PHOTON Lightweight Hash Function
Brandon Horton, Joseph Killian
Final Report
CS 532: Cryptography and Data Security
Dr. Sherif Hashem

Abstract - *PHOTON is a collection of lightweight cryptographic hash functions. The variant we chose to implement was the version with an output size of 80 bits, a bitrate of 20 bits, and an output bitrate of 16 bits. Our implementation was done in Python as opposed to C, which the authors originally implemented the hash functions in. This particular hash function has applications in MAC where there is lower security/complexity requirement. There are two main aspects of this function, the internal permutation function based off of AES, and the Domain Extension Algorithm, based off of the classical sponge framework.*

Introduction:

Lightweight cryptography was made to be introduced into a wide variety of devices, both hardware and software, in the means that it is easy to implement and does not require a mass amount of space to fit on smaller devices. It is also used to reduce key size, cycle rate, throughput rate, and power consumption [2]. Conventional cryptography is unsuited for systems that lightweight cryptography would be appropriate for. Conventional cryptography is good for systems such as: servers, desktops, tablets, and smartphones. While lightweight cryptography is best suited for embedded systems, RFID technology, and sensor networks, or any device with a microcontroller; the most common being 8-bit, 16-bit, and 32-bit microcontrollers [3]. There are a variety of lightweight cryptographic systems, such as block ciphers, hash functions, message authentication codes, and stream ciphers. For this paper, we have chosen to focus on Lightweight Hash Functions. There are a number of these, such as PHOTON, Quark, SPONGENT, and Lesamnta-LW [3]. We have chosen PHOTON to implement in our Lightweight Hash Function. This paper will go through the construction of this function, and how we implemented it in Python, as it is originally written in the C programming language.

Literature Review:

The first paper we used goes over how PHOTON implements it's lightweight hash function. It goes over its many parts such as the Domain Extension Algorithm it uses, and the Internal Permutations that are based on the Advanced Encryption Cipher. It also goes over how the PHOTON hash function was made for small devices such as an RFID tag, and the challenges of fitting a hash function into such a tiny space of memory. It closes off with a security review and a performance review of the function they have created. This is the paper we cited most in our work, and based our hash function implementation around PHOTON. [1]

The second paper that was used gives an overview of hash functions, lightweight hash functions, how they are implemented, and different examples of lightweight hash functions such as PHOTON. It goes through how many are constructed, using either Sponge of Davies-Meyer construction. Then, it goes into each well-known lightweight hash function and explains what each one does and what makes it different from the other functions. Then finally, it lists the many applications of these functions, and compares them against one another in a table of data. [2]

The third paper gives an overview of lightweight cryptographic functions as a whole, their requirements, and the challenges imposed by working on these certain functions. It also gives a list of well-known examples.

PHOTON was included with lightweight hash functions, but it is not limited to just hash functions. It also included examples of stream and block ciphers, and message authentication codes. It goes over their standards and the author of the paper, NIST (National Institute of Standards and Technology), goes over their own Lightweight Cryptography project. [3]

The fourth paper introduces sponge functions and how they work. Sponge functions are not just used for hash functions and are applicable in many cryptographic techniques. This paper is a good resource for understanding how these functions work and how they are applied within PHOTON. [4]

The fifth reference is not a paper, but a website dedicated to the PHOTON family of hash functions. It provides implementations of PHOTON in C, goes over how these implementations work, their components, and gives a good overview of the different PHOTON functions [5].

The sixth reference is another source on how sponge functions work and are constructed. This in tandem with reference 4 provides a good overview of these important functions and how they are implemented within hash functions and other cryptographic systems. [6]

Lightweight Hash Functions:
Hash functions are a known technique in programming and cryptography. This is when a mathematical function converts a numerical input into another compressed value, or a hash value. Simply put, a hash function takes a message of arbitrary input sizes and it produces a message with a fixed size as an output [2]. There are two main parts of any hash functions; these are construction and compression. Construction's purpose is to iterate the compression function. It also needs to follow certain security criteria to be effective. These are collision-resistance: it is difficult to find two different messages m_0 and $m1$ such that $H(m_0) = H(m_1)$; Preimage resistance: given a hash value $H(m)$, it is difficult to find M, and Second Preimage Resistance: given $m0$, it is difficult to find a different input m_1 such that $H(m_0) = H(m_1)$. [2] These requirements are similar to other cryptographic functions. Where hash functions and the same lightweight version of them differ is size. The size of a hash function is usually big, and when working with lightweight functions, there needs to be a cutdown on the amount of memory that is used. This is done using what is called Sponge construction. It's purpose is to reduce the preimage security for the same internal state size [2]. Later in the paper, we will go over how the sponge function fits in with the rest of the hash function.

Applications/Security:
There are three main applications of these lightweight hash functions: Digital Signatures, MAC, and Authenticated Encryption [2]. For digital signatures, the hash function is often utilized in tandem with asymmetric encryption. The sender of a message puts their plaintext message through the hash function then encrypts it with their private key. This results in the digital signature, which is added at the end of the message. The receiver of the message could then use that digital signature to check if the integrity of the message has been compromised by calculating the hash of the message, and checking if it is a match. Message Authentication Codes (MAC) take in both the message and a secret key to generate a tag for that message, which the receiver of a message could use as verification of the integrity of the message being sent to them [2]. In Authenticated Encryption, the plaintext of a message is encrypted separately, and the plaintext is also fed through the hash function with a key, to generate a MAC to be appended to the ciphertext.

The level of security and complexity provided by PHOTON varies by the size of the variant that is implemented. In our case, we implemented the smallest version, which would be most useful where there are lower security requirements. In general PHOTON has a collision resistance of: $\min\{2^{n/2},2^{c/2}\}$, a second preimage resistance of: $\min\{2^{n},2^{c/2}\}$, and a preimage resistance of: $\min\{2^{min\{n,t\}}, \max\{2^{min\{n,t\}-r'}, 2^{c/2}\}\}$[1]. The specific parameters of these resistances are described below in the problem definition section. So, in the variant we implemented the preimage resistance would be 2^{64}, and this variant was designed by the authors of PHOTON to be used in scenarios where this preimage resistance would be adequate, or where 64-bit MAC is enough [1]. In the next section, the variant we implemented is described, followed by a section describing our approach to implementing it in python.

Problem Definition:

Parameters:

The goal was to implement the smallest version of this hash function, that being the version referred to as PHOTON-80/20/16. The parameters related to this particular variant are displayed in Table 1 below:

	T	N	C	R	R'	D	S
PHOTON-80/20/16	100	80	80	20	16	5	4

Table 1: Displays the parameters relating to the PHOTON-80/20/16 variant of the lightweight hash function.

The parameter **T** refers to the size of the internal state matrix, which is made up of **D** rows and columns, with each cell of the matrix containing **S** bits. The bits that make up the internal matrix are the capacity bits **C** and the bitrate bits **R** which satisfy the formula **T** = **C** + **R** [5]. The bitrate refers to the amount of bits that can be absorbed at each phase of the algorithm, and is also the size of the message blocks. The capacity part of the internal state refers to the remaining cells in the internal state matrix. **R'** refers to the number of bits that are output at each phase of the squeezing phase [5]. This variant was chosen due to it being the simplest version to work with, so that it could be easily understood without getting bogged down in the details of the much larger versions of PHOTON with internal state sizes of 256+ bits.

Construction:

The hash function has two main parts, the Internal Permutation and the Domain Extension Algorithm as shown in Figure 1:

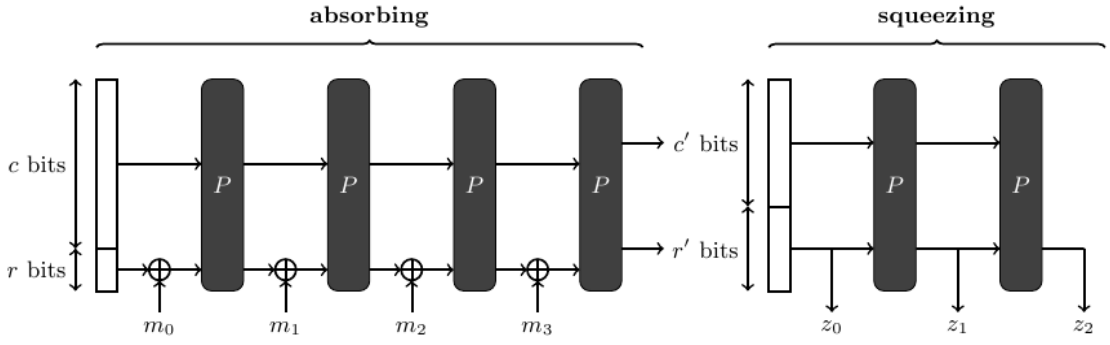

Figure 1: Illustrates the domain extension algorithm based off of the classical sponge framework, and where the permutations occur at each stage of the algorithm. Image taken from [5].

This construction is very similar to the sponge framework as proposed in [4,6], but it differs in that the output bitrate is different from the input bitrate [5]. The absorbing phase applies the message block by block into the internal state matrix until every part of the message has been incorporated. Depending on the length of the message, there could be many absorbing cycles or only a few. Then during the squeezing phase, part of the output hash is extracted from the internal state until the output size has been reached [1,5]. In this particular case, the output size is 80 bits, and the output bitrate is 16 bits, so 5 cycles of squeezing occur to reach the desired output size. In between each phase of either applying the message to the internal state, or extracting output, an internal permutation function is applied to the internal state [5].

Internal Permutation:

The Internal Permutation function consists of four main functions, these being AddConstants, SubCells, ShiftRows, and MixColumnsSerial, and closely mirrors the structure that is present in the AES cipher [5]. It applies these functions to the internal state matrix which in this case is a 5x5 matrix with each cell consisting of 4 bits. As in all variants of PHOTON, this sequence of functions is applied 12 times in a row for each application of the function in the algorithm [1]. The AddConstants function XORs constants to the first column of the internal state based upon the round of the permutation, followed by another XOR of the first column with another set of distinct internal constants which are not round dependent [1]. The constants themselves depend on the type of variant of PHOTON that is being used. Next is the SubCells function which functions in an almost identical fashion to the SubBytes transformation of AES. In this case, a 4-bit Sbox is applied to every single cell of the internal state. Since the bit size in this case is 4 bits, there are only 16 values in the sbox, as opposed to the larger 8-bit Sboxes of different variants which contain 256 values. Following SubCells is the ShiftRows transformation, which is again very similar to AES. This transformation performs a left circular shift of the first row by zero positions, the second row by one position, the third row by two positions, the fourth row by three positions, and the fifth row by four positions. The final function to be applied is MixColumnsSerial. This function takes each column of the internal state and multiplies it by an A matrix. The formula of the new column being => column[i] = A_matrix * column[i]. This multiplication is done GF(2^4) with the irreducible polynomial: $x^4 + x + 1$[1]. Each of these functions are applied in sequence for 12 rounds, which completes the Internal Permutation function.

Domain Extension Algorithm:

The process begins with the input message being padded. This message first has a '1' bit appended to the end of it, and additional zeros (if necessary) in order to make the padded message a multiple of the bitrate, which in this case is 20 bits [1]. The message is then broken up into chunks of 20 bits each to be incorporated into the internal state. These message chunks are "absorbed" one chunk at a time by XORing the first row of the internal state with the message chunk. Following that the internal permutation function is applied and this process repeats until all message chunks have been used up. The message can be arbitrarily long since its function can take in messages of any size, so there is no set number of rounds of absorbing that can occur. After the message has been completely absorbed the second phase of the algorithm begins. First the first 16 bits of the internal state are extracted, these bits make up the first 16 bits of the output of the hash function. Then the internal permutation function is once again applied to the internal state so that more bits can be extracted. Since the output bitrate for this variant of PHOTON is 16 bits, this process occurs 5 times regardless of the input message

size, since the output size is 80 bits. This phase of the algorithm is known as the "squeezing" phase. All of the extracted bits are then appended to each other to produce the output of the hash function

Approach:

There are two main aspects of the hash function that needed to be implemented and combined to form the PHOTON hash function, those being the Internal Permutation function and the Domain Extension Algorithm which are described in the preceding section. We chose to implement the PHOTON-80/20/16 variant, since it would be the easiest to work with, and to avoid getting stuck working with the larger matrices of the other variants. Python was the language of choice as it was the one we are most familiar with. Brandon focused mainly on the functions concerning the domain extension algorithm, and Joseph worked on the Internal Permutation function. The work done by the two of us was then assembled to create the PHOTON lightweight hash function.

Internal Permutation:

For this aspect of the hash function, there were six functions that were created for the purpose of implementing the internal permutation function. The four major functions discussed in detail above, AddConstants, SubCells, ShiftRows, and MixColumnsSerial, a supplementary polynomial multiplication function named polyMult, and the Permutation function itself. For the AddConstants function, the table of round dependent constants, and distinct internal constants was plugged straight in from [1], as it was provided in the appendix of the paper. The sBox needed for the computation in the SubCells was also provided in the appendix of the [1]. ShiftRows did not require any supplementary information, and is a simple left shift in the matrix. MixColumnSerial proved to be the most challenging to implement, as it required multiplication GF(2^4). The irreducible polynomial was provided: $x^4 + x + 1$, and instead of doing the multiplication explicitly, a lookup table was used, which resulted in creating the polyMult Function which possess the lookup table for any multiplication done GF(2^4). Since there were only 256 possibilities this was a faster solution, but would not be viable for the larger versions of PHOTON. The particular Amatrix needed for this variant was provided in the appendix section as well, so no computation was necessary to generate this. All of the matrices and constants obtained, and utilized can be found in the appendix of [1], and also in the reference implementation provided in [5] that is done in C. Finally, these functions were all arranged in the Permutation function, which carries them all out over the course of 12 rounds.

Domain Extension Algorithm:

This section was a lot more straightforward, and faster to implement than the internal permutation, as all it required was the processing of input, and moving it to and from certain functions. No supplementary materials from the paper were required either, which made it much simpler. Overall there were seven functions that we chose to create for this section, those being: padMessage, splitMessage, binToInt, chunkMessage, absorb, squeeze, and convertOutput. The function padMessage appends a '1' bit to the end of the input message always, and appends as many zeros as necessary to get the length of the message up to a multiple of the bitrate which is 20. The function splitMessage simply breaks up the string of 1s and 0s into 4 bit blocks, and the function binToInt converts them to integers ranging from 0 to 15. Then chunkMessage assembles these integers into chunks of 5 (20 bits each). This chunked message can now be used as input to the absorb function, which takes

in the internal state matrix, the next message chunk and XORs the first row of the internal state, and the message chunk. This function then returns the state so the process can be repeated again if needed. The squeeze function extracts the first four bits of the first row of the internal state, and fills in the state with 0s where the extracted integers came from. The squeeze function then returns the state and the list of 4 integers of output from the internal state. Finally, the convertOutput function takes in a list of integers and converts it back to binary to form the output of the hash function.

PHOTON_80_20_16:
This function assembles all of the functions above to create the hash function. The initial state of the internal permutation used was obtained from [1]. The absorbing bits are the first row of the matrix, and the squeezing bits are the first 16 bits of the first row of the matrix as specified in [1]. The message is padded and broken up into chunks of 20 bits, then the absorbing phase occurs. Once all of the message chunks have been absorbed, then the squeezing phase occurs, and this goes for five rounds. The output of the squeezing phase is then converted back to binary to get the result of the hash function.

Results/Conclusions:
In this paper we implemented the PHOTON-80/20/16 variant of the lightweight hash function in Python. The function of the domain extension algorithm, which closely mirrors a classical sponge function, was described along with the construction of the AES-like internal permutation that is also utilized in this algorithm. We also explained some of the possible applications that lightweight hash functions have in general, as well as the applications that are most well suited for this particular variant, given the lower levels of security and complexity that it possesses.

References:
[1] J. Guo, T. Peyrin, and A. Poschmann, “The photon family of lightweight hash functions,” in Annual Cryptology Conference. Springer, 2011, pp. 222–239.

[2] Baraa Tareq Hammad, Norziana Jamil, Mohd Ezanee Rusli and Muhammad Reza Z`aba, A survey of Lightweight Cryptographic Hash Function, International Journal of Scientific & Engineering Research Volume 8, Issue 7, July-2017

[3] McKay, K. , Bassham, L. , Sonmez, M. and Mouha, N. (2017), Report on Lightweight Cryptography, NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.IR.8114 (Accessed March 19, 2021)

[4] Bertoni, G.M. & Daemen, Joan & Peeters, Michael & Assche, Gilles. (2007). Sponge Functions. ECRYPT Hash Workshop 2007.

[5] Guo, J., Peyrin, T., and Poschmann, A., The PHOTON Family of Lightweight Hash Functions. http://sites.google.com/site/photonhashfunction/

[6] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche and Ronny Van Keer, Cryptographic sponge functions. https://keccak.team/sponge_duplex.html