Lightweight version of π-Cipher

Hristina Mihajloska[*], Mohamed El Hadedy[**], Danilo Gligoroski[***] and Kevin Skadron[**]

[*]Faculty of Computer Science and Engineering at Ss. Cyril and Methodius University, Skopje, Macedonia
[**]Department of Computer Science at University of Virginia, USA
[***]Department of Telematics at Norwegian University of Science and Technology (NTNU), Trondheim, Norway

Abstract

In this paper, we describe the lightweight variant of the CAESAR candidate π-Cipher, denoted as π16-Cipher, which provides working with word sizes of 16-bits and security level of 96-bits or 128 bits. It is parallel, incremental, nonce based, tag second-preimage resistant, authenticated encryption cipher with associated data. The basic operations are ARX (Addition, Rotation and XOR) operations. Its security relies on using nonce as pair (PMN, SMN). The design of π-Cipher is highly parallel both for the internal computing engine as well as for the whole crypto primitive. In this paper we provide an optimized low area implementation for 16-bit version of the π-function.

1 Introduction

We are living in a period of a big paradigm shift known as the "Internet of Things". The basic premise in this new paradigm is that the "Internet of Things" will be a network that connects hundreds of billions or even trillions of sensors or smart devices. They will be integrated into the wearable things, integrated in the everyday usage, or will be integrated in our surrounding environment. However, their basic function will be to collect and transmit information. Then, different types of intelligent applications will use that information to offer better quality of life, to offer increased safety, to reduce the energy consumption, the pollution and to offer many other useful actions.

There comes the crucial role of the lightweight cryptography, as a technology that will ensure the confidentiality, integrity and authenticity of the information flowing to and from the "Internet of Things". In order to solve this security problems, many lightweight cryptographic algorithms, which include stream ciphers, block ciphers and hash functions have already been proposed. However, these proposals mostly focus on confidentiality and integrity, but not authenticity of the data, as a fundamental security functionality in the real-world systems.

Only a few lightweight authenticated encryption schemes have been proposed so far, examples are Hummingbird-2 [4] and ALE [3]. Nobody from the ongoing CAESAR competition for AEAD ciphers [2] has proposed a lightweight version of their own schemes.

The π-Cipher has its goal to be also lightweight cipher from the designing time. In its background theory, it uses algebraic structures - quasigroups of order 4 (the smallest one), instead of that from the biggest order and also is designed to be used on 16-bit registers. The hardware-friendly lightweight block ciphers in the literature generally result in very costly implementations in software. As an example, bit permutations are only wiring with zero costs in hardware, while costing several number of cycles in software. The same is valid for substitution layers, they generally require large lookup tables in software. This fact led to the paradox situation that many "lightweight" ciphers are not actually lightweight with respect to software implementations. Instead of this, π-Cipher uses only a set of simple operations, modular addition, bit rotation, and XOR to provide the diffusion and confusion properties, so it results in relatively fast and cheap implementation in software and hardware.

The remainder of the paper is organized as follows. Section 2 gives a specification of the lightweight version of the algorithm. Section 3 introduces some elements of its cryptanalysis. Section 4 provides lightweight implementation in hardware. We conclude in Section 5.

2 The lightweight version of π-Cipher

In this section we describe the lightweight variant of the CAESAR candidate π-Cipher [5]. The recommended variant is with word size of 16-bits and security level of 96 bits, and is denoted as π16-Cipher096, or security level of 128 bits, denoted as π16-Cipher128. Because we are talking about the lightweight version the number of rounds of the cipher here is 2.

2.1 Specification

π-Cipher is parallel, incremental, nonce based, tag second-preimage resistant, authenticated encryption cipher with associated data. It is ARX (Addition, Rotation and XOR) operations based cipher.

The main building element in the operations of encryption/authentication and decryption/verification is a new construction related to the duplex sponge, called triplex component. It uses the permutation function π twice, it injects a counter into the internal state and digests an input string. The triplex component always outputs a tag. Optionally after the first call of the permutation function it can output a string (that can be a ciphertext block or a message block).

The encryption/authentication operation of π16-Cipher can be described in five phases. Here we give a short description of each of the phases, more detailed explanation can be found in the official documentation for the π-Cipher [5].

1. **Padding** The padding rule for π16-Cipher is the classical one, first a "1" is appended to the message M and associated data AD, followed by "0" bits until the end of the block. Than the resulting padded message is split into m blocks each of which 64 bits, $M = M_1||M_2||\ldots||M_m$. The same is done also for the associated data, where the number of splitted blocks is a, $AD = AD_1||AD_2||\ldots||AD_a$.

2. **Initialization** The internal state IS is initialized with the secret key K, and public part of the nonce (public message number) PMN, then it is updated by applying the permutation function π. Because, π-Cipher works in parallel mode, it has an initial value for the state for all of the parallel parts. We call it *Common Internal State (CIS)*. The next part of this phase is initializing the counter ctr. Since it is a 64-bit counter we initialize it from the capacity part of the CIS.

3. **Processing the associated data** The associated data is processed block by block in parallel using e–triplex components. To every block AD_i we associate a unique counter calculated as a sum of the initial counter ctr and the ordinal number of the processed block i and output the block tag. At the end we calculate the tag for AD as a component-wise addition of all the block tags. The final part of this phase is to update the value of CIS. It is done by xoring CIS with the tag T' and applying the π permutation function.

4. **Processing the secret message number (SMN)** This phase is omitted if the length of the secret message number SMN is 0 (it is the empty string). If SMN is not the empty string, it should be a full block (in this case 64-bits). The first step in this phase is a call to the e-triplex component. The second step of this phase is updating the CIS (for free) which becomes the value of the current internal state after the processing of SMN.

5. **Processing the message** The message $M = M_1||\ldots||M_j||\ldots||M_m$ is processed block by block in parallel by e–triplex components. To every block M_j we associate a unique block counter. The input to every e–triplex component is the CIS, block ctr and M_j, and the output is a pair (C_j, t_j). The final tag T is obtained as a $\boxplus_d$ sum of all block tags t_j and the previously obtained tag.

The decryption/verification procedure is defined correspondingly. The only difference is in the last two phases. The decryption of the SMN is performed by the use of a d-triplex component. For the decryption of the rest of the ciphertext we continue to use a d-triplex component. The output is now a decrypted message block and a tag value. At the end, the supplied tag value T is compared to the one computed by the algorithm. Only if the tag is correct, the decrypted message is returned.

2.2 The permutation function π-function

π-Cipher has ARX based permutation function which we denote as π function. It uses similar operations as the operations used in the hash function Edon-R [7] but instead of using 8–tuples here we use 4–tuples. The permutation operates on a $b = 256$ bits state and updates the internal state through a sequence of 2 successive transformations - rounds. The state IS can be represented as a list of four 4-tuples, each of length 16-bits, where $b = 4 \times 4 \times 16$.

The general permutation function π-function consists of three main transformations $\mu, \nu, \sigma : \mathbb{Z}_{2^{16}}^4 \rightarrow \mathbb{Z}_{2^{16}}^4$, where $\mathbb{Z}_{2^{16}}$ is the set of all integers between 0 and $2^{16} - 1$. These transformations do the work of diffusion and nonlinear mixing of the input. In all of these transformations the main operation is the ARX operation $*$.

An algorithmic definition of the $*$ operation over two 4–dimensional vectors $\mathbf{X}$ and $\mathbf{Y}$ for 16-bit word sizes is given in Table 1.

More details about the π-Cipher can be found in the paper [6] and official documentation [5].

3 Security assumptions

The following requirements should be satisfied in order to use π-Cipher securely:

1. The key should be secret and generated uniformly at random;
2. A nonce in the π-Cipher can be only PMN, or (PMN, SMN) pair;

   If the legitimate key holder uses the same nonce to encrypt two different pairs of (plaintext, associated data) (M_1, AD) and (M_2, AD) with the same secret key K then the confidentiality and the integrity of the plaintexts are not preserved in π–Cipher. This can be achieved under the assumption that PMN is always different for any two pairs of messages with the same key.

   Additionally, π-Cipher offers an intermediate level of robustness when a legitimate key holder uses the same secret key K, the same associated data AD, the same public message number PMN but different secret message numbers SMN_1 and SMN_2 for encrypting two different plaintexts M_1 and M_2. In that case confidentiality and integrity of the plaintexts are preserved. However, in that case the confidentiality of SMN_1 and SMN_2 is not preserved.
3. If verification fails, the decrypted message and the wrong tag should not be given as an output;

Cipher-structure (Encrypt than MAC). First we want to point out that it is relatively straightforward to show that the π-Cipher is an Encrypt-then-MAC authenticated cipher. Let us recall the definition for the Encrypt-then-MAC authenticated cipher: We say that the authenticated cipher is Encrypt-then-MAC if a message M is encrypted under a secret key K_1 and then the tag

Table 1: An algorithmic description of the ARX operation $*$ for 16–bit words.

$*$ operation for 16–bit words

Input: $\mathbf{X} = (X_0, X_1, X_2, X_3)$ and $\mathbf{Y} = (Y_0, Y_1, Y_2, Y_3)$
where X_i and Y_i are 16–bit variables.
Output: $\mathbf{Z} = (Z_0, Z_1, Z_2, Z_3)$ where Z_i are 16–bit variables.
Temporary 16–bit variables: $T_0, \ldots, T_{11}$.

μ–transformation for X:

1.
$$\begin{array}{lcl} T_0 & \leftarrow & ROTL^{1}(\texttt{0xF0E8} + X_0 + X_1 + X_2); \\ T_1 & \leftarrow & ROTL^{4}(\texttt{0xE4E2} + X_0 + X_1 + X_3); \\ T_2 & \leftarrow & ROTL^{9}(\texttt{0xE1D8} + X_0 + X_2 + X_3); \\ T_3 & \leftarrow & ROTL^{11}(\texttt{0xD4D2} + X_1 + X_2 + X_3); \end{array}$$

2.
$$\begin{array}{lcl} T_4 & \leftarrow & T_0 \oplus T_1 \oplus T_3; \\ T_5 & \leftarrow & T_0 \oplus T_1 \oplus T_2; \\ T_6 & \leftarrow & T_1 \oplus T_2 \oplus T_3; \\ T_7 & \leftarrow & T_0 \oplus T_2 \oplus T_3; \end{array}$$

ν–transformation for Y:

1.
$$\begin{array}{lcl} T_0 & \leftarrow & ROTL^{2}(\texttt{0xD1CC} + Y_0 + Y_2 + Y_3); \\ T_1 & \leftarrow & ROTL^{5}(\texttt{0xCAC9} + Y_1 + Y_2 + Y_3); \\ T_2 & \leftarrow & ROTL^{7}(\texttt{0xC6C5} + Y_0 + Y_1 + Y_2); \\ T_3 & \leftarrow & ROTL^{13}(\texttt{0xC3B8} + Y_0 + Y_1 + Y_3); \end{array}$$

2.
$$\begin{array}{lcl} T_8 & \leftarrow & T_1 \oplus T_2 \oplus T_3; \\ T_9 & \leftarrow & T_0 \oplus T_2 \oplus T_3; \\ T_{10} & \leftarrow & T_0 \oplus T_1 \oplus T_3; \\ T_{11} & \leftarrow & T_0 \oplus T_1 \oplus T_2; \end{array}$$

σ–transformation for both $\mu(X)$ and $\nu(Y)$:

1.
$$\begin{array}{lcl} Z_3 & \leftarrow & T_4 + T_8; \\ Z_0 & \leftarrow & T_5 + T_9; \\ Z_1 & \leftarrow & T_6 + T_{10}; \\ Z_2 & \leftarrow & T_7 + T_{11}; \end{array}$$

T is calculated with another secret key K_2 as $MAC(K_2, C)$. The pair (C, T) is the output of the authenticated encryption procedure.

If we describe the e-triplex component used in π-Cipher in a mathematical form we have the following. First the message M is encrypted producing the ciphertext C as

$$IS \leftarrow \pi(CIS_{bitrate} \bigoplus counter \;\; |||| \;\; CIS_{capacity}),$$
$$C \leftarrow M \bigoplus IS_{bitrate}.$$

Then, the tag T is calculated as

$$t \leftarrow \pi(C \;\; |||| \;\; IS_{capacity})_{bitrate}.$$

Here, the value of $CIS_{bitrate} \bigoplus counter \;\; |||| \;\; CIS_{capacity}$ has the role of K_1 in the definition of Encrypt-then-MAC, and the value of $C \;\; |||| \;\; IS_{capacity}$ has the role of the pair (K_2, C) in the $MAC(K_2, C)$ part of the definition of Encrypt-then-MAC.

Associated Data and NONCE reuse. If we encrypt two different plaintexts M_1 and M_2 with the same secret key K, associated data AD and nonce $NONCE = (PMN, SMN)$, then neither the confidentiality nor the integrity of the plaintexts are preserved in the π–Cipher. However, as one measure to reduce the risks of a complete reuse of the $NONCE$ we have adopted the strategy of a composite $NONCE = (PMN, SMN)$. If either PMN or SMN are different, then both the confidentiality and integrity of plaintexts are preserved.

Plaintext corruption, associated-data corruption, message-number corruption, ciphertext corruption. We posit that the π-Cipher can straightforwardly be proven INT-CTXT secure under the assumption that the permutation π is an ideal random permutation without any structural distinguishers, by adapting the proof of the XOR-MAC scheme [1]. This is due to the close resemblance of the tag-generation part of the π-Cipher with the XOR-MAC.

Ciphertext prediction. The best distinguishing attack that we know for the π-Cipher is for the versions π16-Cipher096 and π16-Cipher128 with just one round and is described in [5]. The complexity of the attack is 2^{65} computations of the operation $*$, and the space is $2^{65} \times 16 = 2^{69}$ bytes.

Replay and reordering. For the π-Cipher, the standard defense against both replay and reordering is for the sender to use strictly increasing public message numbers PMNs, and for the receiver to refuse any message whose message number is no larger than the largest number of any verified message. This requires both the sender and receiver to keep state.

Sabotage. The π-Cipher puts the encryption of the SMN value as the first block of the ciphertext C. Thus, in protocols that use the π-Cipher, the receiver can make an early reject of invalid messages by decrypting the first block (containing the SMN) and comparing it to its expected value. Only if this check passes the receiver continues with the rest of the decryption and tag computation. Note however, that this requires the protocol to not return error messages to the sender, in order to avoid timing attacks. AES-GCM does not have this property.

Plaintext espionage. Since the attacker's goal here is to figure out the user's secret message, the only feasible attack can happen when the size of the secret message is small by building a table of encrypted secret messages. To defend against this attack the π-Cipher requires the nonce pair $NONCE = (PMN, SMN)$ to have a unique value for every encryption.

Message-number espionage. In the π-Cipher there is a dedicated phase for encrypting the secret message number SMN, and figuring out the value of SMN is equivalent to breaking the whole cipher which is infeasible under the assumptions that the permutation $\pi()$ is random.

4 Hardware evaluation

4.1 ARX Custom Processor

In this paper, we introduce three different custom hardware implementations of π-Cipher. First of all we are going to present how the main operation $*$ from the π-function is built. Here it is called the ARX Custom processor.

4.1.1 Single Width

As shown in Fig. 1, the basic ARX architecture consists of two memories, ten 16-bit adders, two Rotators, and two Xoring banks, distributed in two groups. One part is used to calculate the 4-dimensional vector X, and the other part is used to calculate the 4-dimensional vector Y. Each direction is controlled by a controller, to organize the data flow from the input ports to the output ports.

4.1.2 Memory

The ARX Processor is presented in three different versions based on the data width size. For the 16 bit version, each memory block consists of three 8-byte random-access memories and one 8-byte read-only memory to hold the intermediate values as shown in Fig. 2. Each memory controller is simply a finite state machine controlling two different counters. The first counter is used as a write address to get the input for both directions and located in the memory. Once the data are located, the read counter is used to move the data from the

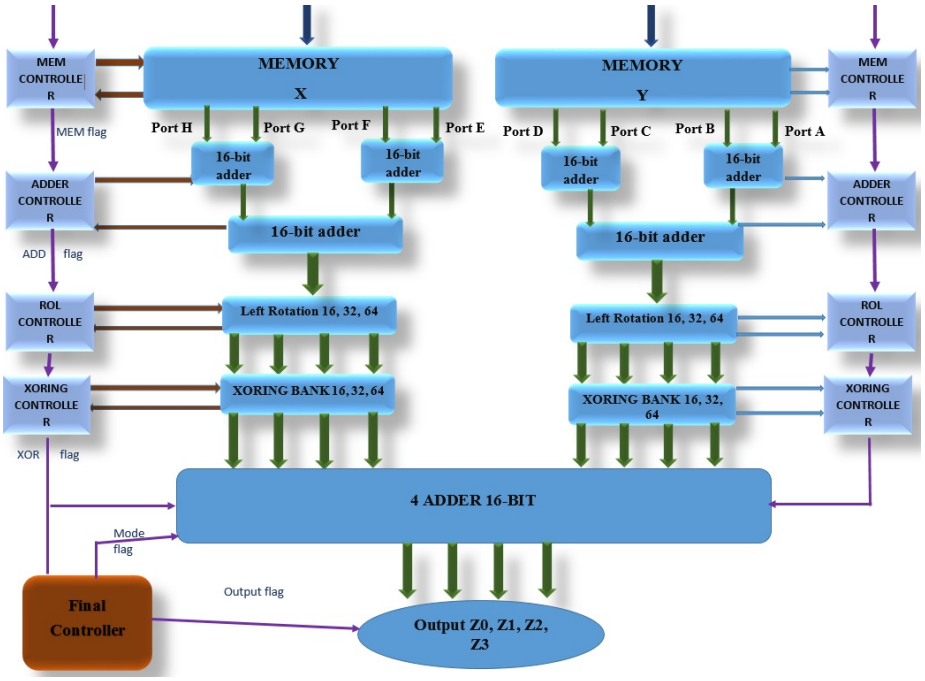


Figure 1: The ARX Processor (Single-Width Core)

memory to the parallel adders to process the intermediate values based on the adder-controller finite state machines.

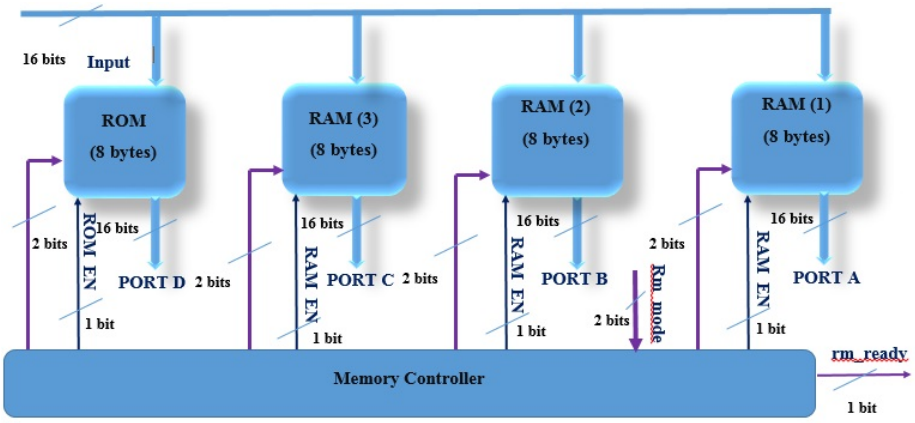


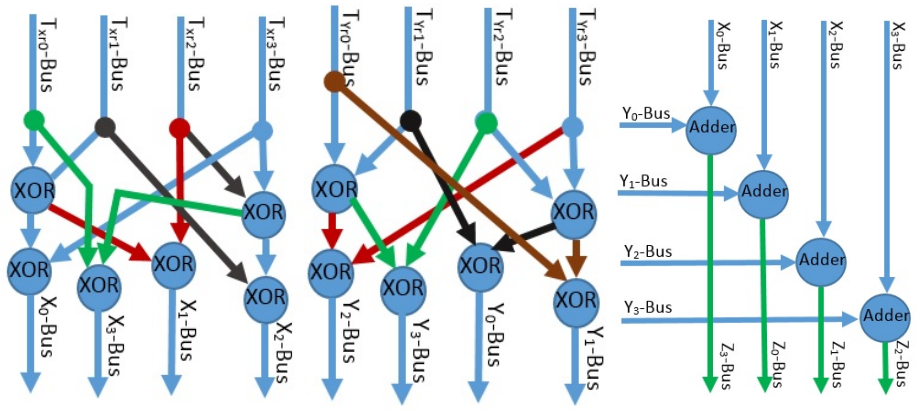
Figure 2: Memory Architecture(16 bit version)

4.1.3 ALU (Single-Width Core)

Once the reading process starts, the adders between the memory and rotators calculates the first intermediate values controlled by the ADDER controller. In the 16-bit version, the controller would move in one cycle per each equation; then rotator starts to calculate the other phase of the equation and hold it in the Xoring bank. The operation will continue until all four equations have been calculated (see Sec. 2) . Then the Xoring bank starts to calculate final values of X and Y in one cycle. The X and Y directions are running in parallel. The output of them are going to be summed by four 16-bit adders to produce the output. The final controller is responsible to receive a control signal from both Xoring blocks in each direction and generate the output flag, which is used to trigger the further blocks.

4.1.4 Rotator (Single Width)

As shown in Fig. 1, the rotator is receiving data from the adders and sending data to the Xoring bank, based on the ROL controller. Simply, the rotator left-rotates the data coming from the adders (see Sec. 2).



(a) X-direction (b) Y-direction (c) Adding Bank

Figure 3: Xoring & Adding Banks for both directions

The Xoring bank recieves the data from the Rotator and buffers them one by one, until the four equations have finished their work in the adder section. Once the four values are ready in the Xoring bank, the Xoring operation starts, controlled by the Xoring controller as shown in Fig. 3, to produce the final result of each direction. Once the data has been processed by the Xoring Bank in each direction, the combination between both outputs Fig. 3c produces the final output of the ARX engine Z_0, Z_1, Z_2, Z_3.

4.1.5 Final Controller (Single Width)

The final controller, once it has received the xor_flag from the Xoring controller, will generate the output flag, which is used in further blocks in the π-Cipher.

4.2 Double-Width Core

Instead of using three adders and one rotator in each direction to compute the equations as in the single core, we use six adders and two rotators in each direction. This decreases execution cycles for a given amount of work. However, this increases memory output ports to 8 instead of 4 ports.

4.3 Quad-Width Core

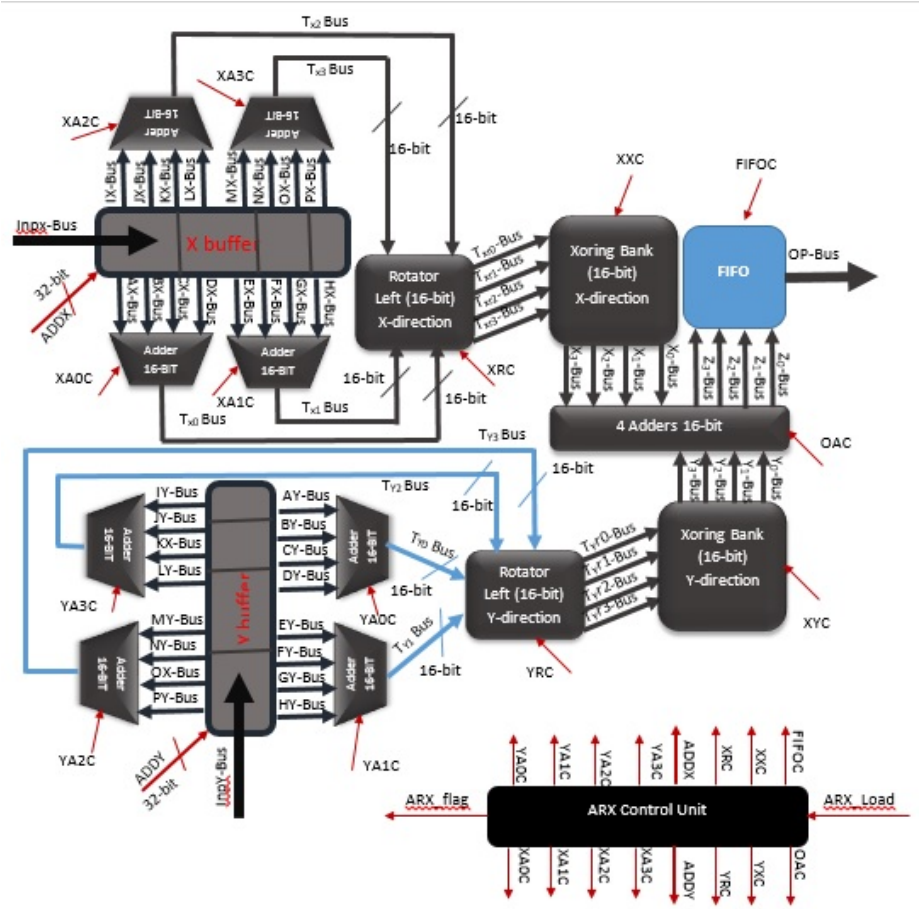


Figure 4: ARX Processor (Quad-Core Width) 16-bit Version

Instead of using six adders and two rotators in each direction to compute the equations as in the double core, we use twelve adders and four rotators in each direction. This increases memory output ports to 16 instead of 8 ports. As shown in Fig. 4, the ARX consists of dual core processors running in parallel, each core processor has 8-byte buffer, it receives the data from 16-bit input port.

The buffer has sixteen reading ports, each port is controlled by 2-bit address bits.The total width of the address port is 32 bits that are coming from the control unit as shown in Fig. 4. Once the data are written on the reading ports, there are four 16-bit ripple carry adders that are going to process these data to produce the summation results controlled by XA0C, XA1C, XA2C, XA3C, YA0C, YA1C, YA2C, and YA3C signals, which come from the control unit. The result of the adders are going to be process by the 16-bit rotator unit through $Tx_0, Tx_1, Tx_2, Tx_3, Ty_0, Ty_1, Ty_2$ and Ty_3 16-bit buses.

The data is going to be written on the $Tx_0, Tx_1, Tx_2, Tx_3, Ty_0, Ty_1, Ty_2$ and Ty_3 16-bit buses once the rotators finish their work, controlled by XRC and YRC control signals, which come from the control unit. The dual core processors results are going to be xor-ed to each other by using Xoring Bank, which are controlled by control signals XXC and XYC. Once the output of the xoring operation data are written on $X_0, X_1, X_2, X_3, Y_0, Y_1, Y_2$, and Y_3 16-bit buses, the four 16-bit ripple carry adders start to add the received data and send it to the FIFO through the Z_0, Z_1, Z_2, Z_3 16-bit buses.

The four adders operation are controlled by the control unit XXC and XYC signals. The 8-byte FIFO is controlled by FIFOC signal and is used to store the data are operated by the adders. Once the Z buses data are stored in the FIFO, the control unit is going to set the arx_flag high. This means the ARX engine processed the data and ready to receive a new data from the input ports.

4.3.1 Buffers

Each processor has eight byte memory, consist of one port for writing and 16 ports for reading. Each port is 16-bit width and each of the reading ports is controlled by 2-bit address port, which comes from the control unit.

4.3.2 Adder

The 16-bit ARX engine relies on using 8, four 16-bit input ports adders to process the data comes out of the buffers as shown in Fig. 4. In reality, each four input port 16-bit adders consists of 3 16-bit ripple carry adders. The first two adders are used to add the buffer results and the last adder is used to sum the both results from the previous adders. The all adders in the engine are controlled by several control bits come from the control unit.

4.3.3 Rotator

The rotator component is responsible on rotating the adders output by different rotation values based on the mathematical model of the π-Cipher[5].

4.3.4 Xoring Bank

As iti is shown in Fig. 3, the xoring Bank does the same functionality as the single-core width in Section 4.1.4.

4.4 FPGA Implementation of the Custom ARX Engines

The area, clock rate, and throughput of the custom ARX processing units are summarized in Tables 2 and 3.

Table 2: The ARX engine

	Single Width		Double Width		Quad Width	
	Frequency(MHz)	Area(Slices)	Frequency(MHz)	Area(Slices)	Frequency(MHz)	Area(Slices)
π16-Cipher096	371	132	324	154	347	266

Table 3: The ARX Performance (π16-Cipher)

	Single Width	Double Width	Quad Width
Throughput	3.57 Gbps	3.68 Gpbs	4.34 Gpbs
Area(Slices)	132	154	266
Throughput/Area (Mbps/slices)	27.69	24.47	16.71

5 Conclusion

In this paper, we describe the lightweight variant of the CAESAR candidate π-Cipher, denoted as π16-Cipher, which provides working with word sizes of 16-bits and security level of 96 bits or 128 bits. We provide an optimized low area implementation for 16-bit version of the π-function. In the future we plan to extend this work on the whole cipher and to compare our results with already published hardware implementations of AEAD ciphers.

Also we are planning to produce the smallest as can be a software implementation for π16-Cipher.

Acknowledgements

This work was supported in part by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] Mihir Bellare, Roch Guérin, and Phillip Rogaway. Xor macs: New methods for message authentication using finite pseudorandom functions. In Don

Coppersmith, editor, *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 1995.

[2] D. J. Bernstein. Caesar: Competition for authenticated encryption: Security, applicability, and robustness. CAESAR web page, 2013. `http://competitions.cr.yp.to/index.html`.

[3] Andrey Bogdanov, Florian Mendel, Francesco Regazzoni, Vincent Rijmen, and Elmar Tischhauser. Ale: Aes-based lightweight authenticated encryption. In Shiho Moriai, editor, *Fast Software Encryption*, volume 8424 of *Lecture Notes in Computer Science*, pages 447–466. Springer Berlin Heidelberg, 2014.

[4] Daniel Engels, Markku-JuhaniO. Saarinen, Peter Schweitzer, and EricM. Smith. The hummingbird-2 lightweight authenticated encryption algorithm. In Ari Juels and Christof Paar, editors, *RFID. Security and Privacy*, volume 7055 of *Lecture Notes in Computer Science*, pages 19–31. Springer Berlin Heidelberg, 2012.

[5] Danilo Gligoroski, Hristina Mihajloska, Simona Samardjiska, Hakon Jacobsen, Mohamed El-Hadedy, and Rune Erlend Jensen. π-cipher v2. Cryptographic competitions: CAESAR, 2014. `http://competitions.cr.yp.to/caesar-submissions.htmls`.

[6] Danilo Gligoroski, Hristina Mihajloska, Simona Samardjiska, Håkon Jacobsen, RuneErlend Jensen, and Mohamed El-Hadedy. pi-cipher: Authenticated encryption for big data. In Karin Bernsmed and Simone Fischer-Hübner, editors, *Secure IT Systems*, volume 8788 of *Lecture Notes in Computer Science*, pages 110–128. Springer, 2014.

[7] Danilo Gligoroski, Rune Steinsmo Ødegård, Marija Mihova, Svein Johan Knapskog, Ljupco Kocarev, Aleš Drápal, and Vlastimil Klima. Cryptographic hash function EDON-$\mathcal{R}'$. In *1st International Workshop on Security and Communication Networks*, pages 85–95, Trondheim, Norway, May 2009. IEEE.