Analysing Chacha20poly1305 Energy Usage Using Chip Whisperer

|  |  |  |  |
| --- | --- | --- | --- |
| Kushkumar Hetalbhai Patel  *Department of System and Computer Engineering*   Carleton University  Ottawa,Canada  kushkumarhetalbhaipa@cmail.carleton.ca | Vishnu Priya  *Department of Electronics*   Carleton University  Ottawa,Canada   nisargjagdishkumarpa@cmail.carleton.ca | *Rasek Thangam ATS*  *Department of System and Computer Engineering*   Carleton University  Ottawa,Canada   nisargjagdishkumarpa@cmail.carleton.ca |  |

*Abstract*—Using an embedded ARM Cortex-M0 architecture, this study examines the energy consumption of the ChaCha20-Poly1305 cryptographic algorithm with an emphasis on key generation, encryption, decryption, signature, and verification. Known Answer Tests (KAT) are used to verify the algorithm's correctness once it has been implemented and compiled on the ChipWhisperer Nano board. After evaluating energy use over 100 runs with random inputs, the standard deviation is investigated. While, running the code which we got from git, we mapped 0.3073uJ energy consumption. After that We explore optimization options, such as various implementations and algorithmic tweaks, to conserve energy while maintaining functionality. According to the findings, energy consumption may be considerably reduced with the right implementation choices, after optimizing the code we found 0.3579% description in energy consumption. Through its insights into the design of energy-efficient cryptography algorithms for embedded devices, this study advances secure communication applications.

Keywords—Cryptographic algorithms, embedded systems, energy consumption optimization, security, ARM Cortex-M0, ChaCha20-Poly1305, ChipWhisperer Nano, side-channel attacks, IoT applications.

# Introduction

Almost every facet of contemporary life includes embedded technology these days. They operate anything from medical implants to sophisticated sensors. As these devices become more and more integrated into essential infrastructure and household appliances, it is imperative to ensure the security and efficiency of cryptographic algorithms. The ChaCha20-Poly1305 algorithm is the most well-known cryptographic primitive because of its robust security features and superior efficiency [1][2]. However, because embedded systems are widely used, they often have to function below very strict power constraints; hence, cryptographic algorithms must optimize energy efficiency. In answer to this challenge, this paper investigates the energy needs of the ChaCha20-Poly1305 algorithm when implemented on an integrated ARM Cortex-M0 platform using the capabilities of the ChipWhisperer Nano board [3][4]. Our objective is to investigate the energy consumption patterns of essential cryptographic processes, such as key generation, encryption, decryption, signature, and verification, in order to uncover strategies to improve energy usage without sacrificing the security guarantees of the algorithm.

Through a rigorous analysis of the energy overhead associated with each cryptographic operation, our work seeks to identify potential optimization strategies that strike a compromise between energy efficiency and cryptographic resilience. Through testing and improvements across a range of implementation options, including hardware configurations and algorithmic changes, we want to shed light on energy-efficient implementations particularly made for ChaCha20-Poly1305 on embedded devices [1][2]. By offering practical guidance for the development of secure and energy-efficient cryptographic solutions for embedded systems, we hope to bridge the gap between energy efficiency and cryptographic security. Ultimately, we want to push the limits of energy-efficient encryption and increase the resilience of modern digital ecosystems against new security threats, therefore encouraging the proliferation of secure and resource-conscious embedded applications.

# Background

Embedded systems are the foundation of many industries in the age of digital transformation. They enable essential applications such as industrial control systems, driverless cars, wearable health monitors, and smart homes. These systems are unique in that they must consistently complete difficult tasks, frequently within size and power limitations. Due to the Internet of Things (IoT), these devices are becoming more and more linked, which exposes them to a variety of security risks that have the potential to breach user privacy, interfere with operations, and jeopardize critical data. Even though they provide the required security protections, standard cryptographic methods sometimes surpass the computational and power constraints of embedded systems, increasing prices and shortening device lifespans owing to energy depletion. The necessity for cryptographic algorithms that strike a balance between strong security and low energy consumption is highlighted by this scenario. In this regard, the ChaCha20-Poly1305 algorithm, which is renowned for its robust security features and computational efficiency, shows promise. Nevertheless, despite its benefits, nothing is known about the precise energy consumption patterns of ChaCha20-Poly1305 in embedded systems. This lack of understanding impedes the creation of efficient cryptographic solutions, which might reduce energy consumption even further without sacrificing security. This is a crucial development for the long-term sustainability and financial sustainability of embedded systems in a variety of sectors.

## Selecting ChipWhisperer Nano Board

The NewAE Technologies NAE-CWNANO ChipWhisperer-Nano Board provides an inexpensive platform for side-channel power analysis attacks [5]. More tutorials and demos may be performed with this device, although its capability is not as extensive as that of the ChipWhisperer-Lite or Pro [6]. By using synchronous sampling, it ensures accurate real-time power analysis of algorithms and devices at a fair price. The built-in target includes an STM32F030F4P6 microcontroller with 16KB of FLASH and 4KB of SRAM. To enable the onboard ADC to be utilized for measuring external devices, the device may also be modified to attach to external targets, such as the CW308 baseboard.

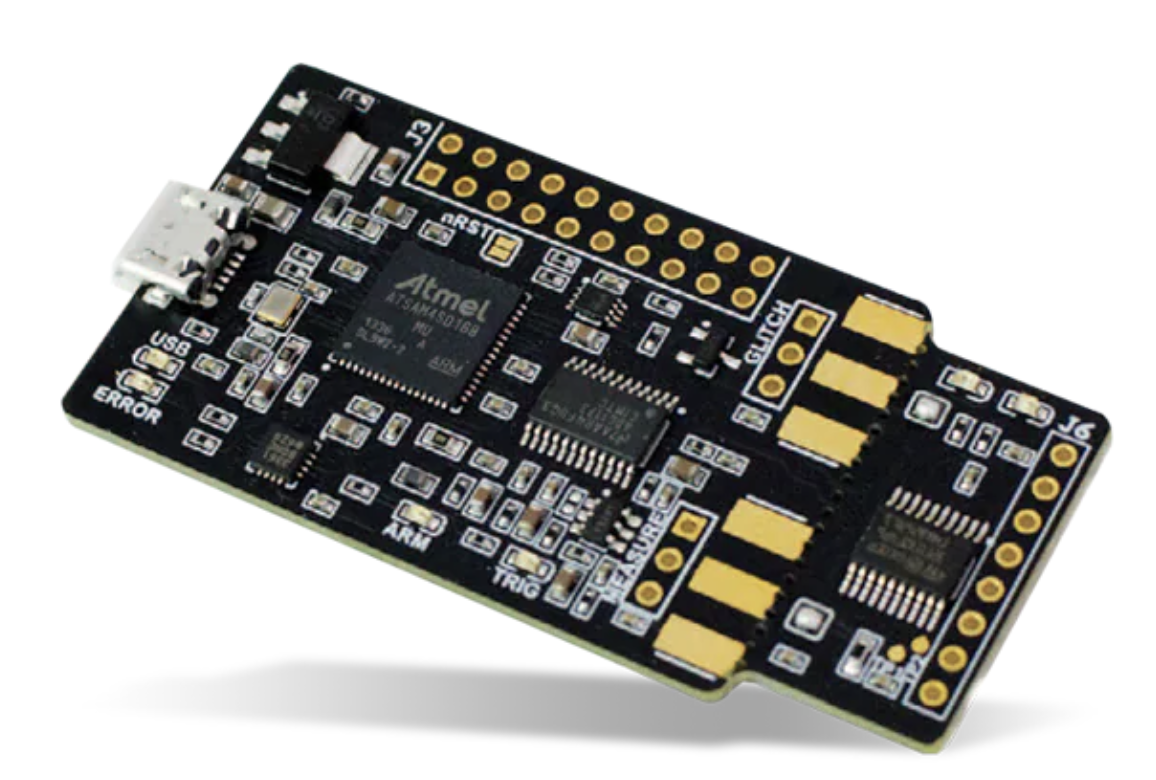


Figure 1. ChipWhisperer Nano Board [4]

Some of the important features of the ChipWhisperer-Nano include an 8-bit ADC with a maximum sample rate of 20MS/s, configurable ADC sample clock sources (internal generator or external input), an AC-coupled fixed-gain analogue input, and a range of GPIO types, including serial, clock, and logic line. Its functioning GPIO voltage range is 3.3V, and its clock possibilities span from 60MHz to 3.75MHz. The device supports both ADC and glitch trigger types, with a minimum glitch width of around 20 nS and an adjustable jitter glitch offset of about 200 nS. It also has a low-power crowbar circuitry voltage glitch type and a 4A crowbar pulse current. The ChipWhisperer-Nano may connect to target devices with STM32Fx bootloader programming, STM32F030F4P6, or STM32F070 by using customized USB firmware. It is also offers 50000-sample buffer size.

## Jupyter Notebook

A flexible, open-source online application called Jupyter Notebook is made for writing and sharing documents that include equations, live code, narrative prose, and visualizations. Its cell-based architecture makes interactive computing easier and allows code to be executed and iterated on the fly along with rich media outputs including LaTeX equations, charts, and graphics [7]. This environment allows for the smooth integration of executable code and explanatory text by supporting rich documentation using markdown integration. In addition to sharing their notebooks via email, GitHub, or the Jupyter Notebook Viewer for collaborative work, users may benefit from step-by-step code execution for in-depth examination [8]. Additionally, thanks to kernel support, it can be used with a variety of programming languages, which makes it perfect for a broad range of applications, including data analysis and teaching. Utilizing Python for thorough data collection, analysis, and visualization, Jupyter Notebook's integration with hardware such as the ChipWhisperer Nano board allows for in-depth fault injection and side-channel analysis on embedded systems, improving research reproducibility and dissemination.

# Design And Structure Of Chacha20-Poly1305

Before The ChaCha20-Poly1305 algorithm is a modern cryptographic scheme that combines two well-known primitives: the ChaCha20 stream cipher and the Poly1305 authenticator [1]. Authenticated encryption, which provides data transit secrecy and integrity, is its primary use.

The employed algorithms and the AEAD architecture are covered in the subsections that follow.

## The Chacha Quarter Round

The ChaCha quarter round is a crucial part of the ChaCha stream cipher algorithm. It is utilized to combine the internal state of ChaCha, which is necessary to generate the keystream required for encryption and decryption. In the quarter round, there are four 32-bit words that are represented by the letters a, b, c, and d.

Every word in the quarter round is added modulo 2^32 along with the subsequent word in the sequence, and each word is rotated to the left by adding a certain amount of bits.

A is rotated 7 bits to the left.

There is a 9-bit leftward rotation on c.

B rotates 13 bits to the left.

D is 18 bits that have been left-rotated.

After the addition and rotation operations, the quarter round produces four new values for a, b, c and d.

A diagram of a computer code

Description automatically generated

Figure 2. ChaCha20poly1305 Algorithm [1]

## A Quarter Round on Chacha State

In the state matrix, a quarter round is a simple operation on four integers or 32-bit words. The state matrix is composed of these sixteen words arranged in a 4x4 grid. The goal of changing every word in a quarter-round operation is to disperse and confuse the state and eventually contribute to the development of the keystream. Bit rotations and addition modulos are performed as described in the previous round. After these steps, four new 32-bit words are generated by the quarter round and added to the state matrix to replace the original words. This process is repeatedly carried out in ChaCha, with a different word participating in each round. This contributes to the creation of the keystream needed for encryption and decryption [9].

## The Chacha20 Block Function

The The ChaCha20 block function is the fundamental working element of the ChaCha20 stream cipher algorithm. In order to retrieve the plaintext during decryption, it is responsible for generating a pseudorandom keystream that is XORed with either the received ciphertext or the plaintext itself. During encryption, this procedure generates ciphertext.

The ChaCha20 block function initializes the 512-bit state it acts on with:

A 256-bit key, divided into eight words of 32 bits each.  
A 96-bit nonce is composed of three 32-bit words [10]. A 32-bit block counter.

The input state is copied to form a working state, which is dynamic as the block function runs. The block function consists of twenty mixing cycles. In every round, the functioning condition is subjected to a quarter round operation. Four quarter round operations are carried out in each round.

Following the completion of the 20 mixing cycles, the state's block counter component is increased by 1. The keystream is created using the state that is left over after the 20 cycles. Apart from raising the block counter, the state's nonce component also gains one increment. The generated keystream is XORed with the plaintext during the encryption process to produce ciphertext, and it is XORed with the received ciphertext to obtain the plaintext during the decryption process.

In order to effectively offer a robust pseudorandom keystream and preserve security against cryptographic attacks like differential and linear cryptanalysis, the ChaCha20 block function was created [10]. It serves as the basis for the ChaCha20 stream cipher, which is extensively employed in a variety of applications needing secure communication and data transfer.

## The ChaCha20 Encryption Algorithm

A symmetric stream cipher, the ChaCha20 encryption method is made to ensure data transfer is both secret and secure.

For ChaCha20, a 96-bit nonce (number used once) and a 256-bit secret key are required. The ChaCha20 state is initialized by combining the constant, key, and nonce variables that specify the behaviour of the algorithm. While the 96-bit nonce is made up of three 32-bit words, the 256-bit key is made up of eight 32-bit words. The 512-bit ChaCha20 state is initialized using these parts and constant values. Every block in the plaintext data encrypted by ChaCha20 is 64 bytes (512 bits) in size. Working with the current state, the block function creates a keystream that results in a 512-bit block of pseudorandom data.

For integrity protection in encrypted communication, it is customary to use an additional authentication technique, such Poly1305, in addition to ChaCha20. Encryption and decryption are synonymous in ChaCha20. Using the same key and nonce, the ciphertext is XORed with the keystream to recover the original plaintext. To prevent nonce duplication, which might compromise security, it's imperative to use a unique nonce for each encryption session.   
The key needs to be kept secret and shared only with authorized individuals.

The ChaCha20 encryption technique balances security, efficiency, and simplicity, making it suitable for a wide range of applications requiring secure data transfer across networks. Its architecture ensures quick encryption and decryption while providing robust protection against cryptographic threats.

## The Poly1305 Algorithm

The Poly1305 algorithm, a cryptographic message authentication code (MAC), was developed by Daniel J. Bernstein. Ensuring the correctness and integrity of supplied data is its primary goal. Combining Poly1305 with encryption algorithms like ChaCha20 or AES is a common way to develop authenticated encryption systems like AES-GCM or ChaCha20-Poly1305 [10].

Poly1305 requires both a message and a secret key in order to authenticate. It is recommended to use a 256-bit (32-byte) secret key. Poly1305 uses extra keying material from the secret key to create a one-time key for each block. Using this derived key, a polynomial authenticator is constructed for each data block. For each communication block, Poly1305 generates a 16-byte authenticator, or tag, using polynomial multiplication. The authenticator is calculated by multiplying a polynomial modulo 2^130 – 5.

Subsequently, a comparison is done between the calculated and received tags. If they match, it is assumed that the communication is sincere and original.

## Generating the Poly1305 Key Using ChaCha20

To produce the Poly1305 key using ChaCha20, first create a keystream using the stream cipher ChaCha20 and use it to build the key. One well-known authenticated encryption technique that makes advantage of this feature is ChaCha20-Poly1305.

Commence with a 256-bit secret key, which is typically generated securely for cryptography using a random number generator. To initialize the ChaCha20 state, use the secret key and the IV created in the previous stages. Use the ChaCha20 algorithm to create a keystream.

ChaCha20 generates a pseudorandom stream of bytes by repeatedly applying its basic quarter round operation on the state. To generate the Poly1305 key, take a section out of the ChaCha20 keystream. Typically, the Poly1305 key is a 256-bit (32-byte) prefix of the keystream. Then, the Poly1305 authenticator for the message—also referred to as the tag—is created using the plaintext and the generated Poly1305 key. The Poly1305 authenticator verifies the integrity of the encrypted communication and ensures it hasn't been tampered with. In the end, the ciphertext is produced by XORing the plaintext and the ChaCha20 keystream.

Next, the ciphertext and the Poly1305 authenticator are sent to the receiver. After receiving the ciphertext and the Poly1305 authenticator, the recipient follows the identical process to determine the Poly1305 key. Subsequently, the received Poly1305 authenticator is computed using the derived key and compared to the received authenticator. When they match, the communication is seen as authentic and unaffected.

# Experimental Setup

## Code Selection and Compilation

First, we choose the firmware or code to be examined using the ChipWhisperer platform. This might include cryptographic algorithm implementations or firmware for embedded systems that is thought to be vulnerable to side-channel attacks. Once the target platform has been chosen, the code is produced using the suitable compiler to ensure that it is prepared for analysis [10].

## Hex File Conversion

After a successful compilation, the binary file is converted to a hexadecimal (hex) file format. This format presents machine code instructions in a human-readable hexadecimal format, making them easier to understand. Depending on the compiler and development environment being used, either built-in tools or third-party utilities/scripts may be used for this conversion.

## Hex File Addition to Target board

Next, the generated hex file is put on the target board of the ChipWhisperer platform. Using a programming tool or interface, this phase often entails flashing the hex file onto the target's non-volatile memory (such as flash memory). The detailed instructions supplied by the target board's manufacturer or the literature specific to the hardware platform serve as a reference for correctly programming the hex file onto the board.

## Capturing Power Traces

After everything is configured, the code is executed by the target board, and power traces are being recorded by the ChipWhisperer platform. On the target board, the code is run simultaneously to perform intriguing calculations or cryptographic procedures. As the code executes, ChipWhisperer gathers data on the target board's power consumption and notes variations brought about by different instructions or activities [4].

## Capturing Power Traces

The collected power traces are examined following the completion of the capture process. To go through the power traces and identify any potential weak points or sensitive data breaches, techniques such as differential power analysis (DPA), correlation power analysis (CPA), and template attacks are employed. The examination and graphical depiction of the results provide significant novel insights into the security posture of the firmware or target code.

# Sequrity Analysis

## Strength of Encryption

The cryptographic properties of its underlying algorithms—ChaCha20 for encryption and Poly1305 for message authentication are the main source of the encryption strength provided by ChaCha20-Poly1305.

The ChaCha20 encryption algorithm is symmetric and depends on creating a stream cipher. It offers excellent security and is resistant to brute-force assaults thanks to its 256-bit key size. ChaCha20 creates a keystream using 64-byte blocks, which is XORed with the plaintext to produce the ciphertext. Its design aims on high speed and excellent diffusion, which ensures robust security against popular cryptographic attacks.

### Poly1305 Authentication: The Poly1305 message authentication code (MAC) approach guarantees the authenticity and integrity of the encrypted data. This fixed-size block operating system generates each data block using a 128-bit authenticator [11]. Poly1305's design is characterized by strong assurances against manipulation and forgery, as well as high security.

### Coupled Security: When coupled, ChaCha20 and Poly1305 provide strong security assurances. Data secrecy is ensured by ChaCha20's encryption, while the ciphertext's integrity is guaranteed by Poly1305's authentication [2]. This combination offers strong defence against a variety of cryptographic attacks, including chosen-plaintext, chosen-ciphertext, and ciphertext-only attacks.

All things considered, TLS (Transport Layer Security) and VPN (Virtual Private Network) implementations, which need confidentiality, integrity, and authenticity, are common uses for ChaCha20-Poly1305 due to its robust encryption [1][2].

## Resistance To Attacks

ChaCha20-Poly1305 is widely recognized for its robust resilience to several cryptographic attacks, ensuring the integrity, confidentiality, and validity of encrypted data. The ChaCha20 encryption technique has a 256-bit key size, making it very resistant to brute-force assaults. Moreover, the architecture of the stream cipher ensures the robust dispersion and prevention of known-plaintext and chosen-ciphertext attacks. The Poly1305 message authentication code adds more security by verifying the integrity of the ciphertext, protecting against forging and tampering attempts. When coupled, ChaCha20 and Poly1305 offer a robust defence against a variety of cryptographic threats, including differential and linear cryptanalysis, chosen-ciphertext attacks, and side-channel attacks.

## Vulnerabilites to Side Channel Attack

While ChaCha20-Poly1305 is widely recognized for its ability to withstand traditional cryptographic attacks, it can still be vulnerable to side-channel attacks, which capitalize on inadvertent information leaks from the algorithm's physical implementation. Some potential side-channel attack locations in ChaCha20-Poly1305 are as follows:

### Timing Attacks: Variations in the execution time of cryptographic operations can disclose information about the plaintext or secret key. By timing the execution of authentication or encryption processes, an attacker can learn specifics about the internal operations of the cryptographic technology or the data being processed.

### Attacks Using Power Analysis: Electromagnetic emissions generated during cryptographic processes can provide information about the internal state of an algorithm. An attacker can infer information about the plaintext or secret key by tracking the electromagnetic radiation that the device emits during authentication or encryption.

### Attacks Employing Electromagnetic Analysis: Electromagnetic emissions generated during cryptographic processes can provide information about the internal state of an algorithm. An attacker can infer information about the plaintext or secret key by tracking the electromagnetic radiation that the device emits during authentication or encryption.

### Cache-Based Attacks: Specific patterns of access to cache memory during cryptographic operations may disclose information about the secret key or intermediate values. An attacker can deduce information about the underlying workings of an algorithm or the data it is processing by looking at cache access patterns.

### To lessen ChaCha20-Poly1305's susceptibility to side-channel attacks, countermeasures such as constant-time implementations, hardware protections, and randomization techniques can be employed. These countermeasures aim to reinforce the algorithm's defence against side-channel attacks by minimizing or eliminating the chance that sensitive data may leak through side channels. Moreover, comprehensive testing and evaluation of implementations in real-world scenarios may help identify and address potential side-channel attack vulnerabilities.

## Known Vulnerabilities or Weakness

While many people consider ChaCha20 and Poly1305 to be secure, many theoretical flaws and vulnerabilities have been discovered in certain circumstances. Here's a summary:

### Chacha20:

#### Nonce Misuse: There might be major security vulnerabilities if a nonce is generated twice using the same key. However, since the purpose of ChaCha20 is to generate a unique keystream for each nonce, it becomes the protocol's duty to reuse nonces [1].

#### Weak Key Attacks: Like every cryptographic method, ChaCha20 is vulnerable to weak key attacks. Its huge key and robust build, however, make it extremely unlikely that you will encounter a weak key.

#### Cryptanalysis: There are no known practical attacks on the full ChaCha20 algorithm, despite the presentation of theoretical ones, such as differential cryptanalysis, against reduced-round variants of the system. However, in reality, these attacks are not seen as a serious danger.

### Poly1305:

#### Key Reuse: If the Poly1305 authenticator is used to transmit several messages with the same key, vulnerabilities can exist. Poly1305 is designed to be used with different keys for each message in order to prevent this scenario.

#### Nonce Misuse: Similar to ChaCha20, Poly1305 is vulnerable to nonce abuse if the same nonce is used with the same key twice. However, appropriate usage of distinct nonces reduces this issue.

#### Cryptanalysis: Despite a comprehensive analysis, no viable cryptanalytic attacks against Poly1305 as a whole have been discovered. However, scientists are actively investigating potential shortcomings and design improvements.

After extensive analysis by cryptographers, both ChaCha20 and Poly1305 are regarded as secure [1][2]. For cryptographic applications, they provide strong security assurances when used properly and in conjunction with pertinent protocols. Adequate implementation and usage, however, are crucial to offset these shortcomings and risks.

# Performance Analysis

## CPA on Firmware AES

To findA side-channel attack known as CPA is non-intrusive and exploits variations in a device's electromagnetic emissions or power consumption as it carries out cryptographic operations. An attacker can deduce information on the secret keys used in the cryptographic method by establishing a correlation between these power consumption traces and known intermediate values, such as intermediate outcomes of AES encryption.

In a CPA attack against Firmware AES, an attacker with access to the target device running the AES firmware can monitor the power consumption of the device while it is encrypting or decrypting data. An attacker compiles a set of power use traces linked to different plaintext inputs. By analyzing the relationship between the observed power usage traces and the anticipated power consumption patterns, the attacker can deduce information about the bytes of the AES key.

## AES Trace Capture

AES trace capture is the process of monitoring a target device's power consumption while it performs AES encryption or decryption. During trace capture, the target device uses plaintext inputs to execute the AES algorithm, and the variations in power consumption caused by the cryptographic processes are recorded as traces [12].

A blue graph with black lines

Description automatically generated

Figure 3. Power Trace of ChaCha20poly1305 Algorithm

A blue graph with white grid

Description automatically generated with medium confidence

Figure 4. Power Trace of ChaCha20poly1305 Algorithm

A blue line graph with black lines

Description automatically generated

Figure 5. Power Trace of ChaCha20poly1305 Algorithm

## Code Optimization

To find the for loop that might be optimized, we examined the code. For this, it was necessary to find a loop that carried out computationally intensive processes or iterated over several items. We have assessed the method used inside the loop and looked at other algorithms that could achieve the same result with fewer computations or fewer repetitions. Looked at using more power-efficient algorithms or data structures to offer the necessary functionality. Following code optimization, the functionality was thoroughly examined and verified to ensure that there were no mistakes or unexpected behaviour. Hardware platforms or power profiling tools were used to assess the power consumption of the revised code in order to confirm the expected decrease in power usage.

## Energy Calculation

When estimating energy consumption using the mean power generated from power traces, it is necessary to estimate the total energy utilized by a device for a certain activity or period of time.

Power traces track the power consumption of a device over time. These traces are used to calculate the mean power usage by averaging the instantaneous power values across the duration of the trace.

The mean power indicates the average rate of energy consumption of the gadget during the recorded activity. The total amount of energy utilized by the device for a certain operation or period of time may be ascertained using the mean power. E = P \* t, where P is the mean power and t is the operation's time, is the formula to calculate energy consumption (E).

### Without Code Optimization: We have the mean power trace value as -0.0408 which will ideally reside between the range -0.5 to 0.5.

We have considered the voltage to be in the range of 0 to 3.3v.

We will convert the mean power in the range of 0 to 3.3v.We use the following formula to convert the value.

A screenshot of a computer code

Description automatically generated

Figure 6. Mean Power Consumption and Standard Deviation

V = (x – xmin) \* (ymax-ymin)/(xmax-xmin) + ymin

V = (-0.0408 – (-0.5)) \* (3.3-0)/(0.5-(-0.5) + 0

V = 0.4592\* 3.3

V = 1.51176

We then calculate power from formula P=V^2/R where we are not considering the resistance in the system.

We have power = 1.51176\*1.51176= 2.2854w

Energy consumption = Power \* Time = 2.2854\*1.34x10^-7

Total Energy consumed is 0.3062 uJ.

### With Code Optimization: In our mean power value has not any significant change. Similarly, we found exactly same AES traces, So there are very low difference which we can not see easily which leads to Therefor, we do not put any traces again.

### We have the mean power trace value as -0.0488 which will ideally reside between the range -0.5 to 0.5.

We have considered the voltage to be in the range of 0 to 3.3v. After converting it into the range of 0 to 3.3 v we have V = 1.51456v.

We will convert the mean power in the range of 0 to 3.3v. We use the following formula to convert the value.

V = (x – xmin) \* (ymax-ymin)/(xmax-xmin) + ymin

V = (-0.0488 – (-0.5)) \* (3.3-0)/(0.5-(-0.5) + 0

V = 1.51456

We have power = V^2/R = 1.51456\*1.51456 = 2.29389w.

Energy consumption = Power \* Time = 2.29389\*1.34x10^-7

Total Energy Consumed is 0.3073 uJ.

## Comparision of the Results

In the below table shows the comparision between results of code without optimization and code with optimization.

Table 1. Comparision of Results

|  |  |  |
| --- | --- | --- |
|  | Without Code Optimization | With Code Optimization |
| Mean Power Trace | -0.0408 V | -0.0488 V |
| Convert into 0 to 3.3V | 1.51176 V | 1.51456 V |
| Power | 2.2854 w | 2.29389 w |
| Total Energy Consumption | 0.3062 uJ | 0.3073 uJ |

We will calculate the percentage decrease in the power,

Percentage decrease = (Old value – New value)/ Old value \* 100.

Percentage decrease = (0.3073-0.3062)/0.3073\*100

Percentage decrease = 0.3579%

We can see that there is no significant changes in the power consumption when the code is optimized for the algorithm.

# Coclusion

The energy calculations carried out during the ChaCha20-Poly1305 implementation in ChipWhisperer provide valuable information on the power consumption characteristics of the cryptographic algorithm [4]. By gathering power traces and tracking energy use, researchers and developers may assess the security and effectiveness of implementations. These calculations help identify potential side-channel attack spots, optimize resource use, and increase the overall energy efficiency of embedded systems. By thoroughly examining and verifying cryptographic systems, energy calculations contribute to their ongoing improvement. For a range of applications, this minimizes power consumption while guaranteeing robust security.

To move with the future work, Even though ChaCha20-Poly1305 is widely accepted as safe, research is currently being conducted to look at potential enhancements, adjustments, and applications for the technique [1][2]. The following are some subjects needing more study in this area:

## Efficiency Gains

Researchers are always searching for ways to optimize ChaCha20-Poly1305 for a variety of systems and application contexts. This entails investigating methods to reduce computational overhead, memory use, and power consumption without sacrificing or improving security.

## Hardware Implementation

Hardware-accelerated ChaCha20-Poly1305 implementations for Internet of Things (IoT) devices, embedded systems, and hardware security modules (HSMs) remain interesting. Hardware acceleration may significantly improve the efficiency and throughput of cryptographic operations, making ChaCha20-Poly1305 a better choice in resource-constrained scenarios.

##### Acknowledgment

We would like to express our sincere gratitude to our professor for their invaluable guidance and support throughout the duration of this project. Their expertise and feedback were instrumental in shaping our work and ensuring its success. We are truly grateful for their mentorship and encouragement.

##### References

1. ChaCha20Poly1305 Algorithm <https://datatracker.ietf.org/doc/html/rfc8439>.
2. Bernstein, D., "ChaCha, a variant of Salsa20", January 2008, <http://cr.yp.to/chacha/chacha-20080128.pdf>.
3. Cryptography-https://www.fortinet.com/resources/cyberglossary/what-iscryptography#:~:text=The%20Importance%20of%20Cryptography&text=Cryptography%20ensures%20confidentiality%20by%20encrypting,cannot%20be%20hacked%20or%20intercepted.
4. J. ChipWhisperer Nano board <https://mouser.ca/ProductDetail/NewAE/NAE-CWNANO?qs=PzGy0jfpSMvY70QksxQLsA%3D%3D>.
5. ChipWhisperer Nano board <https://rtfm.newae.com/Capture/ChipWhisperer-Nano/>
6. ChipWhisperer Nano board <https://mouser.ca/ProductDetail/NewAE/NAE-CWNANO?qs=PzGy0jfpSMvY70QksxQLsA%3D%3D>.
7. Jupyter notebook https://builtin.com/software-engineering-perspectives/jupyter-notebook.
8. Jupyter notebook introduction <https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html>
9. Bernstein, D., "ChaCha, a variant of Salsa20", January 2008, <http://cr.yp.to/chacha/chacha-20080128.pdf>
10. ChaCha20poly1306-https://github.com/aead/chacha20poly1305.git
11. Procter, G., "A Security Analysis of the Composition of ChaCha20 and Poly1305", August 2014, <http://eprint.iacr.org/2014/613.pdf>
12. National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001, <http://csrc.nist.gov/publications/fips/fips197/>[fips-197.pdf](http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf)