched program had an identical power trace for each operation. As such, we have shown that at the cost of a constant increase in computation, our patch produces power-analysis-resistant identical power traces for all commands.

1) homebrew verification: In order to verify our patch for the homebrew challenge, we employed a Jupyter notebook that compiles, programs and interfaces with the original challenge source code as well as our patched source code on the Chipwhisperer Nano. This is done via two source code files, which both use the same key to encrypt supplied plain text. As such, for identical plain text inputs, running the programs should yield identical cipher texts computed using the same key.

We collected power traces of both program executions to analyze them by running them with the same input plain text to get the same output cipher text as shown in figure 12. The original source code was compiled and used to demonstrate how a key led to a specific power trace for a specific input as shown in figure 13. The patched source code was also compiled and used to demonstrate how the same key led to a specific power trace for the same input as shown in figure 13 as well. By examining the original source code power trace, we can observe that there is a difference in the power trace for sections using a 1 bit from the encryption key as opposed to using a 0 bit from the encryption key. This information allows an attacker to correlate power spike patterns to bits of the encryption key, leveraging this to leak the encryption key in the process. However, by examining the power trace produced by our patch source code, we can observe that the power trace is uniform as both calculations for a 1 or 0 bit in the encryption key are carried out every time. As such, by comparing the power traces and outputs of both programs for the same encryption key and the same input, we have shown that the functions produce identical outputs but our patch produces a power-analysis-resistant power trace.



```
rD60000D6641616166464004C0000004C
```

Fig. 12. Cipher text output from both the original and patched Homebrew programs for key $1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2$ and a specific plain text.

IV. Limitations

Throughout this competition, our team experienced multiple issues and obstacles.

One big issue was that the Chip Whisperer Nano software would not download the gcc package on one of our computers, so we could not use that computer to compile code with

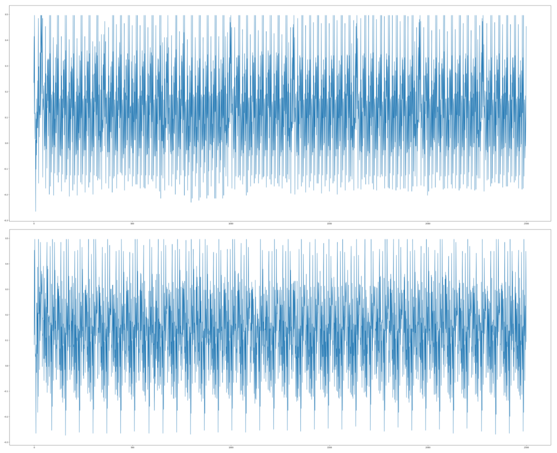


Fig. 13. **Top** - Power trace of original Homebrew program with key $1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2$ and a specific input plain text. **Bottom** - Power trace of patched Homebrew program with key $1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2$ and a specific input plain text.

the Jupyter notebooks. The issue was remedied by sharing snapshots of Jupyter notebooks through either HTML files or images. This allowed those without access to the Jupyter notebook to continue making contributions and discussing flaws in the verification processes with the team.

Additionally, the ChipWhisperer software had missing sections and folder content. For instance, there were no tutorials present in the tutorial folder. Had there been tutorials present, it may have been easier for some of our team members to understand the possible attacks and implement patches.

The limitation for the err0r challenge is that the patch is not the most efficient solution, as it repeats the ECC calculation multiple times and forces the CRC32 error correcting codes to equal each other via the introduced redundancy.

The fizzy challenge had multiple failed approaches to defend against possible attacks. One prominent compilation error occurred because the checks were being defined using rand(), a function that occurs during runtime. This was fixed simply by defining both checks as 0 and reassigning them to rand(), later in the program.

The casino challenge was run under the assumption that the `simpleserial_write` function was intended by CSAW to be patched and now just a typo in the original code. As explained in the casino write-up and verification, the `simpleserial_write` function occurring after the draw function gave out the key wouldn't needing any kind of SCA or FIA to be performed. This was patched by removing that line of code from the patched casino file.

Finally, implementing the use of `srand()` and `rand()` functions within our patches to introduce randomness does not allow for a good seed value via the `time()` function due to the target board. As such, a true random seed for effective random value generation was not available to use in our patches.

V. Conclusion

In this paper, we detailed how we as a team approached each of the three challenge sets for the CSAW ESC 2021 competition. As such, we described how we reasoned about each challenge and our approach to mitigating the vulnerabilities in each source code. We showed how we utilized source code analysis, dynamic analysis via power tracing as well as various research work to inform our mitigation approaches. In patching each source code, we focused on writing efficient mitigation code that would not be optimized during a compiler optimization process which would negatively impact the expected results of our work. Further, we described our verification of each challenge patch via the ChipWhisperer tool chain. We used this tool chain to compile patches, program the ChipWhisperer Nano target board as well as run power traces and fault injections to empirically verify our mitigation implementations. In this way, we solved each challenge by going through an analysis, patching and verification process as detailed in this paper.

VI. Acknowledgements

We would like to thank our team advisor Allen Stewart for his guidance and valuable feedback on our final report.

References

[1] J. Ha, C. Jun, J. Park, S. Moon, and C. Kim, "A new crt-rsa scheme resistant to power analysis and fault attacks," *Convergence Information Technology, International Conference on*, vol. 2, pp. 351–356, 11 2008.

[2] P. Luo, L. Zhang, Y. Fei, and A. A. Ding, "Towards secure cryptographic software implementation against side-channel power analysis attacks," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015, pp. 144–148.

[3] C. Kim, J. Ha, S.-H. Kim, S. Kim, S.-M. Yen, and S. Moon, "A secure and practical crt-based rsa to resist side channel attacks," in *Computational Science and Its Applications – ICCSA 2004*, A. Laganá, M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, and O. Gervasi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 150–158.

[4] H. Mamiya, A. Miyaji, and H. Morimoto, "Efficient countermeasures against rpa, dpa, and spa," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, M. Joye and J.-J. Quisquater, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 343–356.

[5] V. Lomné, T. Roche, and A. Thillard, "On the need of randomness in fault attack countermeasures - application to aes," in *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2012, pp. 85–94.

[6] T. Roche, V. Lomné, and K. Khalfallah, "Combined fault and side-channel attack on protected implementations of aes," in *Smart Card Research and Advanced Applications*, E. Prouff, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 65–83.

[7] J. Park, "Differential fault analysis for round-reduced aes by fault injection," *ETRI Journal*, vol. 33, no. 3, p. 434–442, 2011.

[8] H. Mestiri, F. Kahri, B. Bouallegue, and M. Machhout, "A high-speed aes design resistant to fault injection attacks," *Microprocessors and Microsystems*, vol. 41, pp. 47–55, 03 2016.

[9] J. Zhang, N. Wu, X. Zhang, and F. Zhou, "Against fault attacks based on random infection mechanism," *IEICE Electronics Express*, vol. 13, no. 21, pp. 20 160 872–20 160 872, 2016.

[10] D. Shanmugam, R. Selvam, and S. Annadurai, "Differential power analysis attack on simon and led block ciphers," in *Security, Privacy, and Applied Cryptography Engineering*, R. S. Chakraborty, V. Matyas, and P. Schaumont, Eds. Cham: Springer International Publishing, 2014, pp. 110–125.

[11] S. Bhasin, T. Graba, J.-L. Danger, and Z. Najm, "A look into simon from a side-channel perspective," in *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2014, pp. 56–59.