Efficient SIMD based Implementation of Xoodyak

SOHAM BISWAS

M.tech Thesis Report
Kolkata, India March 15, 2024

CERTIFICATE

This is to certify that the thesis entitled

Efficient SIMD based Implementation of Xoodyak

submitted by

Soham Biswas

(Roll Number: CrS2319)

in partial fulfillment of the requirements for the award of the degree of

Master of Technology

in

Cryptology and Security

at

Indian Statistical Institute

is a bona fide work carried out by him under my supervision. The dissertation has fullfilled all the requirements as per regulation of this institute and,in my opinion,has reached the standard needed for submisson .

Subhabrata Samajder

Dr. Subhabrata Samajder
Assistant Professer
TCG-CREST

Date: 13/2/2026.

Sabyasachi Karati

Dr.Sabyasachi Karati
Assistant Professor
Cryptology and Security Research Unit
(C.S.R.U)
R. C. Bose Centre for Cryptology and Security
I.S.I Kolkata

Place: 04/02/2026.

Acknowledgement

I would like to express my sincere gratitude to my supervisor, **Dr.Sabyasachi Karati**,**Dr. Subhabrata Samajder** for his invaluable guidance, encouragement, and constant support throughout the course of this research work. His insightful suggestions and constructive feedback helped me overcome many challenges and improve the quality of this thesis.

My heartfelt thanks go to my colleagues and friends at the [**Laboratory/Department Name**] for providing a friendly and stimulating environment and for their help during various stages of this work.

I would also like to thank all the teachers at **I.S.I** for their valuable suggestions and discussions throughout the course.

Finally, I am deeply indebted to my family for their love, patience, and unwavering support which kept me motivated throughout this journey.

Soham Biswas
11.07.2025

Contents

Chapter 1

Introduction

Cryptography is the science of securing communication and data through the use of codes in an adversarial setting, so that only intended recipients can understand the information. It has ancient roots—from early civilizations like **Egyptians (circa 1900 BCE)**; one of the earliest known uses of encryption was found in a tomb inscription where non-standard hieroglyphs were used, likely more as a puzzle or status symbol than for secrecy, though, and **Spartan Scytale** (circa 500 BCE); a tool used by the Spartans, the scytale was a cipher device involving a strip of parchment wrapped around a rod. The message made sense only when wrapped around a rod of the same diameter. But the first well-recognized cipher was **Caesar Cipher** (circa 50 BCE); Julius Caesar famously used a simple substitution cipher, shifting letters a fixed number of places in the alphabet to encode military messages.

As political and military affairs became more complex, so did cryptography and we arrive at the Renaissance period. In the 9th century, Arab scholar Al-Kindi wrote A Manuscript on Deciphering Cryptographic Messages, which described frequency analysis—an important breakthrough in codebreaking. By the 16th century, **Vigenère Cipher** became quite popular; this cipher used a keyword to determine multiple Caesar-style shifts, making frequency analysis much harder.

In 19th & early 20th century brings a lot of major breakthroughs.The invention of devices like the Jefferson Disk (1795) and later the Enigma Machine in Germany marked a leap in automated encryption.The Enigma machine is one of the most iconic devices in the history of cryptography .It was invented by German engineer Arthur Scherbius at the end of World War I.The machine resembled a typewriter with a series of rotating discs, or rotors, each of which scrambled letters according to a set internal wiring.each time a key was pressed, the rotors rotated, changing the substitution pattern and creating a polyalphabetic cipher—far more difficult to break than simple substitution ciphers. The number of possible configurations was astronomical—estimated at approximately 150 quintillion combinations for the military version—making brute-force decryption seem impossible at the time. The belief in Enigma's unbreakability was shattered through the efforts of mathematicians and codebreakers. Polish cryptanalysis, including Marian Rejewski, were the first to make any real progress against Enigma.Later developing on their works british codebreakers at Bletchley Park, led by Alan Turing, Dilly Knox, and Gordon Welchman nvented the Bombe, an electromechanical machine designed to test Enigma rotor settings rapidly.Successes in breaking Enigma messages were crucial during the Battle of the Atlantic, allowing Allied forces to anticipate and counter U-boat attacks.The battle against Enigma introduced many foundational ideas like statistical analysis and pattern recognition, Development of digital computers,importance of key management and secure

communication protocols.

1.1 From Encryption to AEADs, Modern day Cryptography

The digital age brought unprecedented challenges and advances. Modern cryptographic systems are designed to fulfill four main objectives i.e **Privacy**:Ensuring that the encrypted message is impossible to understand without the knowledge of a secret key,**Integrity**: Guaranteeing that information has not been altered in transit i.e altered cipher texts will not be decrypted and **Authentication**:Verifying the identity of users or systems . Different cryptographic concepts,techniques and primitives emerged keeping different goals in mind.Primitives Like ***Symmetric Key Encryption***(SKE) and ***Public Key Encryption***(PKE) ensures privacy but cannot ensure integrity and authentication in and of themselves. Symmetric key Encryption uses the same key for both encryption and decryption so for two parties to communicate securely a prior key exchange protocol is necessary. AES and ChaCha20 are two of the most commonly used Symmetric Key Encryption algorithms.It is mostly used for file encryption, secure disk storage etc.Public key encryption uses a shared public key for encryption and a secret private key for decryption so only the receiver needs to run the key generation and anyone can send message to the receiver securely without running a key exchange protocol. Two of the most prevalent example of PKE are RSA (Rivest-Shamir-Adleman) used in SSl/TLS handshake,e-mail encryption and Elliptic Curve Cryptography (ECC) widely used in mobile and IoT.

Another very ubiquitous primitive are Hash Functions and Message Authentication Algorithms or ***MAC***s. Hash Functions maps arbitrary bit strings to a fixed length output string . Approved hash functions (such as those specified in FIPS 180 and FIPS 202) are designed to satisfy the following properties **One-way**: It is computationally infeasible to find any input that maps to any new pre-specified output,**Collision-resistant**:It is computationally infeasible to find any two distinct inputs that map to the same output. Hash functions provides integrity; If the hash of a message is sent with the cipher-text the sender can confirm if message has been mauled in the transit by hashing the decrypted cipher text and matching it with the given hash.Hash Functions are widely used for file integrity check,Password hashing, designing Hash based MACs etc. Some of the most widely used Hash Functions are SHA-2, SHA-3[**SP**].

The message authentication code (**MAC**) is generated from an associated message as a method for assuring the integrity of the message and the authenticity of the source of the message.A Message Authentication Code is a tuple of 3 efficient algorithms (G, S, V) i.e a Key-Generation algorithm(G) that generates a key(k) given a security parameter(λ),a Signing Algorithm $S(k, x) \to t$ that generates tag t given a message x and key k and a verification algorithm $V(k, x, S(k, x)) \to \{0, 1\}$ that given the tag(t),message(x) and key(k) accepts or rejects. As the name suggests MACs provides a way for authentication.Now MAC algorithms can be constructed form Hash Functions such MACs are called HMAC.There are MACs based on other primitives as well like OMAC, CCM, GCM, and PMAC are MACs based on Block Ciphers.

Finally a way provide privacy,integrity and authentication altogether is **Authenticated Encryptions**(AE).AEs are usually comprise of a secure encryption and an MAC algorithm.The encryption

provides privacy and the MAC provides integrity and authentication. There are 3 ways to combine an encryption and a MAC to from and AE; MAC and Encrypt: the cipher text and tag is computed separately using two independent secret keys$t = h_{k_1}(x), C = \mathscr{E}_{k_2}(x)$,and $t \mid\mid C$ is transmitted,MAC then Encrypt:first tag is computed $t = h_{k_1}(x)$ then the plain text is encrypted concatenated with the tag $C = \mathscr{E}_{k_2}(t \mid\mid x)$ lastly Encrypt then MAC:first the cipher text in computed $C = \mathscr{E}_{k_2}(x)$ and instead of the plain text the tag is generated form the cipher text $t = h_{k_1}(C)$.Encrypt then MAC is most preferred among the three.Sometimes but not always an associated data is present with the message that is not encrypted but is included during tag generation as it's integrity and authentication is important.An AE carrying authenticated data is called **AEAD** or Authenticated Encryption with Associated Data. Some of the most popular AEs are AES-GCM,CCM (Counter with CBC-MAC) – used in constrained devices,ChaCha20-Poly1305[DSSS17] – a fast AEAD cipher ideal for mobile and low-power environments.

1.2 Lightweight Cryptography

Many modern devices of today especially in the context of the Internet of Things (IoT), mobile computing, and embedded systems have very limited resources in terms of Processing power (CPU),Memory (RAM/ROM),Bandwidth,Battery life etc. Devices like IoT sensors,Smart cards,Medical implants,RFID tags,Wearables often have very limited processing capabilities (e.g., 8-bit microcontrollers with kilobytes of RAM). Standard cryptographic algorithms might Drain power quickly,cause latency Or not run at all.Lightweight cryptography is specifically designed to run efficiently on such hardware.With IoT growing as rapidly as it is billions of devices are coming online every year , many with low-power chips. These devices still need;Encryption (for privacy),Authentication (to verify identity),Integrity (to detect tampering).International organizations like NIST and ISO are defining standards for lightweight cryptography.NIST's Lightweight Cryptography Project aims to standardize algorithms that balance security and performance for constrained environments.

Xoodyak[Dae+20] is one such modern lightweight cryptographic algorithm designed for constrained environments such as Internet of Things (IoT) devices, embedded systems, and resource-limited applications. It provides authenticated encryption, hashing, and pseudo-random number generation in a compact and efficient manner, making it suitable for applications where memory, power, and computational capacity are severely limited.Xoodyak was developed by a team of cryptographers;Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche — who were also among the designers of Keccak (SHA-3).Its name comes from Xoodoo;the cryptographic permutation at its core.Xoodyak was submitted to NIST's Lightweight Cryptography Project, where it was recognized for its strong security design and efficient performance across multiple platforms.Xoodyak is very eddicient on 8-bit, 16-bit, and 32-bit microcontrollers due to Compact code and memory footprint, Single primitive(Xoodoo Permutation) for multiple cryptographic services.

But it's design is also works very well with high degree of parallelization that allows us to run multiple instances of Xoodyak in simultaneously. It is very effective for a powerful server machine in a network where a few server devices manage many small IoT devices in a network. We explore these implementations on latest intel processor with AVX2 and AVX512 instruction set. One paradigm of such implementation is included in eXtended Keccak Code Package by the Keccak team gives that gives upto 16 way parallelization. We try to exporle some other paradigms for SIMD parallelization running upto 512 many instances at a time.

Chapter 2

Xoodyak

Xoodyak was introduced on in March 2019 for the NIST lightweight cryptography competition as a lightweight, versatile cryptographic scheme designed to serve many symmetric-key cryptographic purposes while being efficient on low-end devices and resilient to side-channel attacks. It is based on the full state variant of duplex construction. It is similar to Markku Saarinen's Blinker, Mike Hamburg's Strobe, and Trevor Perrin's Stateful Hash Objects (SHO). Xoodyak operates by maintaining an internal state that accumulates ("absorbs") input data, and then "squeezes" out output data. It is suitable for a wide range of symmetric-key functions, including hashing, pseudo-random bit generation, authentication, encryption, and authenticated encryption.Internally, Xoodyak relies on the Xoodoo permutation, a 384-bit permutation inspired by Keccak-p and dimensioned like Gimli for efficiency on low-end processors. Xoodoo's state is organized as three 128-bit planes, which interact through mixing and nonlinear operations on 3-bit columns, while otherwise behaving like independent rigid layers. This design makes it well-suited for both 32-bit processors and compact hardware implementations.On top of Xoodoo, Xoodyak employs a mode of operation called Cyclist, which is a lightweight and simplified counterpart to Keyak's Motorist mode. Unlike Motorist, Cyclist is not limited to authenticated encryption

2.1 Usage Overview

Depending upon the initialization by *Cyclist* Xoodyak functions on two modes;Hash and Keyed mode.

Hash Mode

Hash Mode is initialized in Xoodyak by Cyclist$(\epsilon, \epsilon, \epsilon)$ .It can absorb input string of arbitrary length and squeeze out digest of a given length.The function Absorb(X) absorbs the input string in the internal state and $h \leftarrow$ Squeeze(n) produces an n-byte output depending on the data absorbed so far. Xoodayk offers 128-bit security against any attack.For $n \geq 32$ bytes(256 bits) it is 128-bit

Algorithm 1 Xoodyak in Hash Mode

1: Cyclist$(\epsilon, \epsilon, \epsilon)$\{Initialize the Cyclist object in hash mode\}
2: Absorb(X)\{Absorb the input string x\}
3: $h \leftarrow$ Squeeze(n)\{Squeeze to produce an n-byte digest\}

collision resistant and for $n \geq 16$ bytes(128 bits) it is second preimage resistant. A sequence of

string can be absorbed and the digest can be squeezed in between like the following example Here

Algorithm 2 Xoodyak in Hash Mode

1: Cyclist$(\epsilon, \epsilon, \epsilon)$
2: Absorb(X_1)
3: Absorb(Y)
4: $h_1 \leftarrow$ Squeeze(n_1)
5: Absorb(Z)
6: $h_1 \leftarrow$ Squeeze(n_1)

h_1 is digest of $y \circ z$ and h_2 is digest of $y \circ z \circ x$.Digest is on the sequence of the sting rather than the concatenation.

Keyed Mode

In keyed mode Xoodyak can function as stram cipher,Message Authentication Code and Authenticated EncryptionYou can compute a tag (MAC) over a message M by: An standard tag length

Algorithm 3 Message Authentication Code

1: Cyclist(K, ϵ, ϵ)\{Initializing in keyed mode with key K\}
2: Absorb(M)\{Absorb message M\}
3: $T \leftarrow$ Squeeze(n)\{Squeeze generates an t-byte tag\}

is $t = 16$ bytes.tag length would be t = 16 bytes (128 bits).Then, encryption is done in a stream cipher-like way, hence it requires a nonce. Block of keystream is generated form the internal state by Squeeze() and use the output as a keystream .This keystream is XORed with the plaintext block to produce a ciphertext block.Encrypt(P) also absorbs plaintext block into the state as it is being encrypted.Unlike a traditional stream cipher since Xoodyak also absorbs the plaintext after the XOR with the key stream, this means the internal state evolves differently depending on the plaintext.So the leakage is limited to one block when the two plaintexts start to differ. Hence, to encrypt plaintext P under a given nonce, we can run the following sequence:

Algorithm 4 Encryption

1: Cyclist(K, ϵ, ϵ)
2: Absorb$(nonce)$
3: $C \leftarrow$ ENCRYPT(P)\{Get a ciphertext C\}

And the decryption done the following way :

Algorithm 5 Decryption

1: Cyclist(K, ϵ, ϵ)
2: Absorb$(nonce)$
3: $C \leftarrow$ ENCRYPT(P)\{Get a ciphertext C\}

And the Authenticated Encryption is constructed by combining the MAC and Encryption. The associated data A is absorbed by the Absorb() method and plain text P is encrypted by ENCRYPT() method.This mode is initialized by Cyclist(K, ϵ, ϵ) . *nonce* and data A are Absorbed in sequence and the tag T is generated by Squeeze() method after the ENCRYPT() so before the Squeeze() produces the tag from the internal state it is updated according to the plain text as well. Xoodyak

Algorithm 6 Authenticated Encryption

1: Cyclist(K, ϵ, ϵ)
2: Absorb(*nonce*)
3: Absorb(A)
4: $C \leftarrow$ ENCRYPT(P){Get a ciphertext C}
5: $T \leftarrow$ Squeeze(n){generates a tag T that depends upon nonce,A,P}

can be used in many ways beyond the basic examples given earlier. In the next sections, we show more advanced examples — such as how to include the nonce more efficiently (or in a way that better resists side-channel attacks), and how Xoodyak can support sessions with evolving subkeys. We also describe a rigorous (though somewhat complex) security guarantee for Xoodyak in keyed mode, and summarize what this means in practice for some specific use cases. In the earlier example of MAC computation, the scheme does not require a nonce. This gives an attacker more flexibility because they can adapt their queries. Still, this authentication method remains secure against attackers with up to about 2^{128} computational power and up to 2^{64} data complexity. In the later example of AE, we assume the application uses nonces properly and does not output decrypted ciphertexts unless they have been verified. Thanks to the nonce, this scheme is even stronger — it can withstand attackers with computational power up to 2^{128} and 2^{160} block of data.

2.2 Specifications

Xoodyak is a Cyclist(. , . , .) layered on top of Xoodoo permutation. First we discuss the Cyclist mode of operation and then the details of Xoodoo permuation.

2.2.1 Cyclist and it's interfaces

The Cyclist mode of operation is built on top of a cryptographic permutation and provides a stateful interface that users can interact with. It is parameterized by the permutation f, the block sizes R_{hash}, R_{kin}, and R_{kout}, as well as the ratchet size ℓ_{ratchet}, all expressed in bytes. The parameters R_{hash}, R_{kin}, and R_{kout} define the block sizes for the hash function and for input and output in the keyed mode, respectively. Since Cyclist reserves up to 2 bytes for frame bits (used for padding and domain separation), it requires that $\max(R_{\text{hash}}, R_{\text{kin}}, R_{\text{kout}}) + 2 \leq b'$, where $b' = b/8$ is the width of the permutation in bytes.

The usual parametrs are Cyclist$(K, \text{id}, \text{counter})$, the Cyclist object starts in **hash mode** if $K = \epsilon$ and if $K \neq \epsilon$ it starts in **keyed mode** . In keyed mode, the object uses the secret key K along with an optional identifier id, and absorbs the counter progressively if counter $\neq \epsilon$. In hash mode, the initialization parameters are ignored.

The functions available differs depending upon the mode in which the object is initialized: the functions Absorb() and Squeeze() can be called in both hash and keyed modes, while the functions Encrypt(), Decrypt(), SqueezeKey(), and Ratchet() are only available in keyed mode. Below we describe the purpose of each function in detail.

- Absorb(X) absorbs an input string X.
- $C \leftarrow$ Encrypt(P) encrypts P into C and absorbs P.
- $P \leftarrow$ Decrypt(C) decrypts C into P and absorbs P.
- $Y \leftarrow$ Squeeze(ℓ) produces an ℓ-byte output that depends on the data absorbed so far.
- $Y \leftarrow$ SqueezeKey(ℓ) behaves like Squeeze(ℓ) but in a different domain, intended for generating a derived key.
- Ratchet() transforms the state irreversibly to ensure forward secrecy.

Next we break down the inner workings and constructions of these methods. The most fundamental two methods defined by the Cyclist are UP() and DOWN().The Down() method processes a single block of input by XORing it with the current state.More specifically it XORs $(X_i \,||\, '01' \,||\, '00'^* \,||\, c_D)$ i.e it XORs X_i along with 1 in next byte after X_i and a color c_D in the last bytes of the state. This color(c_D) serves to distinguish different domains: it differentiates the first block of Absorb() (using '01' in hash mode or '03' in keyed mode), the first block of AbsorbKey() (using '02'), and all subsequent blocks.The Up() method is designed to generate a single block of output. It first incorporates a color c_U into the state, then applies the underlying permutation f, and finally returns the initial bytes of the updated state. The color c_U ensures domain separation by distinguishing between different operations: a block of keystream ('80'), the first block of Squeeze() ('40'), the first block of SqueezeKey() ('20'), a ratchet ('10'), and other possible uses.There is another attribute phase that keeps track of the last internal call, if was UP() or DOWN().

Next using the UP() and DOWN() method we construct AbsorbAny(), SqueezeAny() and Crypt() . The AbsorbAny() method is responsible for absorbing input data. It splits the input string into blocks of a specified length (determined by the r parameter) and invokes Down() and Up() as needed. Similarly, the SqueezeAny() method generates output of a specified length (given by the ℓ parameter). The Crypt() method alternates between calling Up() to generate keystream and Down() to absorb the plaintext. The attribute R_{absorb} specifies the block size used during absorption — it equals R_{kin} in keyed mode and R_{hash} in hash mode. Likewise, the attribute $R_{squeeze}$ defines the block size used during generating the ourput.It is equal to R_{kout} in keyed mode or R_{hash} in hash mode.

Algorithm 7 UP()

Internal interface: $Y_i \leftarrow \textsc{Up}(|Y_i|, c_U)$
$(\text{phase}, s) \leftarrow (\text{up}, f(s \text{ if mode} = \text{hash else } s \oplus (\text{'00'}^* \parallel c_U)))$
return $s[0] \parallel s[1] \parallel \cdots \parallel s[|Y_i| - 1]$

Algorithm 8 DOWN()

Internal interface: $Y_i \leftarrow \textsc{Down}(X_i, c_U)$
$(\text{phase}, s) \leftarrow (\text{down}, s \ \oplus \ (X_i \parallel' 01' \parallel' 00'^* \parallel (c_D \ \& \ '01' \text{ if mode} = \text{hash else } c_D)))$

Algorithm 9 $[X_i] \leftarrow \text{SPLIT}(X, n)$

If X is empty then return array with a single empty string $[\epsilon]$
return array $[X_i]$,with X=$\|_i X_i$ and $|X_i| = n$

The SPLIT() method takes an input string of arbitrary length and divides it into equal length blocks.So next we discuss about the non-fundamental methods that we construct using these three.First up AbsorbAny(X,r,c_D),so it takes 3 parameters;X is the input string that is being absorbed, r is the length of the blocks that X will be devided into for absorbing and finally c_D is the color.First X is split into r length blocks by SPLIT() method next each X_i is absorbed by DOWN() method.

Algorithm 10 AbsorbAny(X,r,c_D)

Internal interface: AbsorbAny(X,r,c_D),
  for all X_i in SPLIT(X,r)
  if phase $\neq$ up then UP(0,'00')
  DOWN(X_i,c_D if first block '00')

Next up is AbsorbKey(K,id,counter), this method is used during the initiation phase of any keyed mode . It sets the *mode* parameter as *keyed* $R_{squeeze}, R_{absorb}$ as predefined value R_{kin}, R_{kout} respectively, absorbs the key and id into the internal state using AbsorbAny() method and finally if the counter is non empty the absorbs it as well.

Algorithm 11 AbsorbAny(K,id,counter)

Internal interface: AbsorbKey(K,id,counter)
  (MODE, $R_{absorb}, R_{squeeze}$) $\leftarrow$ ($keyed, R_{kin}, R_{kout}$)
  AbsorbAny($K \parallel$ id $\parallel$ enc$_8$(|id|), R_{absorb},$'02'(key)$)
  If counter not empty then AbsorbAny(counter,1,'00')

Now we come to SqueezeAny() method.This method is responsible for generating outputs.It applies UP() method and extracts blocks of output until desired ammount is reached.

Algorithm 12 Y $\leftarrow$ SqueezeAny(l, c_D)

Internal interface: Y $\leftarrow$ SqueezeAny(l, c_D)
  $Y \leftarrow$ UP($min(l, R_{squeeze}), c_D$)
  while $|Y| < l$ do
    DOWN(ϵ,'00')
    $Y \leftarrow Y \parallel$ UP($min(l - |Y|, R_{squeeze})$,$'00'$)
  return Y

Lastly is the CRYPT method, the input text is parsed to the method along with DECRYPT = true or false .First the input text is split into blocks of size R_{kout}.Next UP() method is applied to generate mask of size $|I_i|$ from the internal state.Then pain text is absorbed into the state block by block in both mode ENCRYPT or DECRYPT.Since in DECRYPT mode the output blocks being the decrypted plain text gets absorbed in the internal state by DOWN() method and in ENCRYPT mode the input blocks are the plain text blocks.

Algorithm 13 $O \leftarrow \text{CRYPT}(I, \text{DECRYPT})$

Internal interface:$O \leftarrow \text{CRYPT}(I, \text{DECRYPT})$
  for all I_i in SPLIT(I,R_{kout})
  $O_i \leftarrow I_i \oplus \text{UP}(|I_i|, '80'(\text{crypt})$ if first block else '00')
    $P_i \leftarrow O_i$ if DECRYPT else I_i
    DOWN(P_i,'00')
  return $\|_i O_i$

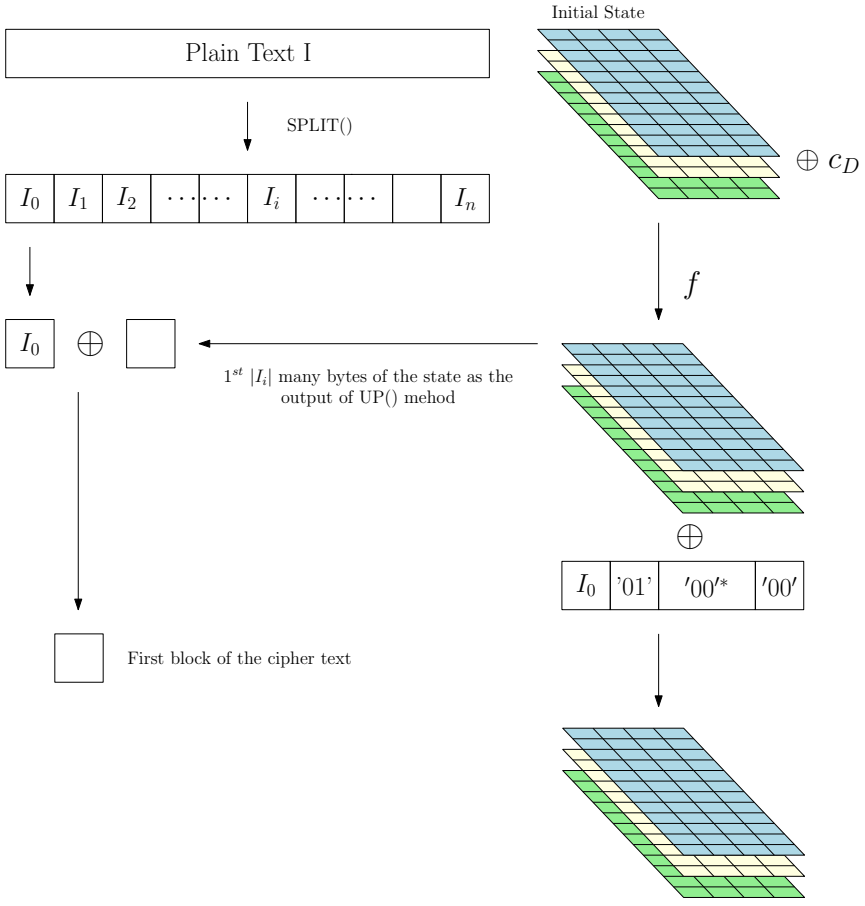

Figure 2.1: A graphical representation of ENCRYPT

Cyclist

Finally we discuss the initiation by Cyclist and all the functions available at the highest level. The CYCLIST($f, R_{hash}, R_{kin}, R_{kout}, l_{ratchet}$) parametrized by the permutation f and block sizes for

hashing, input and output size in keyed mode and output size of RATCHET() method;sets the pahse as up and the initial state as all 0. Next it defines the MODE parameter and input block size(R_{absorb}) for Absorb() method and output block size($R_{squeeze}$) for Squeeze() method.In hash mode both are R_{hash} and in keyed mode it is R_{kin} and R_{kout} respectively. Next if in keyed mode and K is non-empty then K,id and counter is absorbed in the state by AbsorbKey() method.

After initialization by the cyclist depending the mode different function becomes available as we discussed before. Encrypt,Decrypt and SqueezeKey in exclusive to keyed mode. Encrypt() and Decrypt() calls on Crypt method with DECRYPTION tag set to false or true respectively. Absorb(X) just absorbs the string X with color set to '03' for AbsorbAny() method.RATCHET() overwrites part of the internal state with zeros, making it impossible to recover the state as it was before the Ratchet() call.At any point during keyed mode, the user may invoke the Ratchet() function, it reduce the risk if the internal state is exposed, for example, through a side-channel attack.In an authenticated encryption scheme, Ratchet() is usually called either just before generating the authentication tag or immediately after. Calling it right before the tag is more efficient because it requires only one additional application of the permutation f. On the other hand, calling it right after the tag allows the ratcheting process to run asynchronously while the cipher text is being sent and before the next message is processed.If Ratchet() is not the final operation, the number of times it is called needs to be tracked.

Algorithm 14 Defination of CYCLIST($f, R_{hash}, R_{kin}, R_{kout}, l_{ratchet}$)

Instantiation: cyclist $\leftarrow$ CYCLIST($f, R_{hash}, R_{kin}, R_{kout}, l_{ratchet}$)
Phase and state: (phase,s)$\leftarrow$ (up,'00')
Mode and absorb rate:(mode, $R_{absorb}, R_{squeeze}$) $\leftarrow$ (hash, R_{hash}, R_{hash})
if K not empty then AbsorbKey(K, id, counter)

Interface: Absorb(X)
AbsorbAny(X, R_{absorb}, '03'($absorb$))

Interface:$C \leftarrow$ Encrypt(P), with mode = $keyed$
return Crypt(P, false)

Interface:$C \leftarrow$ Decrypt(C), with mode = $keyed$
return Crypt(P, true)

Interface:$Y \leftarrow$ Squeeze(ℓ)
return SqueezeAny(ℓ, '40'(squeeze))

Interface:$Y \leftarrow$ SqueezeKey(ℓ)
return SqueezeAny(ℓ, '20'(key))

Interface:$Y \leftarrow$ RATCHET(), with mode = $keyed$
AbsorbAny(SqueezeAny($\ell_{ratchet}$, '10'(ratchet)), R_{absorb}, '00')

In an authenticated encryption scheme, Ratchet() is usually called either just before generating the authentication tag or immediately after. Calling it right before the tag is more efficient because it requires only one additional application of the permutation f. On the other hand, calling it right after the tag allows the ratcheting process to run asynchronously while the cipher text is being

sent and before the next message is processed.If Ratchet() is not the final operation, the number of times it is called needs to be tracked.An example of how RATCHET() is implemented within an instances of Authenticated Encryption is following.:

Algorithm 15 Authenticated Encryption with Ratchet

1: Cyclist(K, nonce, ϵ)
2: Absorb(A)
3: $C \leftarrow$Encrypt(P)
4: Ratchet() We can call it here or ...
5: $T \leftarrow$ Squeeze(t)
6: Ratchet() ...here

2.3 Xoodoo Permutation

Xoodoo is a family of permutation parametrized by the number of rounds that applies the same round function iteratively to the state. As we discussed before 384-bit sate is comprised of 3 planes 128 bit each.Each plane consists of 4 lanes each 32 bit.Bits of a state are indexed by (x, y, z) where y indicates the plane x indicates the lane and z indexes the bits with in a lane.

The round function has is made of 5 steps. Mixing layers θ,a plane shifting ρ_{west},addition of round constant ι,a non linear layer χ and plane shifting ρ_{east}.

P_y	Plane y of state P
$P_y << (t, v)$	Shifts (x, z) to $((x + t) mod\ 4, (z + v) mod\ 32)$ of plane y
$\overline{P_y}$	Bitwise complement of A_y
$p_y \oplus P_{y'}$	Bit wise XOR of plane y and y'.
$P_y.P_{y'}$	Bit wise AND of plane y and y'.

The internal steps are defined as below:

- θ :

$$\begin{aligned} \Gamma &\leftarrow P_0 + P_1 + P_2 \\ E &\leftarrow \Gamma << (1, 5) + \Gamma << (1, 14) \\ P_y &\leftarrow P_y + E\ for\ all\ y \in \{1, 2, 3\} \end{aligned} \tag{2.1}$$

- ρ_{west} :

$$\begin{aligned} P_1 &\leftarrow P_1 <<< (1, 0) \\ P_2 &\leftarrow P_2 <<< (0, 11) \end{aligned} \tag{2.2}$$

- ι:XORs a round-dependent constant into the state.

$$P_0 \leftarrow P_0 \oplus \_rc \tag{2.3}$$

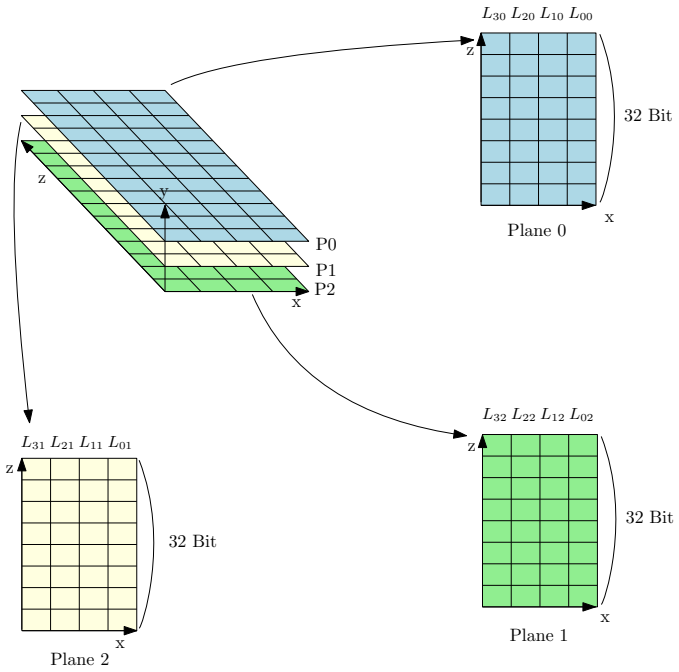

Figure 2.2: Graphical Representation of a Single 384 bit state

- χ :

$$\begin{aligned} P_0 &\leftarrow P_0 + (\overline{P_1}.P_2) \\ P_1 &\leftarrow P_1 + (\overline{P_2}.P_0) \\ P_2 &\leftarrow P_2 + (\overline{P_0}.P_1) \end{aligned} \tag{2.4}$$

- ρ_{east}:

$$\begin{aligned} P_1 &\leftarrow P_1 <<< (0,1) \\ P_2 &\leftarrow P_2 <<< (2,8) \end{aligned} \tag{2.5}$$

The round constants c_i are given below with round number $11 \leq i \leq 0$, in hexadecimal notation.

i	c_i	i	c_i	i	c_i	i	c_i
-11	0x00000058	-8	0x000000D0	-5	0x00000060	-2	0x000000F0
-10	0x00000038	-7	0x00000120	-4	0x0000002C	-1	0x000001A0
-9	0x000003C0	-6	0x00000014	-3	0x00000380	0	0x00000012

The round constant are added to plane $(0, 0, z)$ i.e lane 0 of plane 0.

2.4 Security Claims

An instance of Xoodyak is initialized by Cyclist$[f, R_{hash}, R_{kin}, R_{kout}, \ell_{ratchet}]$ with

- f = Xoodoo[12](all 12 rounds of Xoodoo pemutation)
- R_{hash} = 16 bytes

- $R_{kin} = 44$ bytes
- $R_{kout} = 24$ bytes
- $\ell_{ratchet} = 16$ bytes

With these initiation parameters the security claims by the authors[Dae+20] are.

Claim 1: The success probability of any attack on Xoodyak in hash mode shall not be higher than the sum of that for a random oracle and $\frac{N^2}{2^{255}}$,in calls to Xoodoo[12] or its inverse. where N is the attack complexity in calls to Xoodoo[12] or its inverse.

Claim 2: Let $\mathbf{K} = (K_0, \ldots, K_{u-1})$ be an array of u secret keys, each uniformly and independently chosen from $\mathbb{Z}_2^\kappa$ with $\kappa \leq 256$ and $\kappa \leq 256$ and a multiple of 8. Then, the advantage of distinguishing the array of XOODYAK objects after initialization with CYCLIST$(K_i, \cdot, \cdot)$, with $i \in \mathbb{Z}_u$, from an array of random oracles $\mathscr{RO}(i, h)$, where $h \in (\mathbb{Z}_2^* \cup S)^*$ is a process history, is at most:

$$\frac{q_{\text{iv}} N + \binom{u}{2}}{2^{-\kappa}} + \frac{N}{2^{184}} + \frac{(L+\Omega)N + \binom{L+\Omega+1}{2}}{2^{192}} + \frac{M^2}{2^{382}} + \frac{Mq}{2^{\min(192+\kappa, 384)}}.$$

Here we follow the notation of the generic security bound of the full-state keyed duplex, namely:

- N is the computational complexity expressed in the (computationally equivalent) number of executions of XOODOO[12].
- M is the online or data complexity expressed in the total number of input and output blocks processed by XOODYAK.
- $q \leq M$ is the total number of initializations in keyed mode.
- $\Omega \leq M$ is the number of blocks, in keyed mode, that overwrite the outer state (i.e., the first R_{bout} bytes of the state) and for which the adversary gets a subsequent output block.
  - In particular, this counts the number of blocks processed by DECRYPT$(\cdot)$ for which the adversary can also get the corresponding key stream value or other subsequent output (e.g., in the case of the release of unverified decrypted ciphertext in authenticated encryption).
  - It also counts the number of calls to RATCHET() followed by SQUEEZE(ℓ) or SQUEEZEKEY(ℓ) with $\ell \neq 0$.
- $L \leq M$ is the number of blocks, in keyed mode, for which the adversary knows the value of the outer state from a previous query and can choose the input block value (e.g., in the case of authentication without a nonce, or of authenticated encryption with nonce repetition).
  - This includes the number of times a call to ABSORB() follows a call to SQUEEZE(ℓ) or to SQUEEZEKEY(ℓ) with $\ell \neq 0$.
- $q_{\text{iv}} \leq u$ is the maximum number of keys that are used with the same id, i.e.,

$$q_{\text{iv}} = \max_{\text{id}} \Big| \{ i \mid \text{CYCLIST}(K_i, \text{id}, \cdot) \text{ is called at least once} \} \Big|.$$

Assuming $\kappa \geq 128$ claims 1 and 2 guarantee that XOODYAK achieves 128-bit security in both its hash and keyed modes. The data complexity depends on the parameters q, Ω, and L. As long as these parameters remain within their expected bounds(M), XOODYAK can handle data complexities up to 2^{64} blocks, because the bound remains negligible provided that $N \ll 2^{128}$ and $M \ll 2^{64}$.

In particular, when $L + \Omega = 0$, the construction tolerates even greater data complexities, as the security bound remains negligible as long as $M \ll 2^{160}$.

The parameter q_{iv} accounts for potential security degradation in scenarios involving multi-target attacks: if an exhaustive key search is performed, the security could decrease by $\log_2 q_{\text{iv}} \leq \log_2 u$ bits. However, if the protocol ensures that $q_{\text{iv}} = 1$, then no degradation occurs, and the security level stays at $\min(128, \kappa)$ bits even in the presence of multi-target attacks.

Chapter 3

Background on existing Implementation

We are exploring the SIMD implementation using SSE, AVX2 and AVX512 instruction sets on 128-bit, 256-bit and 512-bit registers. The existing implementations by the Keccak team are included here in eXtended Keccak Code Package [Dae+20]. They have explored two techniques Lane Slicing and Plane Slicing. The existing implementations are :

1. Plane Slicing: On 128-bit register
2. Lane Slicing: On 128-bit, 256-bit, and 512-bit registers

We are exploring the rest, i.e, Plane Slicing implementation on 256-bit and 512-bit registers. We discuss it further in 4Chapter4 . And one other whole different paradigm that we name Bit Slicing, which is discussed in detail in 5Chapter 5.

Now let us discuss the Lane Slicing first. In this implementation, they pack the same lane from multiple instances of the ciphers running in parallel in a single register.Suppose the state bits for i^{th} cipher are $(b^i_{383}, \ldots, b^i_1, b^i_0)$. Each 32 bit are packed with in a lane $(b^i_{32*j+32}, \ldots, b^i_{32*j+1}, b^i_{32*j+0})$. Now lanes from all the parallel instances are packed within a register so one register contains $(b^n_{32*j+31}, \ldots, b^n_{32*j+1}, b^n_{32*j+0} \parallel \ldots \parallel b^0_{32*j+31}, \ldots, b^0_{32*j+1}, b^0_{32*j+0})$ and there are 12 such lanes. Suppose we denote j^{th} lane from i^{th} instance k^{th} plane as L_{ijk} then

$$L_{ijk} = (b^i_{32*(j+4*k)+31}, \ldots, b^i_{32*(j+4*k)+1}, b^i_{32*(j+4*k)+0})$$

. A graphical representation of lanes packed in a register is depicted in Figure 2.1

$$\left|\begin{array}{cccc|c} b^i_{31} & \cdots & b^i_1 & b^i_0 & L_{i00} \\ b^i_{63} & \cdots & b^i_{33} & b^i_{32} & L_{i10} \\ b^i_{95} & \cdots & b^i_{65} & b^i_{64} & L_{i20} \\ b^i_{127} & \cdots & b^i_{97} & b^i_{96} & L_{i30} \end{array}\right| \left|\begin{array}{cccc|c} b^i_{159} & \cdots & b^i_{129} & b^i_{128} & L_{i01} \\ b^i_{191} & \cdots & b^i_{161} & b^i_{160} & L_{i11} \\ b^i_{223} & \cdots & b^i_{193} & b^i_{192} & L_{i21} \\ b^i_{255} & \cdots & b^i_{225} & b^i_{224} & L_{i31} \end{array}\right| \left|\begin{array}{cccc|c} b^i_{287} & \cdots & b^i_{257} & b^i_{256} & L_{i02} \\ b^i_{319} & \cdots & b^i_{161} & b^i_{288} & L_{i12} \\ b^i_{351} & \cdots & b^i_{65} & b^i_{320} & L_{i22} \\ b^i_{383} & \cdots & b^i_{97} & b^i_{352} & L_{i32} \end{array}\right|$$

Now consider $\vec{L}_{jk} = (L_{0jk}, L_{1jk}, \ldots, L_{njk})$ a vector representation of the lanes form different ciphers. So depending on the available registers n=1 for 32 bit registers, n=4 for 128 bit registers, n=8 for 256 bit registers and n=16 for 512 bit registers .So in lane slicing instead of a single lane a lane vector is taken and all the operations are implemented accordingly.Instructions are used such that a single instruction is executed on multiple data i.e the all n lanes in parallel.

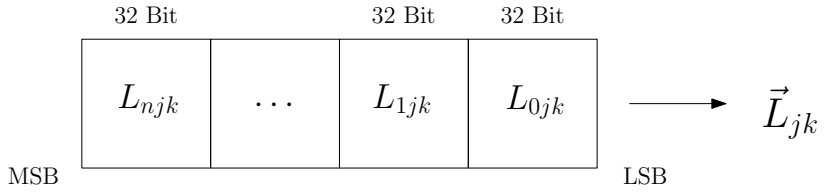

Figure 3.1: 4 Lanes from different ciphers packed in a register

Say the ordered tuple $P_k = \{\vec{L}_{0k}, \vec{L}_{1k}, \vec{L}_{2k}, \vec{L}_{3k}\}$ represents plane k of the state. Plane addition is defined as element wise addition where elements are lane vectors.The lane vector addition is just vector addition where addition operation is just bit wise XOR.The formal definitions are given in Table 1

$\vec{L}_{jk} + \vec{L}_{j'k'}$	$(L_{0jk} \oplus L_{0j'k'}, L_{1jk} \oplus L_{1j'k'}, \ldots, L_{3jk} \oplus L_{3j'k'})$
$P_k + P_{k'}$	$\{\vec{L}_{0k} + \vec{L}_{0k'}, \vec{L}_{1k} + \vec{L}_{1k'}, \vec{L}_{2k} + \vec{L}_{2k'}, \vec{L}_{3k} + \vec{L}_{3k'}\}$
$\vec{L}_{jk}.\vec{L}_{j'k'}$	$(L_{0jk}.L_{0j'k'}, L_{1jk}.L_{1j'k'}, \ldots, L_{3jk}.L_{3j'k'})$
$P_k.P_{k'}$	$\{\vec{L}_{0k}.\vec{L}_{0k'}, \vec{L}_{1k}.\vec{L}_{1k'}, \vec{L}_{2k}.\vec{L}_{2k'}, \vec{L}_{3k}.\vec{L}_{3k'}\}$
$P_k << (t, 0)$	$\{\vec{L}_{(0-t) \bmod 4\, k}, \vec{L}_{(1-t) \bmod 4\, k}, \vec{L}_{(2-t) \bmod 4\, k}, \vec{L}_{(3-t) \bmod 4\, k}\}$
$P_k << (0, v)$	$\{\vec{L}_{0k} <<< v, \vec{L}_{1k} <<< v, \vec{L}_{2k} <<< v, \vec{L}_{3k} <<< v\}$
$\vec{L}_{jk} <<< v$	$(L_{0jk} <<< v, L_{1jk} <<< v, \ldots, L_{3jk} <<< v)$
$\vec{L}^c_{ij}$	$(\overline{L}_{0jk}, \overline{L}_{1jk}, \ldots, \overline{L}_{3jk})$

Table 3.1: Operation Definition for Vector Representation

So the modified steps are following

- θ :

$$
\begin{aligned}
\Gamma &\leftarrow P_0 + P_1 + P_2 \\
\vec{\Gamma}_0 &\leftarrow \vec{L}_{00} + \vec{L}_{01} + \vec{L}_{02} \\
\vec{\Gamma}_1 &\leftarrow \vec{L}_{10} + \vec{L}_{11} + \vec{L}_{12} \\
\vec{\Gamma}_2 &\leftarrow \vec{L}_{20} + \vec{L}_{21} + \vec{L}_{22} \\
\vec{\Gamma}_3 &\leftarrow \vec{L}_{30} + \vec{L}_{31} + \vec{L}_{32} \\
\Gamma &= \{\vec{\Gamma}_0, \vec{\Gamma}_1, \vec{\Gamma}_2, \vec{\Gamma}_3\} \\
E &\leftarrow \Gamma << (1, 5) + \Gamma << (1, 14) \\
\Gamma &\leftarrow \Gamma << (1, 0) = \{\vec{\Gamma}_3, \vec{\Gamma}_0, \vec{\Gamma}_1, \vec{\Gamma}_2\} \\
E &\leftarrow \Gamma << (0, 5) + \Gamma << (0, 14)
\end{aligned}
\tag{3.1}
$$

- Now from implementation perspective the ordered tuple Γ is essentially collection of 4 128 bit registers so $<< (1, 0)$ operation can be executed in effect by XORing Γ vectors with lane

vectors according to the shift without costing any extra operation at all. i.e

$$\begin{aligned}
\vec{L}_{jk} &\leftarrow \vec{L}_{jk} \oplus \vec{\Gamma}_{j-1 \bmod 4} \\
\vec{L}_{0k} &\leftarrow \vec{L}_{0k} + \vec{\Gamma}_3 \\
\vec{L}_{1k} &\leftarrow \vec{L}_{1k} + \vec{\Gamma}_0 \\
\vec{L}_{2k} &\leftarrow \vec{L}_{2k} + \vec{\Gamma}_1 \\
\vec{L}_{3k} &\leftarrow \vec{L}_{3k} + \vec{\Gamma}_2
\end{aligned} \tag{3.2}$$

- ρ_{west}

$$\begin{aligned}
P_1 &\leftarrow P_1 << (1,0) \\
P_2 &\leftarrow P_2 << (0,11) \\
P_1 &\leftarrow P_1 << (1,0) = \{\vec{L}_{31}, \vec{L}_{01}, \vec{L}_{11}, \vec{L}_{21}\} \\
P_2 &\leftarrow P_2 << (0,11) \\
&= \{\vec{L}_{02} <<< 11, \vec{L}_{12} <<< 11, \\
&\vec{L}_{22} <<< 11, \vec{L}_{32} <<< 11\}
\end{aligned} \tag{3.3}$$

- Similar to in θ the operation of $P_1 << (1,0)$ is executed by just changing the index of lane vectors.

- ι :

$$\begin{aligned}
A_0 &\leftarrow A_0 \oplus _rc \\
\vec{L}_{00} &\leftarrow \vec{L}_{00} + (_rc, _rc, _rc, _rc)
\end{aligned} \tag{3.4}$$

- χ : For all j

$$\begin{aligned}
P_0 &\leftarrow P_0 + (\overline{P_1}.P_2) \\
\vec{L}_{j0} &\leftarrow \vec{L}_{j0} + (\vec{L}^c_{j1}.\vec{L}_{j2}) \\
P_1 &\leftarrow P_1 + (\overline{P}_2.P_0) \\
\vec{L}_{j1} &\leftarrow \vec{L}_{j1} + (\vec{L}^c_{j2}.\vec{L}_{j0}) \\
P_2 &\leftarrow P_2 + (\overline{P}_0.P_1) \\
\vec{L}_{j2} &\leftarrow \vec{L}_{j2} + (\vec{L}^c_{j0}.\vec{L}_{j1})
\end{aligned} \tag{3.5}$$

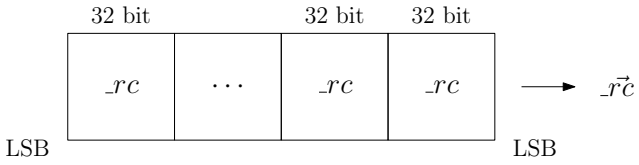

Figure 3.2: Round constant packing

- ρ_{east} :

$$\begin{aligned}
P_1 &\leftarrow P_1 << (0,1) \\
\vec{L}_{01} &\leftarrow \vec{L}_{01} <<< 1 \\
\vec{L}_{11} &\leftarrow \vec{L}_{11} <<< 1 \\
\vec{L}_{21} &\leftarrow \vec{L}_{21} <<< 1 \\
\vec{L}_{31} &\leftarrow \vec{L}_{31} <<< 1 \\
P_2 &\leftarrow P_2 << (2,8) \\
\vec{L}_{02} &\leftarrow \vec{L}_{02} <<< 8 \\
\vec{L}_{12} &\leftarrow \vec{L}_{12} <<< 8 \\
\vec{L}_{22} &\leftarrow \vec{L}_{22} <<< 8 \\
\vec{L}_{32} &\leftarrow \vec{L}_{32} <<< 8 \\
P_2 &\leftarrow P_2 << (2,0)
\end{aligned} \tag{3.6}$$

- Finally $P_2 << (2,0)$ is done by just changing the index same as before

$$P_2 \leftarrow \{\vec{L}_{2k}, \vec{L}_{3k}, \vec{L}_{0k}, \vec{L}_{1k}\} \tag{3.7}$$

Now since round constants are 32 bits so adding a round constant to a palne is equivalent to adding it to first lane so round constants are packed in the manner depicted in Figure and added to $\vec{L}_{00}$ for round constant addition.The no of 32 bit round constant packed is same as no of instances running in parallel i.e 1,4,8 or 16 depending on the registers available.

The Lane Slicing Implementation of Xoodoo pemutation in eXtended Keccak Code Package by the Keccak team contains 4 variants.

- 1 way: 1 lane (32 bit) is packed in a register.
- 4 way: 4 lanes (4.32=128 bit) are packed in a 128-bit register using AVX2 and AVX512 instruction set.
- 8 way: 8 lanes (8.32=256 bit) are packed in a 256-bit register using AVX2 and AVX512 instruction set.
- 16 way: 16 lanes (16.32=512 bits) are packed in a 512-bit register using AVX512 instruction set.

Remarks: So by keeping each lane in separate registers, lane sliced implementation avoids expending any operations on the lane rearrangements i.e $<<<(0, v)$ operation.

In Tab 3.2 we diplay a comparative study of the compilation time for all the functions under Hash mode and Keyed mode and the amount of data being processed.The measurements are in terms of CPU clock cycles per byte.

Register size	Bytes Processed	Hash		keyed			
		Absorb	Squeeze	Absorb	Squeeze	Encrypt	Decrypt
128 bit	192	8.28	8.302	2.895	5.343	8.385	8.364
256 bit	384	3.875	3.880	1.348	2.572	4.356	4.330
512 bit	768	1.335	1.335	0.489	0.882	1.570	1.526

Table 3.2: Measurement result for Lane Slicing

Chapter 4

Plane Slicing

Now let us consider another approach where instead of lanes we pack the same plane from the different instances of the ciphers running in parallel.So Suppose the state bits for i^{th} cipher are $b^i_{383}, \ldots, b^i_1, b^i_0$.Each 128 bits $b^0_{128*k+128}, \ldots, b^0_{128*k+1}, b^0_{128*k+0})$ are packed in a plane. Now just like in lane slicing here packed planes from different instances are packed in a register: $(b^n_{128*k+128}, \ldots, b^n_{128*k+1}$ $, b^n_{128*k+0}) \parallel \ldots \parallel (b^0_{128*k+128}, \ldots, b^0_{128*k+1}, b^0_{128*k+0})$ and there are 3 such planes. We denote the k^{th} plane of the i^{th} instance by $P_{ki} = (b^i_{128*k+128}, \ldots, b^i_{128*k+1}, b^i_{128*k+0})$.A graphical representation is depicted in Figure 4.2 .

$$\left|\begin{array}{cccc|c} b^i_{127} & \cdots & b^i_1 & b^i_0 & P_{i0} \\ b^i_{255} & \cdots & b^i_{129} & b^i_{128} & P_{i1} \\ b^i_{383} & \cdots & b^i_{257} & b^i_{256} & P_{i2} \end{array}\right|$$

Figure 4.1: Bits packed in a plane

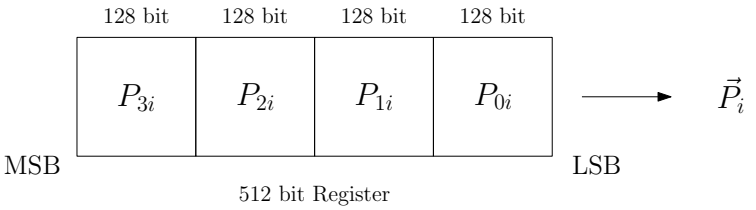

Figure 4.2: Planes from different ciphers packed in a register

Now as depicted in the Figure 4.2 P_{ij} indicates the P_j plane form i^{th} cipher . Consider $\vec{P_i} = (P_{i0}, P_{i1}, \ldots, P_{i3})$ a vector representation of the collection of planes from different ciphers.Now depending on the available register n=1 for 128 bit register,n=2 for 256 bit register and n=4 for 512 bit register.So like in lane slicing in plane slicing a plane vector is taken instead of a single plane and the operations are modified accordingly.The instruction are used in a manner such that same instruction is executed on the multiple planes packed in a register.

Now displayed in Table 3.1 are the definitions of the modified instructions for the vectorized representation .

$\vec{P}_y$	Plane y of the state
$\vec{P}_y << (t, v)$	$(P_{0y} << (t, v), P_{1y} << (t, v), P_{2y} << (t, v), P_{3y} << (t, v),)$
$\vec{P}_y^c$	$(\overline{P}_{0y}, \overline{P}_{1y}, \overline{P}_{2y}, \overline{P}_{3y})$
$\vec{P}_y + \vec{P}_{y'}$	$(P_{0y} \oplus P_{0y'}, P_{1y} \oplus P_{1y'}, P_{2y} \oplus P_{2y'}, P_{3y} \oplus P_{3y'})$
$\vec{P}_y.\vec{P}_{y'}$	$(P_{0y}.P_{0y'}, P_{1y}.P_{1y'}, P_{2y}.P_{2y'}, P_{3y}.P_{3y'})$

Table 4.1: Operation for Plane Vectorization

Now the modified definitions of the intermediate steps of the round functions are following:

- θ :

$$\begin{aligned} \vec{\Gamma} &\leftarrow \vec{P}_0 + \vec{P}_1 + \vec{P}_2 \\ \vec{E} &\leftarrow \vec{\Gamma} << (1, 5) + \vec{\Gamma} << (1, 14) \\ \vec{P}_y &\leftarrow \vec{P}_y + \vec{E} \end{aligned} \tag{4.1}$$

- ρ_{west}

$$\begin{aligned} \vec{P}_1 &\leftarrow \vec{P}_1 << (1, 0) \\ \vec{P}_2 &\leftarrow \vec{P}_2 << (0, 11) \end{aligned} \tag{4.2}$$

- ι :

$$\vec{P}_0 \leftarrow \vec{P}_0 + \_\vec{r}c \tag{4.3}$$

- χ :

$$\begin{aligned} \vec{P}_0 &\leftarrow \vec{P}_0 + (\vec{P}_1^c.\vec{P}_2) \\ \vec{P}_1 &\leftarrow \vec{P}_1 + (\vec{P}_2^c.\vec{P}_0) \\ \vec{P}_2 &\leftarrow \vec{P}_2 + (\vec{P}_0^c.\vec{P}_1) \end{aligned} \tag{4.4}$$

- ρ_{east} :

$$\begin{aligned} \vec{P}_1 &\leftarrow \vec{P}_1 << (0, 1) \\ \vec{P}_2 &\leftarrow \vec{P}_2 << (2, 8) \end{aligned} \tag{4.5}$$

The round constant is packed in a 512 bit register in the way depicted in Figure 3.2. i.e each 4th 32 bit segment of the 512 bit register contains the 32 bit round constant and rest is 0.

There are 3 ways of possible parallelization using plane slicing.

- 1 Ways : 1 128 bit plane is packed in a register and operations are executed using AVX and AVX2 instructions. This implementation is available in the eXtended Keccak Code Package.
- 2 Ways : 2 128 bit planes are packed in a 256 bit register and operations are executed using AVX and AVX2 instructions.
- 4 Ways : 4 128 bit planes are packed in a 512 bit register and operations are executed using AVX512 instruction set.

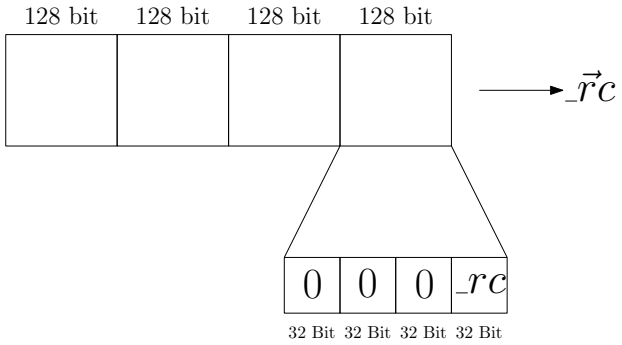

Figure 4.3: Round Constant Packed in a 512 Bit Register

We have implemented the 2 way and 4 way parallelization and compare them with 16 way lane slicing.

Results: The measurements for the plane slicing technique is given in Table 4.2.The Measurements are in terms of clock cycles per byte it takes to execute each functions under keyed and hash mode.

Register size	Bytes Processed	Hash		keyed			
		Absorb	Squeeze	Absorb	Squeeze	Encrypt	Decrypt
128 bit	48	14.375	14.33	4.125	9.166	14.458	14.5
256 bit	96	3.875	5.677	1.656	4.093	5.635	5.729
512 bit	192	1.335	2.031	0.531	1.291	2.343	2.552

Table 4.2: Measurement result for Plane Slicing

The code is publicly available at my github.

Chapter 5

Bit Slicing Implementation

Now we discuss parallelization at the level even smaller than 32 bits; we explore if we can improve the performance by parallelizing at the bit level.Let $b^i_{383}, \ldots, b^i_1, b^i_0$ be the bits of the state of the i^{th} cipher;now since we aim to parallelize to bit level we want to pack i^{th} bit from all states into a register,so consider a vector representation $\vec{b_i} = (b^0_i, b^1_i, \ldots, b^n_i)$, we call it bit vector.

	$\vec{b}_{383}$	$\ldots$	$\vec{b}_1$	$\vec{b}_0$
state 0	b^0_{383}	$\ldots$	b^0_1	b^0_0
state 1	b^1_{383}	$\ldots$	b^1_1	b^1_0
$\vdots$	$\vdots$		$\vdots$	$\vdots$
state n	b^n_{383}	$\ldots$	b^n_1	b^n_0

Table 5.1: Bit Vectors

5.0.1 Packing and Unpacking

Now we come to the implementation of packing and unpacking the state for bit sliced implementation.So there will be 512 many internal states; now suppose `uint8_t M[512][48]` be a 2-dim array where each 1-dim sub array represent a single state of 48 bytes or 384 bits i.e `M[i][]`$=(b^i_{383}, \ldots, b^i_1, b^i_0)$. Now since we want to load $\vec{b_i}$'s in the registers we need a data structure where we have all the i^{th} bits from different states are arranged together, so we re organize the `state` array into `uint8_t dest[384][64]` s.t `dest[i][]`$=(b^0_i, b^1_i, \ldots, b^n_i)$; so after packing the states in the `dest` each `dest[i]` is a pointer to the i^{th} bits of the states and we can load them in a register by a single load instruction.

Algorithm 16 swapmv(a, b, mask, n, temp)

1: temp = a $\oplus$ (b « n);
2: temp = temp & mask;
3: a = a $\oplus$ temp;
4: b = b $\oplus$ (temp » n);

It performs a controlled swap of bits between two variables a and b.The `smalltr` macro applies the `swapmv` [MPC00] operation repeatedly to transform an 8×8 block of bits in the matrix `M` and store

the result in the matrix `dest`. Here is what it does: Starts with the cell `M[j][i]` and successively applies `swapmv` to progressively deeper rows with increasing masks and shift amounts. After each stage,it stores the current result of `M[j+k][i]` into `dest` at a calculated position.Repeats this process for $k = 0$ to 7, effectively performing a bit-level transpose of the 8×8 block starting at (j, i).So after performing the transpose we place the `M[j+k][i]` at `dest[i*8+k][j/8]`.Since elements of the `state` is 8 bit words we basically take i^{th} bytes form state j through $j + 7$ and apply the `smalltr`. So after `smalltr M[j+k][i]` contains $(b_{i+k}^{j+7}, \ldots, b_{i+k}^{j+1}, b_{i+k}^{j})$ so we store it at `dest[i*8+k][j/8]`.now we iterate this process repeatedly to pack the unpacked state.

Algorithm 17 `Packing`$(M, dest)$

1: **Input:** A matrix $M[512][48]$
2: **Output:** A matrix $dest[384][64]$
3: Declare a temporary variable $temp$
4: **for** $i = 0$ to 47 **do**
5: **for** $j = 0$ to 63 **do**
6: Call `smalltr`$(i, j \times 8, temp)$
7: **end for**
8: **end for**

$\vec{b}_0$	b_0^n	$\ldots$	b_0^1	b_0^0
$\vec{b}_1$	b_1^n	$\ldots$	b_1^1	b_1^0
$\vdots$	$\vdots$		$\vdots$	$\vdots$
$\vec{b}_{383}$	b_{383}^n	$\ldots$	b_{383}^1	b_{383}^0

Table 5.2: After packing

5.0.2 Round Function and it's steps

Now since we packed each individual bit separately we can apply the bit rotation($<< (t, v)$ operation) of a plane by simply correctly changing the index.So we implement it by storing these indexes specific to each bit rotation in different arrays like for $<< (1, 5)$ shift_1_5 and shift_1_14 corresponding to $<< (1, 14)$ in general shift_t_v for $<< (t, v)$.In the algorithm $A0, A1, A2$ represents the 3 planes.The detailed execution of theta is explained here:

Algorithm 18 Theta

1: **for** $i = 0$ to 127 **do**
2: $P[i] \leftarrow A0[i] \oplus A1[i] \oplus A2[i]$
3: **end for**
4: **for** $i = 0$ to 127 **do**
5: $E[i] \leftarrow P[\text{shift\_1\_5}[i]] \oplus P[\text{shift\_1\_14}[i]]$
6: **end for**
7: **for** $i = 0$ to 127 **do**
8: $A0[i] \leftarrow A0[i] \oplus E[i]$
9: $A1[i] \leftarrow A1[i] \oplus E[i]$
10: $A2[i] \leftarrow A2[i] \oplus E[i]$
11: **end for**

Now sequentially before the χ there is bit rotation on plane 1 and 2 so we apply χ bit wise on all planes and store it in temp variables arrays B0,B1 and B2 other wise the bits on a plane gets changed before the whole χ layer is done. Since we are employing the bit rotation by re-indexing so an earlier bit getting update may mess up later calls to the plane arrays.

Algorithm 19 Chi

1: **for** $i = 0$ to 127 **do**
2: $B0[i] \leftarrow \text{Chi}(A0[i], A1[\text{shift\_1\_0}[i]], A2[\text{shift\_0\_11}[i]])$
3: $B1[i] \leftarrow \text{Chi}(A1[\text{shift\_1\_0}[i]], A2[\text{shift\_0\_11}[i]], A0[i])$
4: $B2[i] \leftarrow \text{Chi}(A2[\text{shift\_0\_11}[i]], A0[i], A1[\text{shift\_1\_0}[i]])$
5: **end for**
6: **for** $i = 0$ to 127 **do**
7: $A0[i] \leftarrow B0[i]$
8: $A1[i] \leftarrow B1[i]$
9: $A2[i] \leftarrow B2[i]$
10: **end for**

Now in round constant addition we change up things quite a bit.In this implementation we define round constants as an array of length 4 that stores the bit positions where the bit is 1.We take array of length 4 because it is the maximum no of 1's in any round constant.If a round constant have less 1's we just store $\perp$ in the empty positions of the array where $\perp$ is a position out side of the state.So we just XOR a register with all 1 to those particular locations of the planes mentioned in the round constant array.

Algorithm 20 Round Constant Addition

1: $A0[rc0] \leftarrow A0[rc0] \oplus \text{mask\_for\_not}$
2: $A0[rc1] \leftarrow A0[rc1] \oplus \text{mask\_for\_not}$
3: $A0[rc2] \leftarrow A0[rc2] \oplus \text{mask\_for\_not}$
4: $A0[rc3] \leftarrow A0[rc3] \oplus \text{mask\_for\_not}$

Algorithm 21 RhoEast

1: **for** $i = 0$ to 127 **do**
2: $B1[i] \leftarrow A1[\text{shift\_0\_1}[i]]$
3: $B2[i] \leftarrow A2[\text{shift\_2\_8}[i]]$
4: **end for**
5: **for** $i = 0$ to 127 **do**
6: $A1[i] \leftarrow B1[i]$
7: $A2[i] \leftarrow B2[i]$
8: **end for**

So in this implementation we have merged ρ_{east} and χ but after one round is complete the last operation of round 1 is ρ_{east} and the first operation of round 2 is θ so we merge them as Modified Theta.

Algorithm 22 Modified Theta

1: **for** $i = 0$ to 127 **do**
2: $P[i] \leftarrow A0[i] \oplus A1[\text{shift\_0\_1}[i]] \oplus A2[\text{shift\_2\_8}[i]]$
3: **end for**
4: **for** $i = 0$ to 127 **do**
5: $E[i] \leftarrow P[\text{shift\_1\_5}[i]] \oplus P[\text{shift\_1\_14}[i]]$
6: **end for**
7: **for** $i = 0$ to 127 **do**
8: $A0[i] \leftarrow A0[i] \oplus E[i]$
9: $B1[i] \leftarrow A1[\text{shift\_0\_1}[i]] \oplus E[i]$
10: $B2[i] \leftarrow A2[\text{shift\_2\_8}[i]] \oplus E[i]$
11: **end for**
12: **for** $i = 0$ to 127 **do**
13: $A1[i] \leftarrow B1[i]$
14: $A2[i] \leftarrow B2[i]$
15: **end for**

round () is called for the first round of the Xoodoo permutation.After round 1 is complete going foroward mod_round is called for all subsequent rounds except the last one . The last round is just a mod_round() with one additional rhoeast().

Algorithm 23 `round`$(rc0, rc1, rc2, rc3)$

1: Call `theta`()
2: Call `round_const_add`$(rc0, rc1, rc2, rc3)$
3: Call `chi`()

Algorithm 24 `mod_round`$(rc0, rc1, rc2, rc3)$

1: Call `mod_theta`()
2: Call `round_const_add`$(rc0, rc1, rc2, rc3)$
3: Call `chi`()

Algorithm 25 `final_round`$(rc0, rc1, rc2, rc3)$

1: Call `mod_theta`()
2: Call `round_const_add`$(rc0, rc1, rc2, rc3)$
3: Call `chi`()
4: Call `rhoeast`()

remarks: Now depending on the available registers n can be 128 for 128-bit register, n=256 for 256-bit registers and n=512 for 512-bit registers. We have only implemented n=512 version with AVX512 instruction set and measured the CPU Clock cycles per byte to be: **2.996** and plan to visit other versions in the future. The code is publicly available at my github.

Chapter 6

Conclusion And Comparative Study

6.1 Comparison of Implementation Methods

In this section, we present a detailed comparison of three distinct implementation strategies: *Lane Slicing*, *Plane Slicing*, and *Bit Slicing*. Each of these methods offers unique trade-offs in terms of parallelism, memory usage, and computational efficiency. Now, since all the functions in Keyed mode and Hash mode have their efficiency depending on the permutation at the core we explore it's performance in different implementation paradigms. In the 6.1Table: 6.1 we display the cycles per byte it takes to compute the permutation.

Register Size	Lane	Plane	Bit-Slicing
128 bit	3.03	3.87	_
256 bit	0.63	**2.47**	_
512 bit	0.434	**0.427**	**2.996**

Table 6.1: Xoodoo

Lane slicing technique excels in functions available in hash mode.Now lane slicing mainly pulls through because of three main reasons; one is the round function is faster in this implementation, second it deals with much more data than plane slicing so bytes per cycle gets reduced and lastly the output blocks are multiple of 4 bytes so having each lane in separate register gives for more flexibility during storing as well as well as masking i.e XORing. We have also implemented the Bit Slicing method for registers of size 512-bit and it takes **2.996** cycles per byte which is significantly worse than Lane Slicing so we conclude that from our experiments that the function available in the Hash Mode and Keyed Mode will be faster for Lane Slicing as well which can be seen form the 3.2Table 3.2 and 4.2Table 4.2

Also in Keyed mode $R_{kin} = 44$ and $R_{kout} = 24$ the input and out put block size being multiple of 4 bytes or 32 bits lane slicing gives all the advantages we talked about before in Keyed Mode as well. But in case of decryption one more crucial thing happens;in case of decryption the mask size is 24 bytes so after extracting 16 bytes or a plane worth of data we still need to extract 8 more bytes i.e lower half of plane 1 in plane slicing this poses a major challenge as it forces us to unpack the packed planes to access the lower 64 bit of each plane. But in lane slicing it just a matter of right choice of variables.

Remark: Although during our implementation we made sure to implement in such a manner that the output text will be unpacked i.e no extra computation will be necessary for unpacking output text which is not the case for the existing lane slicing implementation. Although we saw that it only increases the performance of the plane slicing implementation by a mere 3% with out the internal unpacking.But as lane slicing deals with significantly more data for a given register size the performance drop might be significantly larger for it.

Bibliography

[MPC00] Lauren May, Lyta Penna, and Andrew J. Clark. "An Implementation of Bitsliced DES on the Pentium MMX^{TM} Processor". In: *ACISP 00: 5th Australasian Conference on Information Security and Privacy*. Ed. by Ed Dawson, Andrew Clark, and Colin Boyd. Vol. 1841. Lecture Notes in Computer Science. Brisbane, Queensland, Australia: Springer Berlin Heidelberg, Germany, 2000, pp. 112–122. DOI: 10.1007/10718964_10.

[DSSS17] Fabrizio De Santis, Andreas Schauer, and Georg Sigl. "ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications". In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 2017, pp. 692–697. DOI: 10.23919/DATE.2017.7927078.

[Dae+20] Joan Daemen et al. "Xoodyak, a lightweight cryptographic scheme". In: *IACR Transactions on Symmetric Cryptology* 2020, Special Issue 1 (2020), pp. 60–87. DOI: 10.13154/tosc.v2020.iS1.60-87. URL: https://tosc.iacr.org/index.php/ToSC/article/view/8618.