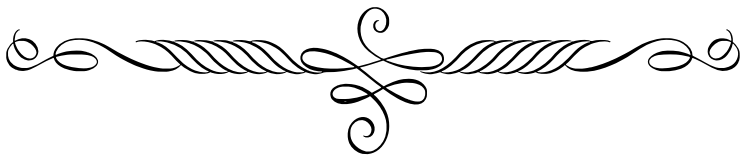
Technical University of Denmark
Department of Applied Mathematics and
Computer Science

Efficient Authenticated Encryption

Efficient implementation of PRIMATEs

Thesis Project



by
Jonas Bruun Jacobsen s122528

Abstract

It is a known issue with many modern block ciphers such as AES that they do not provide any assurances on the integrity and authenticity of the encrypted data. However, a family of block ciphers called authenticated ciphers exists, which has these goals as an integral part of their design. The research into this family of block ciphers is still limited and thus, a cryptographic competition named the CAESAR competition is currently being held to further research into this area. One of the submitted ciphers is PRIMATEs.

In this paper we have designed a bit sliced implementation of the 80- and 120-bit PRIMATEs permutation using the AVX2 instruction set on an Intel CPU with the Broadwell architecture. To do this, we had to find a bit-sliced implementation of the S-box used by PRIMATEs, as well as efficient bit sliced representations for the mix columns, shift rows and constant addition transformations.

We have designed three new modes of operation that can utilize the parallel processing in a bit sliced implementation to its fullest on single messages. We use these to see how efficient data can now be encrypted and decrypted.

We are able to achieve speeds that are a factor 31x faster than the non-bit sliced reference implementations of the three closely related modes. We are able to encrypt or decrypt data at speeds of 46 cycles per byte for the 80-bit security-level and 76 cycles per byte for the 120-bit security level. The bit sliced PRIMATEs permutation alone processes data at speeds of 44 and 74 cycles per byte over 6 rounds for the 80- and 120-bit security level, respectively.

Contents

1 Introduction

There has always been a need for secure communication, even before the digital age. This can be seen with the presence of devices such as the Enigma from the early twentieth century or even before that, back in the Roman Republic, where the dictator Julius Caesar has been said to employ a shift-cipher aptly named the Caesar Cipher [1].

Although there has always been a need to share messages in secret, the nature of cryptology has changed in the present times. Technology is moving fast and what is considered secure today, might not be so tomorrow. Furthermore, research in the field of cryptology has grown exponentially; both due to technology making continuously stronger ciphers breakable, but also due to the increased need for cryptology, even for the average user of the Internet.

Due to the rapid advancements in technology and in the field of cryptology – and due to a more connected world in general – it has been necessary to define standards to be used by the industry. A wide range of standards has therefore been chosen, each of which aims at fulfilling its own area of cryptology for which it was designed. A few of these major areas of cryptology as well as their standards are:

- Symmetric encryption; also commonly known as shared-key or secret-key encryption. AES is widely accepted as the standard in this area. It superseded DES as the standard, since DES was no longer sufficiently secure.[1] 3DES is still a standard, although AES is more efficient and widely the most used.
- Asymmetric encryption; also commonly known as public-key encryption. Diffie-Hellman is commonly used for key-exchange (e.g. in TLS 1.2 [3]) and RSA for asymmetric encryption and signature signing (if the public-key infrastructure for RSA is in place).
- Hashing: The currently approved hashing standards by NIST are SHA1, SHA2 and SHA3 [4]. They recommend not using SHA1 for anything other than legacy purposes [5].

All of these areas have their clearly defined strength and weaknesses. Symmetric encryption for instance is generally considered faster than asymmetric encryption, but requires that a secure key-exchange has found place beforehand. Asymmetric encryption does not have this issue, as the public keys can be shared openly (although a trusted public-key infrastructure is still needed to prevent man-in-the-middle attacks). Hashes has many uses and has been used for encryption as well (although the issue here is that they too are often slower than symmetric-ciphers) and e.g. for MACs.[2]

However, some areas of cryptology does not yet have what could be considered

[1] Since DES only has an effective key of 56-bits.
[2] Message authentication codes.

good standards in practice. Two of these areas are the areas of authenticity and integrity; the arts of verifying the source of data and protecting it against unintentional changes.
The current standards by ISO[3] uses MACs or digital signatures in conjunction with ciphers to provide authentication guarantees, but as these are not an integrated part of the cipher itself, they have to be built as a mode of operation around the cipher [6]. This separation leads to implementation risks in practice.

A new ongoing competition – the CAESAR competition[4] – intends to contribute to the research in this area, by calling for a submission of authenticated ciphers in the hope of creating a boost to this research area in cryptology.

One of the authenticated ciphers that has been submitted to this competition is named PRIMATEs. This paper will contribute to the research surrounding the PRIMATEs ciphers, by seeing how effectively the PRIMATEs permutation can be implemented, while remaining at least as secure as before. After all, a cipher should be secure, not only in its design, but also in its actual implementations. In addition, a cipher also needs to be efficient in practice, if it is to be widely adopted in the world of businesses, where speed is critical and computing power is considered a valuable resource.

The exact contributions of this paper will be:

A PRIMATEs S-box implementation that uses only logical operators

An implementation of the PRIMATEs S-box, which uses **only** the logical operators XOR and AND. An S-box that uses only logical instructions are ideal for the bit slicing technique, which we will use to implement the permutation used by PRIMATEs. To the best of our knowledge, it is also the fastest (publicly) suggested implementation of the PRIMATEs S-box for bit slicing. The S-box can be found in chapter 7.

A complete bit sliced PRIMATEs permutation and its inverse

As mentioned above, we will present a full bit sliced implementation of the PRIMATEs permutation and its inverse. We will implement the permutation for both the 80-bit and 120-bit security levels of PRIMATEs. This means that the *shift rows*, *constant addition*, *sub elements* and *mix columns* transformations will be bit sliced. To the best of our knowledge, this is the first time this is publicly done for the full PRIMATEs permutation on any of the security levels. The design process for bit slicing the permutation can be found in chapter 7.

Three new modes of operations for the PRIMATEs family

The current three modes of operation (for each security level) in the PRIMATEs suite of ciphers, cannot effectively utilize the bit sliced PRIMATEs permutation

[3] The International Organization for Standardization.
[4] *Competition for Authenticated Encryption: Security, Applicability, and Robustness*: http://competitions.cr.yp.to/index.html.

on single messages. Due to the design of the current modes, it is not possible to process multiple sections of the same message in parallel, which is what makes bit slicing efficient. We will suggest three new modes of operations for PRIMATEs, which can process multiple parts of a single message in parallel to speed up encryption and decryption. The three new modes of operation which we name *APE-BS*, *HANUMAN-BS* and *GIBBON-BS* will be closely related to the existing three modes and will also work on both security levels. The specifications and designs for the new modes can be found in chapter 8.

New performance results for PRIMATEs

We will publish the results achieved using the three new modes of operation, as well as the results for the underlying PRIMATEs permutation itself. All of these results will be for both security levels. The results will be a measurement of the performance in *cycles per byte*. Finally, we will compare these results to how they hold up against some other submissions of the CAESAR competition. The results can be found in chapter 9, while an evaluation of the results and the comparison to the other submissions can be found in chapter 10.

While all the design-choices and results of the above contributions can be found in this paper, the implementation can be found at: https://github.com/opolo/Bitsliced-Primates

2 Notation

Throughout this paper, a number of terms and abbreviations will be used. The special notations and abbreviations can be found in the list below:

- **Nonce**: *Number used once.* It is a random number, which is used just once.
- **AD**: *Associated data.* It is the data that can be given to an AEAD-cipher, which will not be encrypted, but which will still be authenticated.
- **AEAD**: *Authenticated encryption with associated data.* Whenever we have an authenticated cipher, which also takes as input associated data, it is referred to as an AEAD-cipher.
- **Scheme/Mode**: We use the word scheme and sometimes mode in place of *modes of operation.* Thus, whenever we refer to PRIMATEs' modes of operations, we refer to it as PRIMATEs' schemes or modes.
- **PRIMATEs-*s***: We will often use either the wording PRIMATEs-120 or PRIMATEs-80, when we refer to one of the two security levels of PRIMATEs.
- **Intrinsic**: An intrinsic is an assembly-coded function, which allows us to use function calls in our chosen programming-language in place of actually using assembly in the code.
- **AVX**, **AVX2** and **AVX-512**: *Advanced vector extensions.* These are instruction sets on Intel and AMD CPUs. AVX is the predecessor to AVX2, while AVX2 is the predecessor to AVX-512.
- **YMM**: When the term appears in code-examples, it is a type definition for the data-type `__m256i`. The type can hold data, which can be loaded into an YMM register. When we use the term YMM outside the code-examples, we refer to the 256-bit registers available in CPUs that support the AVX2 instruction set.
- **XMM** and **ZMM**: Registers used by the AVX and AVX-512 instruction sets, respectively. While they will not appear in code, they will appear in the paper during instruction set comparisons.
- **SIMD**: *Single Instruction Multiple Data.* It is one of the commonly terms used for parallel processing of data using the registers of SSE, AVX and similar instruction sets.
- **Processed**: Whenever a scheme has processed some data, we think of the data as having both been *encrypted* **and** *decrypted* again.
- **S-box**: *Substitution box.* An S-box is a cryptographic construction used to disrupt the linearity in encryption algorithms. They contain a one-to-one mapping between a finite set of inputs and outputs.

- **10^* bit-padding**: Whenever this padding is applied to data, it means that a 1-bit is appended directly after the data, followed by as many 0's of padding as is needed.
- **LSB**: *Least significant bit.* The LSB of an integer means its lowest bit.
- **MSB**: *Most significant bit.* The MSB of an integer means its highest bit.
- **AES**: *Advanced Encryption Standard.* A symmetric encryption algorithm that was chosen by NIST (the National Institute of Standards and Technology) as the encryption standard to supersede DES (another block cipher). It was developed by Joan Daemen and Vincent Rijmen [2].

3 Previous Work

3.1 Authenticated Encryption

Many ciphers have been designed with the limitation that they do not directly support authentication of the processed message.
This means that an attacker can flip a bit in a ciphertext, and the ciphertext would decrypt into something else. Those that are able to decrypt the ciphertext are not able to verify that the message has not been tampered with (i.e. that its integrity is intact), nor that the source of the message is correct (although in many practical cases, flipping a bit will have the ciphertext decrypt to garbage in the eyes of the receiver, which can be a telltale sign of tampering with the message). However, this is not a new way of attacking ciphers and as such, this has already been researched and addressed to some extent. Some more common approaches to mitigate this, is by either signing the data or through the use of MACs.

Signing the messages can work, but requires a trusted public-key infrastructure to be in place in order to verify the signature. MACs on the other hand are more simple to use in this situation. A MAC is at its core simply a string of bits, created with an algorithm that has as its input the message and the secret key used to encrypt it. Only those with knowledge of the secret-key will thus be able to create a tag and likewise, only those with knowledge of the secret-key will be able to verify them (and as a result verify the integrity of the message as well).
MAC algorithms today are often based on hashing algorithms, as they are a natural fit for the task of "taking an arbitrary sized message and key and mapping them to a MAC of a fixed-size".

The above definition of MACs has some (intended) ambiguities.
What sort of message is it that should be used as input for the MAC? One possibility is the plaintext, while another is the ciphertext. Also, if the plaintext was being used as the input, should the MAC then be encrypted along with the plaintext afterwards? Some research has gone into this, which has led to 6 schemes on how to handle a MAC together with a cipher being standardized in 2009 [6].[5] An example of the present-day use of these standardized schemes, is the GCM scheme that is being used with AES today in the TLS[6] 1.2 protocols [7].

Even then, there are still issues with authenticated encryption. One of these is that the prior schemes are not part of the encryption-scheme itself, so it is up to the individual users to choose an encryption and authentication scheme to use, and afterwards implement the system properly. Another issue is that some schemes are incompatible with some ciphers. As an example, GCM is only defined for block ciphers which uses block sizes of 128 bits (such as AES).

[5] These are: OCB 2.0, Key Wrap, CCM, EAX, Encrypt-then-MAC (EtM) and GCM.
[6] Transport layer security.

With all these issues with the present-day standards in this field, it is easy to understand the reason behind the CAESAR competition.

3.2 Efficient Encryption

While security in some form is always the end goal when using cryptology, other parameters matters as well. One of these is efficiency.

A great deal more iterated product ciphers would still be secure today, had they performed a 1,000 iterations instead of perhaps just 10. However, as secure as that may be, this is not a feasible approach in practice. An algorithm that can process ten times the amount of data with the same security guarantees, will for most practical applications be preferred over the slower algorithm. Thus, ciphers needs to be designed with efficiency in mind and fortunately, this has led to a lot of research in both efficient designs and implementations.

Of course, efficiency is a broad term and does not just mean speed (which is usually measured in cycles per byte). It can also mean efficiency in terms of memory-usage, power-consumption or even code-size, if the algorithm is to be run on a lightweight system. It could also mean a low gate-count for hardware-implementations.
In this paper, we will focus only on optimizing the PRIMATEs family for speed. The only two exceptions to this will be:

- When gains in speed makes the implementation unfeasible for other practical purposes. We will not accept a better cycles/byte processing ratio, if the trade-off is that the implementation will no longer be able to execute in any realistic environment due to e.g. an abnormally large RAM usage.
- When lowering the other factors will also have a positive effect on the speed. An example is if we could reduce the amount of used registers to avoid register-swapping in the CPU, or shrink some data to reduce cache-swapping.

To return to the topic of previous research, some research that quickly comes to mind, is the research published by Emilia Käsper and Peter Schwabe on optimizing AES [8]. In their paper they suggest implementing AES by utilizing a bit slicing approach, and using this approach, they set the record for the fastest AES implementation. Their bit sliced implementation is currently still the fastest AES implementation, which does not use the AES-NI instruction set.[7] They suggest an approach, where they bit slice the AES state into CPU registers, and are thus able to process 8 AES blocks in parallel on the CPU. They utilized the XMM registers for this. This is very akin to what we did for our PRIMATEs implementations, as we also bit sliced our state into registers as well. We use the YMM registers and the AVX2 instruction set for our implementations. Details on our implementation can be seen in chapter 7.

Another paper regarding efficient implementations is submitted by Ko [9]. He analyzes the S-boxes from the CAESAR competition. In his paper, he analyzed

[7] The AES-NI instruction set provides hardware-acceleration for the AES cipher and exists on newer CPUs.

the S-boxes of some submissions and optimized them for different factors, such as multiplicative complexity and bit sliced gate complexity. One of the S-boxes he did optimize, was the one from PRIMATEs, but he did not find an efficient bit sliced implementation for it. Nonetheless, his work helped us in the right direction, when it came to implementing the S-box for our implementation.

Finally, we would like to highlight an issue that we often encountered, when searching for previous work in this field. Research on efficient implementations was problematic, as the approaches presented are often heavily optimized for one specific platform. Likewise, many efficient implementations goes down to the hardware-level, which was seldom of use for a software-implementation.

4 PRIMATEs

Before going into depth with the efficient implementation of the PRIMATEs permutation, we will cover what exactly the PRIMATEs family of ciphers consist of.

4.1 Overview of PRIMATEs

PRIMATEs is a family of ciphers, which was submitted to the CAESAR competition by Elena Andreeva et al [10]. It is one of the ciphers, which has made it to the second round of the competition.

The PRIMATEs family of ciphers consist of 3 schemes built on the PRIMATEs permutation, namely: *APE*, *HANUMAN* and *GIBBON*.
The authors' recommendation for lightweight authenticated encryption is HANUMAN, while their recommendation for lightweight authenticated encryption where speed is critical is GIBBON. They recommend APE for lightweight authenticated encryption, where additional security is needed.
The recommendation are roughly based on the fact that APE uses a larger key, while GIBBON's permutation uses half the number of rounds of the other two schemes.

Each scheme also supports two different security levels; a *120-bit* level and a *80-bit* level. Table 4.1 shows the different key, tag, and nonce-sizes for the different schemes under the 80-bit and the 120-bit security level.

	APE-s	HANUMAN-s	GIBBON-s
k (key size)	2s	s	s
t (tag size)	2s	s	s
n (nonce size)	s	s	s
PRIMATE	p_1	p_1, p_4	p_1, p_2, p_3

Table 4.1: The key, tag, nonce size and used permutations for the different schemes under different security levels. The table is identical to table 2 in the PRIMATEs submission paper [10].

The tag in table 4.1 is a MAC, which is created alongside the ciphertext for verifying the authenticity of the message (and its associated data), when it is decrypted. The last row in the table shows the underlying PRIMATEs permutation(s) that the scheme uses. All the 4 permutations (p_1, p_2, p_3, p_4) uses the same transformations, but the amount of rounds for each permutation as well as the round-constants varies. Table 4.2 shows the amount of rounds and starting value of the round-constant for each. Appendix A shows a full table of all the round constants for each permutation.

	p_1	p_2	p_3	p_4
Number of rounds	12	6	6	12
Initial round-constant	1	24	30	24

Table 4.2: Rounds and starting round-constant for each PRIMATEs permutation. The table is identical to the one in section 2.4 in the PRIMATEs submission paper [10].

4.2 Schemes of PRIMATEs

The schemes of PRIMATEs are all built on an identically structured internal state. Internally, all the schemes are sponge-ciphers that keeps an internal state of 7×8 elements of 5-bit each for the 120-bit security level, and a state of 5×8 elements for the 80-bit security level. A state is split up into a *rate* and a *capacity* part. This can be seen in figure 4.3.

r	r	r	r	r	r	r	r
c	c	c	c	c	c	c	c
c	c	c	c	c	c	c	c
c	c	c	c	c	c	c	c
c	c	c	c	c	c	c	c
c	c	c	c	c	c	c	c
c	c	c	c	c	c	c	c

Table 4.3: The internal state of PRIMATEs for the 120-bit level. The first row contains the rate, the remaining rows contains the capacity. Each element is 5 bits wide.

The 8 elements of the rate – each element consisting of 5 bits – is the part of the cipher that is directly interacted with for processing data. How each scheme in PRIMATEs does this interaction vary, however.

For GIBBON and HANUMAN, the bytes of the rate is XORed to 5 bytes of the plaintext to create 5 bytes of ciphertext. Each time it is done, the state (with the new rate) undergoes a PRIMATEs permutation (table 4.1 shows which) before 5 bytes more of ciphertext can be produced in the same way.

For APE we do the same as for GIBBON and HANUMAN, but we do not have the ciphertext, until after the permutation. After the permutation, the resulting bytes of the rate is our ciphertext.

In both cases the tag is produced by XOR of the key with the capacity of the final state. Since the key is only half the size for HANUMAN and GIBBON, the key is XORed to the first half of the capacity and only half of the capacity is hereafter returned as the tag.

The key difference between these two approaches – at least for our optimized implementations – is that HANUMAN and GIBBON each uses the same PRIMATEs permutations for both encryption and decryption, while APE needs its

inverse PRIMATEs permutation for decryption. Since APE technically needs to *rewind* through its permutations to inverse the encryption, the tag is also needed in APE for the actual decryption and not just for the verification of the data. After all, when the key is XORed to the tag, the tag of APE contains the capacity of its final state, which will be the initial state (along with the last ciphertext as rate) for the decryption.

Lastly, since not all messages can be expected to be integral, all schemes of PRIMATEs apply a bit-wise 10^* padding.

4.3 PRIMATEs Permutation

The permutation that is applied on the internal state of PRIMATEs is a traditional SPN.[8] It is composed of the following 4 transformations:

$$SE \circ SR \circ MC \circ CA$$

This sequence of transformations (which will be described briefly below) is the same for all the PRIMATEs permutations (p_1, p_2, p_3 and p_4). The only thing that differs between the different PRIMATEs permutations is the number of rounds, as well as the round-constants used.

Constant Addition (CA): The constant addition transformation XORs a constant to the second element of the second row each round. The sequence of constants is a predefined sequence, which, as mentioned earlier, differs between the PRIMATEs permutations. Appendix A contains a table of all the sequences.

Shift Rows (SR): The shift rows transformation is a simple transformation that shifts the rows of the state element-wise to the left. For the 120-bit security level, the row i is shifted $s_i = \{0, 1, 2, 3, 4, 5, 7\}$ elements to the left. For the 80-bit level, the row i is shifted $s_i = \{0, 1, 2, 4, 7\}$ elements to the left.

Mix Columns (MC): The mix columns transformation is a left-multiplication between the state and a predefined multiplication matrix (which can be seen in Appendix A). The multiplications are performed over the Galois field $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. The matrix multiplication is applied 5 times with the PRIMATEs-80 security level, and 7 times for PRIMATEs-120.

Sub Elements (SE): Sub elements is regular substitution using an S-box, which can be seen in appendix A.

[8] Substitution-permutation network.

5 Bit Slicing

5.1 Why Bit Slicing?

After much consideration, we deemed that using bit slicing for implementing the PRIMATEs permutation would be our best choice. The logic behind this choice, was that other similar SPN sponge ciphers (such as AES which is nearly identical in its SPN) has shown good results being bit sliced [8].
We also valued highly in our considerations the fact that both GIBBON and HANUMAN uses only the forward PRIMATEs permutations to encrypt and decrypt data. In case bit slicing proved time-consuming, we could focus on heavily optimizing only the forward permutation and still improve both the encryption and decryption of HANUMAN and GIBBON. Of course this would mean that the decryption of APE would not be as optimized, since it uses the inverse permutation to do decryption. However, APE was not as fit for bit slicing as GIBBON and HANUMAN to begin with (which is explained in chapter 8). Nonetheless, we implemented APE as well.

Aside from these reasons, a bit sliced implementation also holds strengths against some side-channel attacks. If we could make a bit sliced implementation of PRIMATEs, we could prove that it was possible to make resilient implementations against these attacks. Since bit sliced implementations have constant time complexity in regard to the data-content (only the data-size matters) and uses no lookup-tables, bit sliced implementations are not prone to cache-timing nor timing attacks.

In this chapter, we will explain how bit slicing works. In the next chapter, we will go into depth with how we bit sliced the permutation.

5.2 Bit Slicing in Detail

Bit slicing is an implementation technique that mimics in software how an algorithm would have been implemented in hardware using only logic gates. This means that a cipher is rewritten using only logical instructions that works on the bit-level – such as AND, OR and XOR – just as it would have been implemented via hardware-gates.

The bit slicing technique was first used by Eli Biham in 1997, when he suggested it is a technique for implementing DES [11]. The results of his bit sliced implementation of DES that he published with his paper, was about 5 times faster than the currently fastest DES implementation at that time. However, Eli never used the term bit slicing in his paper to refer to his technique and as such, the name itself seems to have been coined later by Matthew Kwan [12]. Regardless of where the name originated from, it is easy to see how that name was chosen.

In a bit sliced implementation, all the matching bits of all the elements in a state is separated into their own registers. Thus, in the case of PRIMATEs, 5 registers would be needed to store all the bits of a state. This is because all the LSBs (least significant bits) of the elements goes into the same register, all the MSBs (most significant bits) goes into the same register, and so forth.

Of course, this alone yields no advantages in speed; likely it is the contrary, as bit slicing the bits of the elements into the registers has an overhead.
The efficiency in bit sliced implementations comes from the fact that a single register can contain more than just the bit of a single element. Since the AVX2 instruction set was launched (with Intel's Haswell CPU architecture in 2013 [14]), the registers have had a length of 256 bits. This means that for our S-box example, when we perform the bit sliced S-box transformation just once on our registers, we perform it on all 256 bits in each register concurrently. This means that if the registers were filled with 256 PRIMATEs elements, we would perform the transformation on 256 elements simultaneously.

It is easy to see and understand why it always takes constant time for a bit sliced implementation to process some data. Regardless of the value of the bits in our registers in the above example, we always perform the exact same logic instructions on all the bits of the registers at once. This removes conditionals in code which can lead to timing attacks. Since it also removes the lookup-table of the S-box in favor of logical operations, it also removes cache-timing attacks.

While this sounds ideal for most, if not all, cipher implementations, bit slicing is not ideal for every cipher. In order for bit slicing to be effective for a cipher, two issues needs to be addressed effectively:

1. An effective way of representing the state in the registers needs to be found.
2. An effective way of implementing the transformations for the state using only logical operations on the registers needs to be found.

These two issues are heavily linked to each other so for clarity, we will explain them as one below.

The issue with finding an effective way of representing the state in the registers, is due to it being insufficient to simply bit slice the elements randomly into the registers. It will work for transformations such as the S-box, which operates on single elements, but transformations such as shift rows and mix columns operates on multiple elements.
Figure 5.1 and 5.2 illustrates two different bit sliced PRIMATEs-120 states, where shift rows needs to be implemented in two different ways.

Row 0										...	Row 6									
Col 0			Col 1			...	Col 7			...	Col 0			Col 1			...	Col 7		
State 0	...	State 7	State 0	...	State 7	...	State 0	...	State 7	...	State 0	...	State 7	State 0	...	State 7	...	State 0	...	State 7

Table 5.1: A register. The lowest row is the bits of the register and indicates which state the bit is from. The first (leftmost) bit is thus from row 0, column 0 and state 0.

State 0											...	State 7										
Row 0			Row 1			...	Row 6			...	Row 0			Row 1			...	Row 6			
Col 0	...	Col 7	Col 0	...	Col 7	...	Col 0	...	Col 7	...	Col 0	...	Col 7	Col 0	...	Col 7	...	Col 0	...	Col 7

Table 5.2: A register. The lowest row is the bits of the register and indicates which state the bit is from. The first (leftmost) bit is thus from row 0, column 0 and state 0.

In table 5.1 the bits are stored in the register such that shift rows can be performed by just a byte-shuffling instruction. The lowest unit of data, which can be accessed in AVX2, is the byte, which is ideal here, as all 8 bits of the matching element from the 8 different states are stored within a byte. By doing a byte-shuffling here, we can shuffle the individual elements around in the state.

In table 5.2 the bits are stored in the register such that in a single byte, the 8 bits of a row from the same state is stored together. This means that in order to shift the rows within a state, the bytes in the register should be bit-wise rotated to the left. They should be rotated different indices to the left as well, depending on which row the individual byte represents. Unfortunately, there is no bitwise-rotate operation on bytes in AVX2, and thus no easy way to do this. Also, as was mentioned earlier, the lowest data-unit which can be accessed with AVX2 is the byte, so there is no way to do a bit-shuffling.

As seen in the former paragraphs, even shift rows can prove complex depending on the bit sliced design of the state. This is just one of the issues (the available register instructions) which should be taken into account, when deciding the bit-sliced representation of the state. Another is the platform it is going to be designed for. With AVX2 on Intel CPU architectures, it is not possible to do a byte-shuffling across the 128-bit lane inside the 256-bit wide YMM registers, not even with byte shuffling. If bytes should be shuffled across the middle of the register, the permute operation should be used instead, which is 3 times as costly. This is due to the YMM registers on Intel CPUs actually being 2 registers. The XMM registers from the older SSE instruction sets makes up the first 128 bits, while a new 128-bit register is combined with the XMM register to create the 256-bit wide YMM registers for AVX.

Section 7.1 walks through how we chose the design of the bit sliced state for our implementations.

6 Technologies used

In the coming chapters, we will go into depth with the design of the bit sliced permutation, the bit sliced schemes and finally the benchmarks for both of these. In order to fully understand these chapters, it is necessary to first have an idea of the technologies used in the implementations, thus we will cover those relevant for all three topics here.

6.1 Programming Language

The implementation has been written purely in C and utilizes Intel Intrinsics for access to the AVX2 instruction set. The choice fell on C for a number of reasons, namely:

- Due to the speed-factor. C compiles directly to machine-code unlike languages such as Java or C#, which compiles to intermediate byte-code for their respective virtual machines.
- The reference implementations for the schemes of PRIMATEs was made in C++. It was easier getting started on the implementations, by using a language similar in structure to the existing implementations.
- The Intel Intrinsic library is designed for C and C++.

Intel Intrinsics is a well-documented publicly available library designed by Intel for use with C-style languages [15]. The library provides a wide range of well-documented intrinsic functions. The intrinsics of Intel Intrinsic are inline functions that are understood by the compiler, and which gets substituted with assembly instructions at compile time. Thus, by using Intel Intrinsics, we can use the instruction sets of the CPU in C-code, without writing assembly instructions to do so.

We chose to use Intel Intrinsics, since it is a publicly available and documented library, and by keeping the code free of assembly, we believed it would both improve the readability of the code and the debugging of it.
Finally the compiler likely knows better on how to integrate the intrinsics into the code, and thus produces much more effective code than we would have by manually implementing the instruction calls in the code.

Based on all the above reasons, the choice was either C or C++. Since we were most comfortable with C of those two, the choice fell on C.

6.2 Instruction Sets

Intel Intrinsics makes a wide array of instruction sets available, herein all the SIMD instruction sets: SSE, SSE2, SSE3, SSSE3 AVX, AVX2, AVX-512, etc.

We have no interest in the older SSE instruction sets, as they operate on the XMM registers in the CPU. These are only 128-bits wide. The AVX and AVX2 instruction set uses the newer YMM registers, which are 256-bit wide and thus more ideal for containing the bit-sliced states.
The AVX instruction set contains mostly the floating point operations, whereas AVX2 contains most of the logical integer operations, so the bit sliced implementations will be using AVX2. AVX will in a few situations be used, namely when we load data into the YMM registers using 8-, 16-, 32- or 64-bit data types, as these intrinsics are located in AVX.

The AVX-512 instruction set would have been more ideal to use than AVX2, as the full bit-sliced state described in section 7.1 could have been contained in the registers. The reason for not choosing AVX-512, is due to it only being available on Intel Xeon processors (with the Skylake architecture) currently, which we did not have access to. AVX-512 is scheduled to become available in consumer CPUs with Intel's Cannonlake architecture, which is scheduled for the second half of 2017 [13].
Had AVX-512 been used, we would have expected speeds almost double of those with AVX2. Fortunately, the current implementation should be easily adaptable to AVX-512 once it is released.

We invite anyone to improve the implementation when AVX-512 is available, as it would be interesting to see which results can be obtained with the newer instructions and registers.

6.3 Compiler

The ICC (Intel C++ compiler) is used to compile the code. The choice fell upon this compiler due to it compiling the fastest code in our tests. The other compilers we tested were MSVC[9] and GCC[10]. When we compiled the bit sliced implementations with ICC, we observed an approximately 34% increase speed, compared to MSVC. Code compiled with MSVC ran approximately 4-5% faster than code compiled with GCC.
In order to make these results reproducible for others, we have designed our code such that it can be compiled with all three compilers.

The code is being compiled for the x64 architecture. This is necessary as the intrinsics `_mm256_set_epi64x` and `_mm256_set1_epi64x` for loading 64-bit integers into the registers, do not work with the x86 architecture.

For all the bit-sliced implementations, the following compiler flags for optimizations was used:

- /O3 – Maximize the code for speed.
- /QxHost – compile the code for the current machines CPU architecture. This flag allows us to use AVX2 when compiled on machines, where it is available.

[9] Microsoft Visual Studio Compiler.
[10] Gnu Compiler Collection.

There are many other optimization flags for each of the compilers. Most often these do not need to be set, as they are already included with the `/O3` flag [16]. There are also some experimental optimization flags such as `-Ofast` with GCC, which can sometimes improve the speed. However, in our case the speed degraded when using these flags.

7 Bit Sliced PRIMATEs Permutation

7.1 Chosen bit sliced state

Finding an efficient state for PRIMATEs was challenging. The challenge is that while some transformations are efficient on one state, some other transformations might not be. Thus, finding an efficient state comes down to finding a state, where all the different transformations as a whole are the most efficient. The first serious outline for a state (for PRIMATEs-120) was the one shown in figure 7.1.

State n																					Byte
Row 0			Row 1			Row 2			Row 3			Row 4			Row 5			Row 6			Free
Col 0	...	Col 7	Col 0	...	Col 7	Col 0	...	Col 7	Col 0	...	Col 7	Col 0	...	Col 7	Col 0	...	Col 7	Col 0	...	Col 7	Free

Table 7.1: The first outline for a PRIMATEs-120 state. The bottom line columns represents the bits in the register. Only 64 bits (a single state) of a register are shown, since the design repeats itself for each 64-bits. For PRIMATEs-80 the rows 5 and 6 would be free as well.

The logical reasoning that caused us to use this state initially is straightforward: The state of (120-bit security level) PRIMATEs consists of 7 rows and 8 columns – each consisting of 5-bit PRIMATEs elements. This gives a total of 56 5-bit elements, and hence the full size of a state is 280 bits.
In chapter 5, it was explained that since a PRIMATEs element consists of 5-bits, 5 registers are necessary. We use the AVX2 instruction set and the 256-bit wide YMM registers, thus we can contain a total of 1280bits of state in our 5 registers.

Based upon on this logic, it was natural that we at first attempted a bit sliced representation of PRIMATEs, which would allow us to store 4 complete 120-bit states into the 256-bit wide registers.

We did consider the fact that 5 registers can also contain 6 states of PRIMATEs with the 80-bit security level. This did at first seem more impressive, but after attempts at packing the 6 states effectively into the registers, we decided this was not the case.
One big issue is that accessing and storing data in/from the registers are usually done in *8-bit* (byte), *16-bit* (word), *32-bit* (dword) or *64-bit* (qword) sizes. For PRIMATEs-120 the qword was ideal, since a 280-bit state uses 56-bit in each register, however for PRIMATEs-80 there is no ideal unit.
PRIMATEs-80 uses 40 bits per register, and thus still needs to use a qword (of which there are only 4 in a register) to load and store the 6 states, which would

give us some difficulties in transposing the states to the registers effectively.
The YMM registers also has some limitations when it comes to performing instructions across the 128-bit lanes of the registers. Splitting a state across the 128-bit lanes of the registers would lead to worse performance and would perhaps pose implementation difficulties.
These issues and a lower security level made us opt for bit slicing only the 120-bit PRIMATEs permutation for this register-layout.
We cannot rule out that a 6-states 80-bit security level implementation could have been made, however, which can challenge our final implementation and chosen state, so we invite anyone to try to beat our results this way and would like to hear more, if it is done.

After choosing the state from table 7.1, attempts were made at implementing the transformations efficiently for this state.
The S-box was trivial, as it would work for any bit-sliced state.
Constant addition was trivial as well, since the only thing that matters for constant addition, is where the element that is XORed with the constant is located in the registers.
Shift rows proved to have some difficulty, since the AVX2 instruction set does not have any byte rotate instructions. These have been added to the AVX-512 instruction set that supersedes AVX2, but it was not available to us, as discussed in chapter 6. AMD processors do have rotate instructions available with their XOP instruction set, but we did not have any AMD processors available to see if this could be utilized.
Despite this, we found a satisfying byte-rotation implementation, which in short utilized multiplication with powers of two 2 to left-shift the bytes differently (depending on the amount that the row (byte) should be shifted) into a new register. We then used the `_mm256_blendv_epi8` intrinsic to select the different bits from the two registers in such a way that we would get a rotate effect and not a shifting effect. This required 6 instructions plus a multiplication constant for each row.[11]
We were never able to find an efficient implementation for doing the mix columns transformation for this state, before settling for a new state.

While researching and designing the mix columns transformation for the older state, we started to gain interest in how the bit sliced state proposed by Emilia Käsper and Peter Schwabe for AES was designed [8].
We had already considered their state earlier, but their design requires a byte in each of the registers for each element in the state, and thus we had opted not to use it. Figure 7.1 shows their design.

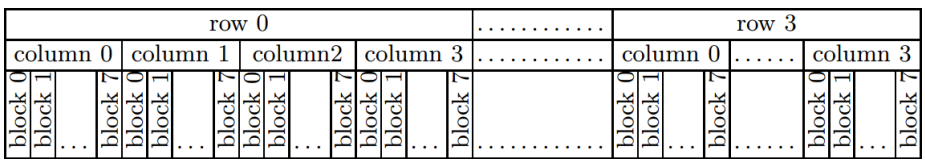


Figure 7.1: The bit sliced state layout used by Käsper and Schwabe for AES. The figure is figure 1 from their paper.

[11] However, keep in mind that `_mm256_mullo_epi16` (the multiplication intrinsic we used) is 5 times more costly than e.g. the XOR intrinsic. `_mm256_blendv_epi8` is twice as costly.

The reason we opted not to use it, is that since they use one byte in each register for each element, we would need 56 bytes in each register to implement PRIMATEs-120 and 40 to implement PRIMATEs-80 (and then this design would support up to 8 states in parallel, since each byte has 8 bits).
There are only 32 bytes in the YMM registers – which were insufficient – and this had made us move away from this approach at first.

When we revisited this design, we considered how it would affect performance, if we used 2x YMM registers in extension to achieve 512-bit registers, which was what we needed.
The disadvantage of doing this, is that we would have to apply each bit sliced transformation twice on the state - once for the first half of our 512-bit registers and once for the second half.
A second disadvantage would be that when performing mix columns, the transformation would need to access elements from both the first and second half of the registers. This is more costly than accessing values within the same register.

One counter-argument against these issues is that we also encrypt twice the amount of states in parallel with this approach, so applying the same transformations twice is acceptable from a performance point of view. The former 4x bit-sliced states could encrypt 20 bytes at once (due to 4x rates of 5 bytes), while the new encrypts 40 bytes at once.
The only issue here is that if we encrypted some data, which were a multiple of 20, but not 40. Then we could have saved ourselves one set of permutations, by utilizing one set of 5x registers. However, we found this difference negligible for anything but the smallest data-sizes.

Another argument for the new state was that we could potentially be able to find faster implementations for mix columns and shift rows. Shift rows could clearly be implemented more efficiently in the new state, using only two byte-shuffling operations (one for each half of the 512-register).

We could also implement PRIMATEs-80 easily with this state, as the only difference between PRIMATEs-80 and PRIMATEs-120 layout-wise in the registers would be whether the bytes 8-23 in the second set of registers would be free or used as capacity. And only mix columns would need some adjusting here, which would be necessary anyway, since they use different matrices. The disadvantage of using this register-layout for PRIMATEs-80, is that we can only expand with 8 new states at a time. We could only fit 8 PRIMATEs-80 states into the registers, even though there were nearly enough space for 16 states.

Finally, the biggest counter-argument is the coming of the AVX-512 instruction set. AVX-512 is as mentioned earlier, currently only available in Intel Xeon server CPUs, but is scheduled for consumer Intel CPUs for the second half of 2017 (with the Cannonlake CPU architecture). In AVX-512, the registers **will** be 512 bits wide and can therefore contain the full 8 bit sliced states of both security-levels. This means that the bit sliced transformations will have to be applied just once to the registers, which will almost double the speed of this bit sliced implementation.

Based on these arguments, we felt that the state similar to Käsper and Schwabe's was a better choice and a more future-proof design to go for. The final bit sliced states for our bit sliced implementations can be seen in figure 7.2 and 7.3.

Row 0										...	Row 4									
Col 0			Col 1			...	Col 7			...	Col 0			Col 1			...	Col 7		
State 0	...	State 7	State 0	...	State 7	...	State 0	...	State 7	...	State 0	...	State 7	State 0	...	State 7	...	State 0	...	State 7

Table 7.2: The layout of the state in each register for PRIMATEs-80. For simplicity, it is omitted that the state is split in two groups of 256-bit wide registers, instead of a single group of 512-bit wide registers

Row 0										...	Row 6									
Col 0			Col 1			...	Col 7			...	Col 0			Col 1			...	Col 7		
State 0	...	State 7	State 0	...	State 7	...	State 0	...	State 7	...	State 0	...	State 7	State 0	...	State 7	...	State 0	...	State 7

Table 7.3: The layout of the state in each register for PRIMATEs-120. For simplicity, it is omitted that the state is split in two groups of 256-bit wide registers, instead of a single group of 512-bit wide registers

7.2 Bit Sliced Transformations

At the heart of the schemes of PRIMATEs lies the PRIMATEs permutation. The amount of CPU time that will be spent outside of the permutation is minuscule, compared to the time spent performing it. Therefore, the permutations are the main focus of this paper. The remaining code outside the permutations mainly deal with initializing the states, loading data into/out of the registers and creating the tag.

In the following subsections we will explain how we designed the bit sliced implementations (for our bit sliced state) for each of the transformations in the PRIMATEs permutation.

7.2.1 Constant Addition

The constant addition transformation was trivial to bit slice due to its simplistic nature. The most challenging part of bit slicing the constant addition transformation was to make sure that the 5 registers that contains the constant bits, had the correct bits raised for their constants and at the correct place in the registers. We needed to place the constants in the registers, such that they would be XORed with the second element of the second row in the states.

As an example of this, let us have a look at how the constant $0x17$ (decimal value 23) would be stored in the registers. It is the round constant used by p_1 in its seventh round.
In binary $0x17$ is written as $0b10111$. This means that we need to raise the bits of the byte at the second-row second-element index in all the round-constant registers, except the one that contains the 4th bit.

Since our state registers has the layout shown in section 7.1, the bits for the second element of the second row (for the 8 states) are stored at byte-index 9 in

the registers. This leads to the C-code shown below for our example, where we XOR the state with the round-constant registers with their bits correctly raised for the value $0x17$.

```
YMM rc_bit_0 = _mm256_set_epi64x(0, 0, 0xFF00, 0);
YMM rc_bit_1 = _mm256_set_epi64x(0, 0, 0xFF00, 0);
YMM rc_bit_2 = _mm256_set_epi64x(0, 0, 0xFF00, 0);
YMM rc_bit_3 = _mm256_set_epi64x(0, 0, 0, 0);
YMM rc_bit_4 = _mm256_set_epi64x(0, 0, 0xFF00, 0);

state_bit_0 = _mm256_xor_si256(state_bit_0, rc_bit_0);
state_bit_1 = _mm256_xor_si256(state_bit_1, rc_bit_1);
state_bit_2 = _mm256_xor_si256(state_bit_2, rc_bit_2);
state_bit_3 = _mm256_xor_si256(state_bit_3, rc_bit_3);
state_bit_4 = _mm256_xor_si256(state_bit_4, rc_bit_4);
```

We always use the intrinsic `_mm256_xor_si256` for XORing two YMM registers. We use the intrinsic `_mm256_set_epi64x` for when we want to load 4x 64-bit values into a 256 bit register. In the actual implementation, the code is slightly more complex due to:

- Multiple rounds each with a different constant.
- The state is too large to be kept in 5x 256-bit YMM registers.

The first issue is handled by having the round-constants in 5 arrays (one array for each bit). Each array index contains the register for the round-constant for that given round. When a round needs its matching constant, it XORs the state registers with the registers in the arrays at the matching index.
The second issue is no problem, since the constant is only located in one of the halves (the first half) of the state. Thus, we disregard the second half of our state in the constant addition transformation.

7.2.2 Shift Rows

Due to the design of the bit sliced state, one byte in the registers contains the bit from the same elements from each of the 8 states. Thus, to perform the shift rows transformation, only byte shuffling within the registers is needed.
It is not possible to shuffle bytes across the 128-bit lane in an YMM register (the permute intrinsic can do it, but it has triple the calculation latency). However, since a row consists of 8 elements, this means that the rows in the registers will be 64-bit aligned and this will not become an issue.

As the bit sliced state does not fit into 5x 256-bit wide registers, we need to split the state and use $2x5$ 256-bit registers. Therefore, byte-shuffling needs to be done separately on the first and second half of the state. Furthermore, the rows in the second half needs to be shuffled differently than the rows of the first half, as each row is shifted a different amount of elements. This leads to the C-code below for PRIMATEs-120 with two different shuffle-masks:

```
YMM shuffleMaskFirst = _mm256_setr_epi8(
     0,  1,  2,  3,  4,  5,  6,  7,
```

```
     9, 10, 11, 12, 13, 14, 15,  8,
    18, 19, 20, 21, 22, 23, 16, 17,
    27, 28, 29, 30, 31, 24, 25, 26);
YMM shuffleMaskSecond = _mm256_setr_epi8(
     4,  5,  6,  7,  0,  1,  2,  3,
    13, 14, 15,  8,  9, 10, 11, 12,
    23, 16, 17, 18, 19, 20, 21, 22,
    255, 255, 255, 255, 255, 255, 255, 255);

for (int reg = 0; reg < 5; reg++) {
    //State[Bit][Half]
    state[reg][0] = _mm256_shuffle_epi8(state[reg][0],
        shuffleMaskFirst);

    state[reg][1] = _mm256_shuffle_epi8(state[reg][1],
        shuffleMaskSecond);
}
```

We always use the intrinsic `_mm256_shuffle_epi8`, when we want to do byte-shuffling inside a register without crossing the 128-bit lane. The parameters of _mm256_shuffle_epi8 specifies, where each byte in the register should get its new value from. The second shuffle-mask thus causes the first byte in the registers to take the value the fifth byte had before the shuffling.

If you instruct the shuffle intrinsic to get its value from a byte with an index too high (such as index 255 in the second shuffle-mask), the byte takes the value zero. We do this in the second mask, since we do not use the last 8 bytes in the second set of registers.

The only difference between the shift rows transformation for PRIMATEs-120 and PRIMATEs-80 is the shuffle masks used.

7.2.3 Sub Elements

The difficulty of implementing the bit sliced sub elements transformation comes down to the difficulty of finding an efficient bit sliced representation of the S-box; i.e. a representation of the S-box that works by using only logic operators such as AND, XOR, OR and NOT on the input bits.

Ko Stoffelen has found optimized S-box implementations of the round 2 candidates (herein also PRIMATEs) of the CAESAR competition for various criteria [9]. Some criteria he optimizes the S-boxes for are: Multiplicative complexity, bit sliced gate complexity, gate complexity and depth.

While he listed PRIMATEs' optimized S-box implementation for multiplicative complexity, he did not list the optimized one for bit sliced gate complexity. We attempted to reproduce his results with the purpose of creating an efficient bit sliced S-box.
We managed to express PRIMATEs' S-box in formulas written in CNF[12] using

[12]Conjunctive normal form.

his tools.[13] Formulas in CNF form can be solved using SAT-solvers[14] (if they have a solution), of which many both commercial and free exists.

Despite using various SAT-solvers for PRIMATEs CNF-formulas, the SAT-solvers were unable to find a solution within what we believed was reasonable time (a couple of days). To verify these results we contacted Ko Stoffelen, which confirmed that he was unable to find an efficient bit sliced S-box representation for PRIMATEs.[15]

We decided on another approach and constructed the ANF[16] formulas shown in appendix C for our PRIMATEs S-box.

The ANF formulas for our S-box could be converted directly into a bit sliced implementation using only AND and XOR. It would use a total of 42 logical instructions and 5 assignments (assuming it does not use temporary variables to assign temporary results to).

We did some manual nudging of the above formulas in our implementation to reduce the formulas to use 33 logical instructions and 16 assignments instead. We did this by introducing temporary variables. It seemed to improve the S-box implementations speed, but the difference was so small we cannot rule out that it was simply due to noise.
The circuits with the temporary variables can be seen in the final substitute elements transformation in the implementation.

Since we also decided to implement APE, we also had to bit slice the inverse S-box, as APE needs the inverse permutations as well. We took the same approach as for the normal S-box, and ended up with the ANF formulas for the inverse S-box that are also shown in appendix C.

The S-box directly constructed from those ANF formulas would cost a total of 134 instructions and 5 assignments (assuming it does not use temporary variables to assign temporary results to).

As with the old S-box, we did some manual nudging of the formulas by using temporary variables, and managed to change the instruction count to 74 logical instructions and 34 assignments to temporary variables, which also seemed to give better results. (If the compiler uses temporary variables in the formulas, our implementation would be better still, as it has no redundant temporary variables with the same values).
This final circuit for the inverse S-box can also be seen in the implementation, in the inverse substitute elements transformation.

The inverse S-box is not as effective as the normal S-box. Along with the larger tag (described in section 8.2), this makes a bit sliced version of APE less interesting than GIBBON and HANUMAN.

Note that there is no bit-wise NOT operation in the AVX2 instruction set. We worked around this by XORing the registers with a register of all 1's, each time

[13] Available at: https://github.com/Ko-/sboxoptimization.
[14] Boolean satisfiability solvers.
[15] The question and the answer can be found at Github under *closed issues*: https://github.com/Ko-/sboxoptimization.
[16] Algebraic normal form.

we needed a NOT instruction.
There exists an ANDNOT intrinsic, but as all the AND'ed values are used in multiple equations, we could not use ANDNOT on an existing temporary variable. We would have to create a new temporary variable, which negates the instruction that we would have saved by using the ANDNOT intrinsic.

The final intrinsics we ended up using for the S-box implementation were `_mm256_xor_si256` and `_mm256_and_si256`.

7.2.4 Mix Columns

Mix columns in the bit sliced version of AES by Käsper and Schwabe were able to exploit the circularity of the matrix used in the AES mix columns transformation [8]. This allowed them to design a formula which could be applied to each element of AES' state to achieve the mix columns transformation. Applying this formula to their registers, they only spent 43 operations in total on the mix columns step for each round.
PRIMATEs' mix columns matrix does not have the same circularity, and no other bit sliced mix columns transformations, had found any shortcuts applicable to this case. As a result of this, we implemented the mix columns transformation in its traditional way, by performing the matrix multiplication over $F_{2^5} \cong F_2[x]/(x^5 + x^2 + 1)$.

Fortunately, it is not costly to do the multiplications on every element over $F_{2^5} \cong F_2[x]/(x^5 + x^2 + 1)$. Multiplying the whole state with 2 can be done with 2 XOR operation and 10 assignments. It can be done with 1 XOR and 5 assignments, had the registers been 512-bits wide instead, as the simplified example below shows:

```
//Simplifying code by assuming the state can be in 5x YMM.
//The variable state[5] already exists in this example.
YMM T2_state[5];

//We do not need to XOR the first, since we would XOR with 0.
T2_state[0] = state[4];
T2_state[1] = state[0];
T2_state[2] = _mm256_xor_si256(state[1], state[4]);
T2_state[3] = state[2];
T2_state[4] = state[3];
```

Since the mix columns matrix (which is located in appendix A) for PRIMATEs-120 contains 4 different values that needs to be multiplied to the registers: 1, 2, 9, and 15, we would need a total of 46 XOR operations and 50 assignments to get the multiplied values.[17] PRIMATEs-80 would require 18 XOR operations and 30 assignments.[18] All of these numbers would be halved on AVX-512.

[17] T2, T4 and T8: 2 XOR and 10 assignments each. T9: 10 XOR and 10 assignments. T15: 30 XOR and 10 assignments.
[18] T2, T4, T8 and T16: 2 XOR and 10 assignments each. T18: 10 XOR and 10 assignments.

After multiplying the elements in the registers with the values of the matrix, we shuffle the bytes in the registers upwards in the columns. This is due to the design of the mix columns matrices (which can be seen in appendix A). Unfortunately, we have to shuffle bytes across columns and not across rows, so we cross the 128-bit lane in the YMM registers, and have to use the more expensive permute intrinsic.
After the shuffling, the elements of the last row are computed (which is now empty due to shuffling the elements upwards). We align all the relevant multiplied values of the state and XOR these together to find the values for the last row.

The above matrix multiplication is applied 7 times in the PRIMATEs-120 permutation and 5 times in PRIMATEs-80, which means that we are essentially performing the following computation (for PRIMATEs-120 as an example):

$$State = ((((((State \times A1) \times A1) \times A1) \times A1) \times A1) \times A1) \times A1 \tag{7.1}$$

Even if the matrix is a simple one, this computation is expensive to repeat 5 or 7 times. Fortunately, it is possible to optimize it.

Since we multiply with the same matrix seven or five times in a row – depending on the security level – we can do the math beforehand and multiply the matrices together. This means that instead of multiplying the state with the same matrix seven times for PRIMATEs-120, as was shown in equation 7.1, we could compute the matrix $A2 = A1 \times A1$ beforehand for PRIMATEs-120 and instead perform the shorter computation:

$$State = (((State \times A2) \times A2) \times A2) \times A1 \tag{7.2}$$

Or perhaps it is more efficient to calculate the matrix $A3 = A1 \times A1 \times A1$ and do the computation:

$$State = ((State \times A3) \times A3) \times A1 \tag{7.3}$$

It is possible to compute all the intermediate matrices $A1$ ... $A7$ for PRIMATEs-120 and $A1$ ... $A5$ for PRIMATEs-80, but each matrix becomes increasingly complex. Performing the computation with $A7$ might not be faster than e.g. the computation in equation 7.3 with $A3$. As an example of this: in $A1$ we need to multiply the registers with 4 different values (1, 2, 9, 15), while for $A7$ we need to multiply with 23 different values.

We did some benchmarking for each of the matrices to see, which one would be fastest for the permutation for both PRIMATEs-120 and PRIMATEs-80. This benchmarking can be seen in table 7.4, while all the variant matrices for mix columns and its inverse variant matrices can be found in appendix B.

Level	Matrix	Clock cycles	Index	Mutual Index
80-bit	A1	15884	100	100
	A2	24912	217	217
	A3	42508	240	240
	A4	49084	261	261
	A5	56064	318	318
	Optimized A1	NA	NA	NA
	Optimized A2	24448	204	204
	Optimized A3	37020	209	209
	Optimized A4	45116	233	233
	Optimized A5	47512	271	271
120-bit	A1	16000	100	101
	A2	43432	271	273
	A3	49308	308	310
	A4	77260	482	486
	A5	83972	524	528
	A6	101172	632	636
	A7	100040	625	629
	Optimized A1	NA	NA	NA
	Optimized A2	38136	238	240
	Optimized A3	42432	265	267
	Optimized A4	69320	433	436
	Optimized A5	74240	464	467
	Optimized A6	91208	570	574
	Optimized A7	89808	561	565

Table 7.4: The time it took to do a computation with the different mix columns matrices for PRIMATEs-80 and PRIMATEs-120. Keep in mind that while A1 is the fastest for both security-levels, it needs to be applied five or seven times.

It is clear from the benchmarks that no combination of multiplication with any of the matrices would be faster than a single multiplication with the $A7$ matrix for PRIMATEs-120 (or $A5$ for PRIMATEs-80), so we implemented the mix columns transformation using these matrices.

We managed to optimize some of the matrices, herein also the matrices that we ended using. The optimization utilizes the fact that some values only needs to be multiplied to one half of the states, thus we do not always need to multiply both halves. As an example: Matrix $A7$ used in PRIMATEs-120 only needs the number 25 multiplied to values, which are stored in the second half of the state. This is an optimization that is unique for the AVX2 implementation. It would not work with an AVX-512 implementation, where the 8x PRIMATEs states can be stored in 5x registers. AVX-512 would still be faster though.

8 Bit Sliced PRIMATEs Schemes

8.1 Design Challenges

One of the biggest strengths with bit sliced implementations, their speed, is also one of the drawbacks, due to how it is achieved. For our case, it is achieved by applying the PRIMATEs permutations to 8 states in parallel. Some issues this brings are:

- 8 messages needs to be processed in parallel for optimal efficiency. Should the cipher wait until there are 8 messages waiting? Would the system then be idle, while it waits for 8 messages to be batched?
- If the messages (or their associated data) are of different lengths, the effectiveness drastically decreases. The parallel processing needs to finish for all 8 messages, before the processing of new messages can start.
- How can we create the correct tags from the final states of each message, if we continue transforming the states until the last message has finished processing?

The last issue is the least troublesome of those mentioned. A solution could be to raise a flag each time one the messages had finished processing. The state could then be stored in separate registers. When all 8 states had finished, the tags could then be produced in parallel.
However, this would have some overhead. Double the amount of registers would be needed to store the finished states, while the other keeps being processed (and some latency would be introduced, since the tags would not be produced before the last state had finished encrypting).

The first and second issues are harder to solve, since there is no ideal solution for both of them. Messages could be batched, such that messages (nearly) equal in length would be processed together. This would somewhat mitigate the second issue, but worsen the first.

What can be done then? Looking at the 3 issues identified earlier, they all have one thing in common: The problem is not with the parallelism of bit slicing, but with the need for multiple messages. If it was possible to parallelize the processing of a single message, these issues would be solved. We came up with 3 schemes that utilize this approach, which will be described in section 8.2.

8.2 Design Choices

In our attempts at finding a solution for parallel processing of a single message, we initially came up with the design shown in figure 8.1.

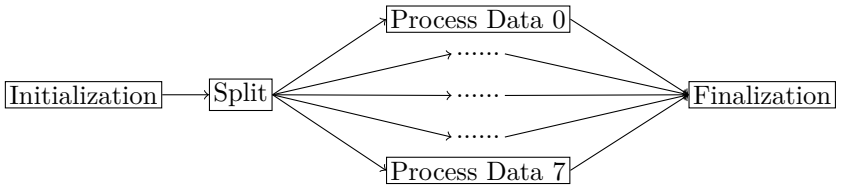


Figure 8.1: Initial overall design of a PRIMATEs scheme for parallel processing of single messages. The state is split into eight identical states, which handles the data split into eight parts.

By splitting a single message into eight parts and processing these parts in the eight identical initial states, we could process a single message in parallel. However, this approach caused new issues. These were:

- The ciphertext would no longer be the same for any of the schemes, as the data were not fed in the same sequence to a single state anymore.
- How would the tag be created, since we now have 8 different states to create the tag from?
- Was it okay from a security-wise perspective to process 8 parts of the message in parallel?

We found no way to work around the first issue. All the current schemes in PRIMATEs work with the state in a CBC[19] similar fashion; that is, the state of the cipher for the next round, depends on the state from the round before it. If we were to be able to produce the same ciphertext as the reference implementations, it would require that the final first state was the initial state of the second. The final second state was the initial state of the third, and so forth. This is not possible, as the states are processed in parallel.

Facing the dilemma of either having to accept that only different messages could be processed in parallel – or make slight changes to PRIMATEs current schemes to allow bit sliced implementations and solve the issues listed earlier – we chose the second option of making changes in the current schemes. The resulting ciphertext will be different from the reference implementations of the current schemes, which means that we are essentially designing three new schemes. The three new schemes will still utilize the same underlying PRIMATEs permutation as the existing schemes, and they will be built heavily upon the design of the existing schemes.
We have named the new schemes APE-BS, HANUMAN-BS and GIBBON-BS to show their close relationship with the current PRIMATEs schemes.

As we are now designing new variants of the current schemes, we have more

[19]Chain block cipher.

freedom in optimizing and solving the issues we found with the bit sliced implementations. Each issue we found will have its own following subsection.

8.2.1 Tags

HANUMAN-BS and GIBBON-BS

In order to create the tag, we first XOR the 8 parallel states together into a single state. The tag is then created from this single state as GIBBON and HANUMAN do in the PRIMATEs submission paper.

However, there is one deviation in how we create the tag, when compared to HANUMAN and GIBBON. While GIBBON and HANUMAN returns the bits of the tag in the "element order", we return them in the "bit-significance order", where all the LSBs of all tag-elements are returned first, starting with the first element. The only reason the bits are returned in this order, is because the bits are ordered this way in the registers and thus this is faster. Otherwise, we would have to blend the bits together properly. Both ways are shown in table 8.1 for PRIMATEs-80.

Input	Elements used in tag for PRIMATEs-80										
Element	PRIMATEs Element 0					...	PRIMATEs Element 15				
Bit	P0,0	P0,1	P0,2	P0,3	P0,4	...	P15,0	P15,1	P15,2	P15,3	P15,4

Tag	Return order (GIBBON & HANUMAN)										
Bit	P0,0	P0,1	P0,2	P0,3	P0,4	P1,0	P1,1	P1,2	P1,3	P1,4	...

Tag	Return order (GIBBON-BS & HANUMAN-BS)										
Bit	P0,0	P1,0	P2,0	P3,0	P4,0	P5,0	P6,0	P7,0	P8,0	P9,0	...

Table 8.1: The order of the first bits of the tag returned by HANUMAN & HANUMAN-BS and GIBBON & GIBBON-BS.

We could have extracted 1/8 of the tag from each existing state to create the final tag. This would have been faster, as we would not have had to use resources on XORing the states together. We opted against this, as each eighth part of the created tag would have its bits independent of the other bits. While this might have been acceptable security-wise, it is still something that would have required a proper security-analysis in order to be acceptable.

APE-BS

APE-BS is more complex when it comes to the tag.

Due to the design of APE (described in chapter 4), the tag is needed for the initial state in the decryption. This means that when processing 8 states in parallel, there is no way to reduce the tag to the same size as the non-bitsliced APE, which has only one state. Due to this, we chose the same bits as APE

does in the state – but for all 8 states – and process them in the exact same way as APE.

This leads to the tag returned from APE-BS being of 1920 bits in size for APE120-BS and 1280 bits for APE80-BS. In contrast, the tag of APE80 is 160 bits and 240 bits for APE120.
This, along with the issue that APE-BS also needs to have a bit-sliced inverse of the permutation it uses (p_1), is the main drawbacks of a bit sliced APE scheme.

8.2.2 Security

One important security issue arose from processing different sections of the same message in parallel; namely, that even though different parts of the message were processed in each state, each state would still use the same key, nonce, and AD. The issue is then that if the same data repeated itself in two (or more) sections of the plaintext – and these identical sections were being processed in parallel in their respective states – the encryption would produce the same ciphertext for these sections. This is not acceptable, as it reveals information about the content of the plaintext in the ciphertext. The issue is very akin to the issue which occurs when someone uses the ECB[20] mode of operation with block ciphers.

We worked around this issue in the new bit sliced schemes, by initializing each state slightly different from the others. Each state is initially XORed with a different constant and afterwards the state undergoes a PRIMATEs permutation. It is important that we do a PRIMATEs permutation after adding the constant to each state to avoid the weakness shown in table 8.2. If the states do not undergo a PRIMATEs permutation after adding the constant, the data added to the states after the constants can be chosen such that the states would become identical again. If the states undergo a PRIMATEs permutation after the constants are added, an attacker would no longer know what data to deliberately choose to exploit this weakness.

Constants	1000010	0000101
Chosen message	0111101	1111010
Identical XOR results	1111111	1111111

Table 8.2: Two different messages selected such that XORing with the different state-constants creates the same parallel states.

We chose the first eight round-constants used in the PRIMATEs permutation p_1 as constants for this. These constants were chosen based on the fact that they were chosen to be secure enough for the permutation, hence we considered them sufficiently secure for this purpose as well. This choice is acceptable because if we assume that the PRIMATEs permutation maps the same input to the same output, and the mapping is randomly enough for no patterns to be recognized in the mapping between input and output, then the choice of the round-constants would not matter at all, as long as they make the states different from each other.

[20] Electronic code book.

Using the same constants as those in the permutations also made the implementation simpler, as these constants are already loaded into data-types for the YMM registers due to them being used in the permutation.

Instead of using predefined constants, we could have used an initialization-vector with the constants. We opted against this, as we wanted the designs to be very close to the original schemes of PRIMATEs. Adding another input-parameter to the design of APE-BS, HANUMAN-BS and GIBBON-BS would make them more distinct from the original schemes.

8.2.3 Input Bit Slicing

Since we no longer have the requirement to produce the same output, we can optimize the way we bit slice (transpose) the data into the registers. Usually in a bit-sliced implementation in order to produce the same output as the non-bit-sliced implementation, the data is transposed into the registers as shown in figure 8.3. However, as we have no need for transposing the data this way to achieve the same ciphertext, we can optimize the design.

Input										
Element	PRIMATEs Element 0					PRIMATEs Element 1				
Bit	P0,0	P0,1	P0,2	P0,3	P0,4	P1,0	P1,1	P1,2	P1,3	P1,4

Registers	Register bits									
Reg.0	P0,0	P1,0	...	...	...	...	...	...	...	...
Reg.1	P0,1	P1,1	...	...	...	...	...	...	...	...
Reg.2	P0,2	P1,2	...	...	...	...	...	...	...	...
Reg.3	P0,3	P1,3	...	...	...	...	...	...	...	...
Reg.4	P0,4	P1,4	...	...	...	...	...	...	...	...

Table 8.3: The PRIMATEs elements needs to be split up in bits, and each bit of each element inserted in its own register.

The bit sliced implementation of PRIMATEs needs 5 registers as a minimum (one for each bit in a PRIMATEs element), thus each register holds 1/5 of the rate of the states. As the rate of one PRIMATEs state is 5 bytes and there are 8 states, we can process 40 bytes of data at a time.
Instead of transposing 40 bytes of data into the registers – which requires both bit-wise AND (for extracting the individual bits from the input-bytes) and bit-wise shifts (to make space in the registers) for each bit – we instead just load 5 bytes of message directly into each of the rates, as can be seen in table 8.4.

Input	Data									
Element	PRIMATEs Element 0					PRIMATEs Element 1				
Bit	P0,0	P0,1	P0,2	P0,3	P0,4	P1,0	P1,1	P1,2	P1,3	P1,4

Registers	Register bits									
Reg.0	P0,0	P0,1	P0,2	P0,3	P0,4	P1,0	P1,1	P1,2	P1,3	P1,4
Reg.1	...	...	...	...	...	...	...	...	...	...
Reg.2	...	...	...	...	...	...	...	...	...	...
Reg.3	...	...	...	...	...	...	...	...	...	...
Reg.4	...	...	...	...	...	...	...	...	...	...

Table 8.4: The PRIMATEs elements no longer need to be split up in bits one at a time and inserted into the respective registers. Now we can insert the 5 bytes of data directly into the registers, thus removing overhead from transposing the input to the registers.

In the "traditional" bit sliced sense, this means that the initial data (before we transposed it to the registers) had been something else, but this does not matter to us. We just need to transpose the ciphertext the same way, when we decrypt the data again.

Due to this design, we now technically have a single rate of 40 bytes. This means that if there are less than 40 bytes of data available, we pad the data with $10*$ padding to have 40 bytes of data to load into the registers.

8.2.4 Design Conclusion

Quite a few different design-choices was present in the former subsections, so for clarity these are summarized as a whole here.

Figure 8.2 shows the final high-level design for all the new single-message PRIMATEs schemes.

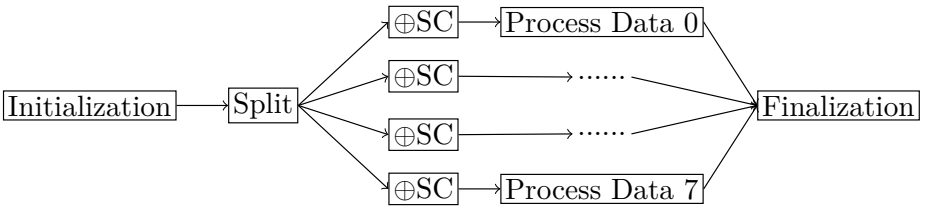


Figure 8.2: Final overall design of a PRIMATEs scheme for parallel processing of single messages. The state is split into 8 identical states, which handles the data split into 8 parts. Different state constants are added to each state to ensure that different states with the same data does not encrypt to the same ciphertext.

We have the *initialization*, where the key and nonce are processed identically for all 8 states. This process is done similar to how the current PRIMATEs

schemes perform it.
Following this we have the *split* and *state-constant*, where each state is made differently from the others with a constant. As was mentioned in the security-section, it is important to do a PRIMATEs permutation afterwards.
After this comes the *processing* of the AD and the message. Each state is permutated the same way the original PRIMATEs schemes permutates a state, since it is the same underlying permutations performed.
Finally we create the tag from the final states in the *finalization*.

Section 8.3 shows the lower-level algorithms for the new 3 individual bit sliced schemes and the specifications of their input and output.

8.3 Scheme Specifications

Since we have designed some new schemes in this chapter, we will list the input and output specifications, as well as the algorithms for these new schemes here.

Table 8.5 shows the final input and output specifications for the new schemes.

s = Security level	APE-BS	HANUMAN-BS	GIBBON-BS
k (key size)	$2s$	s	s
t (tag size)	$8s$	s	s
v (nonce size)	s	s	s

Table 8.5: Input/Output specifications for APE-BS, HANUMAN-BS and GIBBON-BS.

The remaining sections of this chapter shows the lower-level algorithms for the 3 bit sliced schemes. They will use some terms which does not exist in the original PRIMATEs paper. The explanations for these are:

- Expand: Expands data into all 8 states at the same place. "$V \leftarrow Expand(0^r||K||N)$" will thus expand the key and nonce identically into the capacity of all 8 states.
- AddSC: Add state constants. This method adds the **different** state constants to the rate of each state.
- Combine: Combine the 8 parallel states into a single state by XORing them together.
- V_R: When a capital R is used, it means the rates of all 8 states as one huge rate. The only exception is with the notation "$V_R||V_C$", which means all the individual rates, appended with all the individual capacities.
- V_C: When a capital C is used, it means the capacity of all 8 states as one huge capacity. The only exception is with the notation "$V_R||V_C$", which means all the individual rates, appended with all the individual capacities.

Finally, whenever the nonce, AD or message is split into sections, these sections will be of 40 bytes each in contrast to the schemes of the original PRIMATEs paper, which split the data into blocks of 5 bytes.

8.3.1 APE-BS

Algorithm 1 $\mathcal{E}_K(N, A, M)$

Input: $K \in C$, $N \in C^{\frac{1}{2}}$, $A \in \{0,1\}^*$, $M \in \{0,1\}^*$
Output: $C \in \{0,1\}^*, T \in 8C$

$V \leftarrow Expand(0^r \mid\mid K)$
$V \leftarrow AddSC(V)$
$V \leftarrow p_1(V)$
$N \leftarrow N \mid\mid 10^*$
$V \leftarrow p_1(N \oplus V_R \mid\mid V_C)$
if $A \neq \varnothing$ **then**
    $A[1]A[2]\dots A[u] \leftarrow A$
    $A[u] \leftarrow A[u] \mid\mid 10*$
    for $i = 1$ to u **do**
        $V \leftarrow p_1(A[i] \oplus V_R \mid\mid V_C)$
    end for
end if
$V \leftarrow V \oplus Expand(0^{b-1} \mid\mid 1)$
$M[1]M[2]\dots M[w] \leftarrow M$
$\ell \leftarrow |M[w]|$
$M[w] \leftarrow M[w] \mid\mid 10*$
for $i = 1$ to w **do**
    $V \leftarrow p_1(M[i] \oplus V_R \mid\mid V_C)$
    $C[i] \leftarrow V_R$
end for
$C \leftarrow C[1]C[2]\dots C[w-2]$
$C \leftarrow C \mid\mid \lfloor C[w-1] \rfloor_\ell$
$C \leftarrow C \mid\mid C[w]$
$T \leftarrow V_C \oplus Expand(0^r \mid\mid K)$
return (C, T)

Algorithm 2 $\mathcal{D}_K(N, A, C, T)$

Input: $K \in C$, $N \in C^{\frac{1}{2}}$, $A \in \{0,1\}^*$, $C \in \{0,1\}^*, T \in 8C$
Output: $M \in \{0,1\}^*$ or $\perp$

$IV \leftarrow Expand(0^r \mid\mid K)$
$IV \leftarrow AddSC(IV)$
$IV \leftarrow p_1(IV)$
$N \leftarrow N \mid\mid 10^*$
$IV \leftarrow p_1(N \oplus IV_R \mid\mid IV_C)$
if $A \neq \varnothing$ **then**
    $A[1]A[2]\dots A[u] \leftarrow A$
    $A[u] \leftarrow A[u] \mid\mid 10^*$
    for $i = 1$ to u **do**
        $IV \leftarrow p_1(A[i] \oplus IV_R \mid\mid IV_C)$
    end for
end if
$C[1]C[2]\dots C[w] \leftarrow C$
$\ell \leftarrow |C[w]|$
$C[w] \leftarrow \lceil C[w-1] \rceil_{r-\ell} \mid\mid C[w]$
$C[w-1] \leftarrow \lfloor C[w-1] \rfloor_\ell$
$C[0] \leftarrow IV_R$
$V \leftarrow C[w] \mid\mid Expand(K) \oplus T$
$V \leftarrow p_1^{-1}(V)$
$M[w] \leftarrow \lfloor V_R \rfloor_\ell \oplus C[w-1]$
$V \leftarrow V \oplus M[w]10^* \mid\mid 0^C$
for $i = w-1$ to 1 **do**
    $V \leftarrow p_1^{-1}(V)$
    $M[i] \leftarrow C[i-1] \oplus V_R$
    $V \leftarrow C[i-1] \mid\mid V_C$
end for
$M \leftarrow M[1]M[2]\dots M[w]$
if $IV_c = V_c \oplus Expand(0^{c-1} \mid\mid 1)$ **then**
    return M
else
    return $\perp$
end if

8.3.2 HANUMAN-BS

Algorithm 3 $\mathcal{E}_K(N, A, M)$

Input: $K \in C^{\frac{1}{2}}$, $N \in C^{\frac{1}{2}}$, $A \in \{0,1\}^*$, $M \in \{0,1\}^*$
Output: $C \in \{0,1\}^*, T \in C^{\frac{1}{2}}$

$V \leftarrow Expand(0^r \mid\mid K \mid\mid N)$
$V \leftarrow AddSC(V)$
$V \leftarrow p_1(V)$
if $A \neq \varnothing$ **then**
  $A[1]A[2]\dots A[u] \leftarrow A$
  $A[u] \leftarrow A[u] \mid\mid 10^*$
  for $i = 1$ **to** $u - 1$ **do**
    $V \leftarrow p_4(A[i] \oplus V_R \mid\mid V_C)$
  end for
  $V \leftarrow p_1(A[u] \oplus V_R \mid\mid V_C)$
end if
$M[1]M[2]\dots M[w] \leftarrow M$
$\ell \leftarrow |M[w]|$
$M[w] \leftarrow M[w] \mid\mid 10^*$
for $i = 1$ **to** w **do**
  $C[i] \leftarrow M[i] \oplus V_R$
  $V \leftarrow p_1(C[i] \mid\mid V_C)$
end for
$C \leftarrow C[1]C[2]\dots C[w-1] \lfloor C[w] \rfloor_\ell$
$V \leftarrow Combine(V)$
$T \leftarrow \lfloor V_c \rfloor_{\frac{c}{2}} \oplus K$
return (C, T)

Algorithm 4 $\mathcal{D}_K(N, A, C, T)$

Input: $K \in C^{\frac{1}{2}}$, $N \in C^{\frac{1}{2}}$, $A \in \{0,1\}^*$, $C \in \{0,1\}^*, T \in C^{\frac{1}{2}}$
Output: $M \in \{0,1\}^*$ or $\perp$

$V \leftarrow Expand(0^r \mid\mid K \mid\mid N)$
$V \leftarrow AddSC(V)$
$V \leftarrow p_1(V)$
if $A \neq \varnothing$ **then**
  $A[1]A[2]\dots A[u] \leftarrow A$
  $A[u] \leftarrow A[u] \mid\mid 10^*$
  for $i = 1$ **to** $u - 1$ **do**
    $V \leftarrow p_4(A[i] \oplus V_R \mid\mid V_C)$
  end for
  $V \leftarrow p_1(A[u] \oplus V_R \mid\mid V_C)$
end if
$C[1]C[2]\dots C[w] \leftarrow C$
$\ell \leftarrow |C[w]|$
for $i = 1$ **to** $w - 1$ **do**
  $M[i] \leftarrow C[i] \oplus V_R$
  $V \leftarrow p_1(C[i] \mid\mid V_C)$
end for
$M[w] \leftarrow \lfloor V_R \rfloor_\ell \oplus C[w]$
$V \leftarrow p_1((M[w] \mid\mid 10^* \oplus V_R) \mid\mid V_C)$
$M \leftarrow M[1]M[2]\dots M[w-1]M[w]$
$V \leftarrow Combine(V)$
$T \leftarrow \lfloor V_c \rfloor_{\frac{c}{2}} \oplus K$
return $T = T' \ ?\ M : \perp$

8.3.3 GIBBON-BS

Algorithm 5 $\mathcal{E}_K(N, A, M)$

Input: $K \in C^{\frac{1}{2}}$, $N \in C^{\frac{1}{2}}$, $A \in \{0,1\}^*$, $M \in \{0,1\}^*$
Output: $C \in \{0,1\}^*, T \in C^{\frac{1}{2}}$

$V \leftarrow Expand(0^r \mid\mid K \mid\mid N)$
$V \leftarrow AddSC(V)$
$V \leftarrow p_1(V)$
$V \leftarrow V \oplus Expand(0^r \mid\mid K \mid\mid 0^{\frac{c}{2}})$
if $A \neq \varnothing$ **then**
    $V \leftarrow p_2(V)$
    $A[1]A[2]\ldots A[u] \leftarrow A$
    $A[u] \leftarrow A[u] \mid\mid 10^*$
    for $i = 1$ **to** $u - 1$ **do**
        $V \leftarrow p_2(A[i] \oplus V_R \mid\mid V_C)$
    end for
    $V \leftarrow A[u] \oplus V_R \mid\mid V_C$
end if
$M[1]M[2]\ldots M[w] \leftarrow M$
$\ell \leftarrow |M[w]|$
$M[w] \leftarrow M[w] \mid\mid 10^*$
$V \leftarrow p_3(V)$
for $i = 1$ **to** w **do**
    $C[i] \leftarrow M[i] \oplus V_R$
    $V \leftarrow p_3(C[i] \mid\mid V_C)$
end for
$V \leftarrow Combine(V)$
$V \leftarrow p_1(V_r \mid\mid (K \mid\mid 0^{\frac{c}{2}}) \oplus V_c)$
$C \leftarrow C[1]C[2]...C[w-1] \lfloor C[w] \rfloor_\ell$
$T \leftarrow \lfloor V_c \rfloor_{\frac{c}{2}} \oplus K$
return (C, T)

Algorithm 6 $\mathcal{D}_K(N, A, C, T)$

Input: $K \in C^{\frac{1}{2}}$, $N \in C^{\frac{1}{2}}$, $A \in \{0,1\}^*$, $C \in \{0,1\}^*, T \in C^{\frac{1}{2}}$
Output: $M \in \{0,1\}^*$ or $\perp$

$V \leftarrow Expand(0^r \mid\mid K \mid\mid N)$
$V \leftarrow AddSC(V)$
$V \leftarrow p_1(V)$
$V \leftarrow V \oplus Expand(0^r \mid\mid K \mid\mid 0^{\frac{c}{2}})$
if $A \neq \varnothing$ **then**
    $V \leftarrow p_2(V)$
    $A[1]A[2]\ldots A[u] \leftarrow A$
    $A[u] \leftarrow A[u] \mid\mid 10^*$
    for $i = 1$ **to** $u - 1$ **do**
        $V \leftarrow p_2(A[i] \oplus V_R \mid\mid V_C)$
    end for
    $V \leftarrow A[u] \oplus V_R \mid\mid V_C$
end if
$C[1]C[2]\ldots C[w] \leftarrow C$
$\ell \leftarrow |C[w]|$
$M[w] \leftarrow M[w] \mid\mid 10^*$
$V \leftarrow p_3(V)$
for $i = 1$ **to** w **do**
    $M[i] \leftarrow C[i] \oplus V_R$
    $V \leftarrow p_3(C[i] \mid\mid V_C)$
end for
$M[w] \leftarrow \lfloor V_R \rfloor_\ell \oplus C[w]$
$V \leftarrow p_3((M[w] \mid\mid 10^* \oplus V_R) \mid\mid V_C)$
$M \leftarrow M[1]M[2]\ldots M[w-1]M[w]$
$V \leftarrow Combine(V)$
$V \leftarrow p_1(V_r \mid\mid (K \mid\mid 0^{\frac{c}{2}}) \oplus V_c)$
$T \leftarrow \lfloor V_c \rfloor_{\frac{c}{2}} \oplus K$
return $T = T' \ ?\ M : \perp$

9 Benchmarking

This chapter contains the performance results obtained from the bit sliced implementations. The first section of the chapter contains information about the test environment, the second about the test setup, and the third will contain the test results. Chapter 10 presents an evaluation of the results.

9.1 Test Environment

All tests were run on a Lenovo T450 laptop with the specifications shown in table 9.1.

System manufacturer	Lenovo
System Model	20BV001BMD (Thinkpad T450)
Operating System	Windows 10 Pro 64-bit (10.0, Build 10586)
CPU	Intel Core i7-5600U @ 2.60 GHz (4 CPUs)
Memory	8192MB RAM

Table 9.1: Test machine specifications

9.2 Test Setup

All the tests were performed the following way:

- Each test was run on a cold booted machine. We allowed the machine to be idle for around 5 minutes after the cold boot, to ensure that all boot processes had finished.
- No other applications were run on the machine, while the tests ran. The only running applications were the processes that comes bundled with Windows 10 as default and starts upon boot.
- The messages, key, nonce and associated data used in the tests will always consist of only zeroes. Since this is a bit-sliced implementation, the speed is constant independent of the content of its input (only the input length matters). However, the results for the reference-implementations might have been affected by this. The correct functionality of the implementation has been tested with other test vectors that were not only zeroes.
- The benchmarking for the tests were done by reading the CPU cycle counter in the CPU (its TSC register) just before the encryption and just after the decryption. The performance we measure will be the median performance of the encryption and decryption operations individually.

- The compiler and corresponding flags used for all the tests are the same as those specified in chapter 6.
- All tests were run on a single core. Intel Turbo Boost was turned off as well, to have the CPU run at a consistent frequency.

9.3 Benchmark Results

In this section the results for APE-BS, HANUMAN-BS and GIBBON-BS will be listed. We will also list the speed-wise results we obtained for the reference implementations for APE, HANUMAN and GIBBON. We are aware that the reference implementations were most likely designed with a focus on readability over speed. Yet, we see two valid reasons for also measuring the performance of the reference implementations:

- It is hard to make any conclusions about the bit sliced performance-increases without having a bottom-line to compare the results to. They will serve as the bottom-line.
- If someone would want to reproduce, verify or improve our results in their own test environment, the reference implementations' performance will be needed.

The following subsections will contain the results of the bit-sliced and non-bit sliced implementations. The evaluation of the results will be in section 10.1.

9.3.1 APE & APE-BS

9.3.1.1 Small Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 40 bytes
- Encryption and decryption iterations: 100.000

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
APE-120	498612	409540	~6232	~5119
APE-80	126260	141180	~1578	~1764
APE120-BS	24340	24968	~304	~312
APE80-BS	15020	15572	~187	~194

Table 9.2: Performance results for APE-BS and APE in CPU cycles for small data sizes.

9.3.1.2 Medium Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 4.000 bytes
- Encryption and decryption iterations: 20.000

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
APE-120	19241920	11264328	~4762	~2788
APE-80	5105816	6469028	~1263	~1601
APE120-BS	619324	664680	~153	~164
APE80-BS	379296	421660	~93	~104

Table 9.3: Performance results for APE-BS and APE in CPU cycles for medium data sizes

9.3.1.3 Large Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 4.000.000 bytes
- Encryption and decryption iterations: 50

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
APE-120	1896129216	1099808624	~4739	~2749
APE-80	505403720	640877504	~1263	~1602
APE120-BS	303451488	325654260	~151	~162
APE80-BS	186731360	207815652	~93	~103

Table 9.4: Performance results for APE-BS and APE in CPU cycles for large data sizes

9.3.2 HANUMAN & HANUMAN-BS

9.3.2.1 Small Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 40 bytes
- Encryption and decryption iterations: 100.000

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
HANUMAN-120	450480	450760	~5631	~5634
HANUMAN-80	117840	118412	~1473	~1480
HANUMAN120-BS	18328	18388	~229	~229
HANUMAN80-BS	10988	10760	~137	~134

Table 9.5: Performance results for HANUMAN-BS and HANUMAN in CPU cycles for small data sizes.

9.3.2.2 Medium Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 4.000 bytes
- Encryption and decryption iterations: 20.000

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
HANUMAN-120	19243804	19191688	~4763	~4750
HANUMAN-80	5021292	5025128	~1242	~1243
HANUMAN120-BS	614012	616708	~152	~152
HANUMAN80-BS	373620	361040	~92	~89

Table 9.6: Performance results for HANUMAN-BS and HANUMAN in CPU cycles for medium data sizes

9.3.2.3 Large Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 4.000.000 bytes
- Encryption and decryption iterations: 50

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
HANUMAN-120	9527071268	9499707220	~4763	~4749
HANUMAN-80	2493059288	2494505872	~1246	~1247
HANUMAN120-BS	304801880	304919744	~152	~152
HANUMAN80-BS	186217180	179286452	~93	~89

Table 9.7: Performance results for HANUMAN-BS and HANUMAN in CPU cycles for large data sizes

9.3.3 GIBBON & GIBBON-BS

9.3.3.1 Small Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 40 bytes
- Encryption and decryption iterations: 100.000

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
GIBBON-120	276428	275932	~3455	~3449
GIBBON-80	72080	72592	~901	~907
GIBBON120-BS	21364	21432	~267	~267
GIBBON80-BS	12516	12488	~156	~156

Table 9.8: Performance results for GIBBON-BS and GIBBON in CPU cycles for small data sizes.

9.3.3.2 Medium Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 4.000 bytes
- Encryption and decryption iterations: 20.000

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
GIBBON-120	9787940	9789160	~2422	~2423
GIBBON-80	2554272	2544664	~632	~629
GIBBON120-BS	319640	322248	~79	~79
GIBBON80-BS	191744	192520	~47	~47

Table 9.9: Performance results for GIBBON-BS and GIBBON in CPU cycles for medium data sizes

9.3.3.3 Large Data-sizes

Test:

- AD-size: 40 byte
- Message-size: 4.000.000 bytes
- Encryption and decryption iterations: 50

Results:

Cycles (median)	Encryption	Decryption	Cycles/byte (enc)	Cycles/byte (dec)
GIBBON-120	4833373548	4837914236	~2416	~2418
GIBBON-80	1275264732	1273161580	~637	~636
GIBBON120-BS	153868480	153910192	~76	~76
GIBBON80-BS	93299376	92599540	~46	~46

Table 9.10: Performance results for GIBBON-BS and GIBBON in CPU cycles for large data sizes

9.3.4 PRIMATEs Permutations Isolated

Test:

- Encryption and decryption iterations: 500.000

Results:

80-bit	Bit sliced	Reference	bit sliced CPB	Reference CPB
p_1	3525	6163	~88	~1232
p_2	1759	3089	~44	~617
p_3	1760	3088	~44	~617
p_4	3515	6160	~88	~1232
p_1^{-1}	3995	7963	~100	~1592
p_2^{-1}	2001	3986	~50	~797
p_3^{-1}	2002	3984	~50	~796
p_4^{-1}	3984	7967	~100	~1593

Table 9.11: Performance results when only testing the permutations. The measurements are in CPU cycles per byte, and the values are the median values. Remember the bit sliced permutations performs 8 parallel permutations.

120-bit	Bit sliced	Reference	bit sliced CPB	Reference CPB
p_1	5965	23798	~149	~4759
p_2	2993	11867	~75	~2373
p_3	2974	11890	~74	~2378
p_4	5955	23780	~149	~4756
p_1^{-1}	6413	13708	~160	~2741
p_2^{-1}	3204	6857	~80	~1371
p_3^{-1}	3188	6848	~80	~1369
p_4^{-1}	6469	13708	~161	~2741

Table 9.12: Performance results when only testing the permutations. The measurements are in CPU cycles per byte, and the values are the median values. Remember the bit sliced permutations performs 8 parallel permutations.

10 Benchmark Results

10.1 Evalutation of Results

The benchmarks of the bit sliced implementations has shown that in most cases, we achieve a factor 13-14x increase in speed, when compared to the reference implementations for PRIMATEs-80. The improvements are a factor 31-32x for PRIMATEs-120. These improvements are the same for all the bit-sliced schemes, which makes sense as the majority of the processing time for all the schemes is spent inside the same permutation, which we have optimized.

Both HANUMAN-BS and APE-BS are able to encrypt medium- and large-sized data at a speed of about 151-152 cycles per byte. The fact that they use approximately the same amount of time is no surprise, as the permutations run in constant time in a bit sliced implementation, and their permutations use the same amount of rounds. They are less effective for small data-sizes, but so are the reference implementations. This is expected as more time is being spent on the other parts of the schemes, which deal with initializing the state and creating the tag. This creates some constant overhead to the performance.

It is interesting to see that already at the medium-sized data, we achieve almost the same speeds as the ones we achieve on large data. This is good, since the data will not have to be extraordinarily large for the schemes to be near their full potential (of course the performance will always continue to improve asymptotically towards the speed of the permutation itself, as more data is being processed).

In terms of APE, it is good to see that the inverse S-box – which had a larger instruction count than the normal S-box – does not make APE much less competitive compared to neither its non-bit sliced variants or HANUMAN (APE120-BS is about 6% slower at decrypting than HANUMAN120-BS. APE80-BS is about 15% slower than HANUMAN80-BS). Due to this, we believe the biggest drawback to APE is the large tag needed.

GIBBON-BS is, as expected, about twice as fast as HANUMAN-BS and APE-BS, since most of its permutations use half the number of rounds. Naturally, this is also reflected in the reference implementation. The bit sliced implementation encrypts or decrypts the data at speeds of about 46 clock cycles per byte for large data-sizes.

The speed of the permutations themselves are also interesting to observe. Just like with the schemes, the bit sliced implementations are 13-14x or 31-32x times faster, depending on the security level. However, it is not only the speed increase that is interesting here. It is interesting to observe that while the permutations p_2 and p_3 encrypts data at speeds of 44 cycles per byte, the GIBBON-BS scheme processed them at around 46 cycles per byte. This means that the GIBBON-BS scheme built on top on the permutation is quite effective, as it is only slightly slower than the permutation itself. It also means that if the scheme is

to be optimized further, it would have to be in the permutation that any big optimizations are found. The same effectiveness can be observed between the permutations and APE-BS and HANUMAN-BS.

Finally, we observed something odd in the reference results. The inverse permutations for PRIMATEs-120 are nearly double as effective as the normal permutations. This behavior can also be seen reflected in the results for the reference implementation of APE120, where decryption happens significantly faster. This oddity does not happen for the 80-bit PRIMATEs permutations.
We compiled the code with the GCC and MSVC compilers as well to observe, if this peculiarity would happen with other compilers than ICC, which it did not. In this case, both the normal and inverse permutations would run at speeds somewhat slower than that of the normal permutation, when it had been compiled with ICC. This means that the ICC compiler optimizes the inverse permutation of the reference implementations heavier than the normal permutations for PRIMATEs-120. We tested with different optimizations flags and parameters, and the inverse permutation kept performing faster. Since we did not want to alter the reference implementations, we decided to accept this peculiarity. However, we would like to point out that due to this, the reference implementations could likely be changed slightly in the forward permutations to gain speed equal to the inverse permutation. In this case, the bit sliced PRIMATEs-120 implementations would still be ~18x times faster.

We would like to point out that the bit sliced implementations would have been close to double the speed, had the AVX-512 instruction set been used in place of AVX2. AVX-512 has not yet been released, as was explained in chapter 6.

10.2 Comparison to other CAESAR Submissions

We chose to compare our results with the results of the following ciphers from the CAESAR competition:

- Ascon [17].
- NORX [18].

ASCON was chosen, since it (like PRIMATEs) is a sponge-based family of ciphers built on a round-based SPN that uses constant addition, substitution of elements, and linear diffusion. Finally, it has been designed with bit slicing in mind and thus has a bit sliced S-box and a state split into 5 registers, which is exactly what we have achieved in our bit sliced PRIMATEs implementation as well.
NORX was chosen since they have designed an AVX2 implementation of their cipher, just as we have. NORX is also a cipher, which the designers themselves state has been designed with performance on 64-bit CPUs in mind, so a comparison to another cipher designed for performance on high-end hardware would be interesting.

We did not expect to be close to the speeds of NORX or Ascon, since they have both been designed with SIMD-programming in mind, while PRIMATEs has been designed with a focus on a very low hardware-area and resource-footprint.

However, due to this difference in focus areas, we also deem it acceptable that they are likely faster. Nonetheless, for a bit sliced implementation a comparison to these two are interesting.

The fastest results which we were able to find for NORX, was those published in the paper for NORX [18]. In table 4.2 in that paper, a list of all the performance results obtained in software-implementations is provided. They were able to obtain a speed of 1.99 cycles per byte on data-sizes larger than 4096 bytes, and a speed of 2.20 cycles per byte on data-sizes of exactly 4096 bytes. In contrast, we were able to obtain speeds of ~46 cycles per byte on data-sizes larger than 4040 bytes with GIBBON80-BS.

The fact that NORX is faster is no surprise, as we are comparing the ciphers on the hardware for which NORX was designed. NORX is designed to only require AND, XOR, and shift-operations, which plays well on processor architectures capable of executing efficient SIMD-programming, such as Intel CPUs with the AVX2 instruction sets. Due to this, even though the bit sliced PRIMATEs permutation would perform far better on AVX-512, NORX would most likely get a similar increase in performance.
On the plus-side for PRIMATEs, since NORX is built around AND, XOR, and shift-operations only (there is e.g. no Galois field multiplications or S-box), its speed comes heavily from SIMD-possibilities. On very simple hardware-designs where there are only small registers, it is hard to say how well NORX will perform compared to PRIMATEs.

For Ascon, the fastest results we were able to find were those published on the homepage for Ascon [17]. They state that Ascon-128 is able of encrypting at 14.7 cycles per byte, while Ascon-128a encrypts at 10.5 cycles per byte. This is faster than PRIMATEs as well and quite interesting, since both are built on bit slicing with a state split in 5 registers.

It is no surprise that Ascon is faster, as Ascon was designed with bit slicing in mind. Since Ascon was designed with bit slicing in mind, it has a very simple bit sliced S-box, compared to the one we were able to find for PRIMATEs. Furthermore, its linear diffusion layer is more efficient for bit slicing. Their linear diffusion layer is implemented with only a few shifts and XORs, while PRIMATEs main resource-hog in the diffusion layer is the mix columns transformation with its Galois field multiplications. Their linear layer has the advantage there for bit slicing, when it comes to using long registers.
All that being said, we find it very interesting that PRIMATEs – which is a family of ciphers designed to be very lightweight and run in constrained environments – performs so well in comparison to other ciphers such as Ascon on 64-bit architectures. It is approximately 4-4.5 times slower, even though it was not designed for high-end hardware.

All in all we are satisfied with the results we achieved for PRIMATEs, when it comes to comparing results with other CAESAR submissions on high-end SIMD-capable hardware. It would have been great to see that our implementation would actually be faster, but we never expected this on high-end hardware, which NORX and Ascon were designed for more so than PRIMATEs. It would be interesting to see a comparison between these ciphers on the hardware-area required, and on the performance in lightweight environments, since this is where

PRIMATEs should be more efficient than them.

A final note in these comparisons is that the rate of PRIMATEs is 40 bits. In contrast, the rate of e.g. Ascon-128 is 64 bits, while the rate of Ascon-128a is 128 bits. It would be interesting to play around with PRIMATEs rate-size – granted that this could be secure to some degree – and do a comparison again with a possible larger rate. Perhaps it would be possible to convert some capacity to rate with PRIMATEs-120, which has a larger capacity. After all, our bit sliced permutation and schemes technically increases the state of PRIMATEs by a factor eight, so it would pay off well here, given it is possible.

11 Future Work

During our work with bit slicing the PRIMATEs permutation and designing schemes for it, we identified some areas which we believe could be the topic for future work. This chapter presents these areas.

11.1 Improved Bit Sliced S-box

It is possible that there are better implementations for the bit sliced S-box and its inverse. Especially for the inverse S-box, we believe that there likely exists a better implementation, since:

- It is much larger than the bit sliced normal S-box.
- We manually nudged the S-box circuits for better results.

Both of these are good indicators that at least the inverse S-box can be improved. However, even though the improvement to be had on the inverse S-box might be larger, we want to remind that it only affects APE-BS. Improvements to the normal S-box – even if smaller – will improve all three new schemes.
There does not seem to exist much theory currently on bit slicing S-boxes in general (based on what we could find), so future work research could be done in this field.

11.2 Improved Bit Sliced Mix Columns

The mix columns transformation is by far the most computationally heavy transformation in the bit sliced PRIMATEs permutation. No shortcuts were found to implement this transformation without performing the standard matrix multiplication in the Galois field. The only shortcut we found was to multiply the multiplication matrix together beforehand to lower the amount of times we had to apply it.

Schwabe and Käsper managed to optimize their bit sliced mix columns transformation to avoid performing matrix multiplication for AES. Despite their technique not being applicable to the matrix used for PRIMATEs, it is possible that other shortcuts exist to speed up the mix columns transformation, and future work could be put here.

11.3 Implementation of the PRIMATEs Permutation using AVX-512

As was explained in section 10.1, using AVX-512 should in theory nearly double the speed of the bit-sliced PRIMATEs permutation. It would be interesting to see the permutation being implemented using the AVX-512 instruction set and the actual performance-results it could achieve.

11.4 Bit Slicing Existing Schemes

We were able to heavily optimize the processing-speed of single messages, since we can process it in parallel and transpose the plaintext and ciphertext into and out of the registers very efficiently.
Unfortunately, it does not seem possible to implement a bit sliced scheme that works efficiently on single messages and also produces the same ciphertext as the existing schemes. However, it would be possible to use the bit sliced PRIMATEs permutation on 8 different messages in parallel and achieve the same ciphertext for each message as the current schemes. It would be interesting to see if an implementation that handles 8 messages at once would be feasible.

11.5 Design of New Schemes

It would be interesting to see the design of new non-bit sliced PRIMATEs schemes, which would also allow different parts of a message to be processed in parallel. This would allow a bit sliced and non-bit sliced implementation to achieve the same ciphertexts.

11.6 Bit Slice PRIMATEs for other Hardware-platforms

In this paper we designed a bit sliced implementation of the PRIMATEs permutation with the AVX2 instruction set available on AMD and Intel processors. ARM processors as an example uses other SIMD instruction sets such as NEON. It would be interesting to see PRIMATEs bit sliced on such processor architectures and how efficient it would be.

11.7 Comparison on low-end hardware between PRIMATEs and CAESAR submissions

We compared the achieved cycles per byte of our bit sliced PRIMATEs to other ciphers, which were designed with bit slicing techniques and SIMD-programming

in mind. As we discussed in chapter 10, this comparison may not be fair. It would be interesting to see a comparison between PRIMATEs and these ciphers on other parameters such as hardware-area required or efficiency on lightweight hardware. This is what PRIMATEs is designed for, and it may therefore outperform the other contestants on these areas.

12 Conclusion

In this paper we have proposed three new modes of operation for the 80- and 120-bit security levels of PRIMATEs. These three new modes are named APE-BS, HANUMAN-BS and GIBBON-BS. These new modes for PRIMATEs are similar to the functionality and specifications of the three existing modes, and they are built upon the same PRIMATEs permutation as well.
The new modes are designed with efficiency via parallelism in mind, and fully supports the parallel processing of single input messages. This is ideal for bit sliced implementations, where multiple encryption- or decryption-operations are performed in parallel on the input.
Since the new modes are designed with parallelism in mind and the old modes do not support this on single messages, the new modes produces different ciphertext than the existing modes of operations. The new modes also have a different tag-size than the existing modes, when it is necessary.

With our new modes, we have shown that it is possible to implement the underlying PRIMATEs permutations efficiently with bit slicing. We have achieved a factor 13~14x increase in speed with the bit sliced modes over the non-bit sliced for the 80-bit security level. The improvements for the 120-bit security level of PRIMATEs are at a factor 31~32x, although we believe that the 120-bit reference implementations can have their speed almost doubled easily.
Aside from the increases in speed between the old and new modes, the underlying bit sliced PRIMATEs permutation also processes data with the same increases in speed, when compared to the non-bit sliced permutation.
We have not changed the underlying PRIMATEs permutation in its design, so it still produces the same output from the same input as before.

With the new modes, we can encrypt data at about 46 cycles per byte for GIBBON80-BS (76 cycles per byte for GIBBON120-BS), while the reference implementations does so at 632 cycles per byte for GIBBON80 (and 2422 cycles per byte for GIBBON120). The approximate difference between the bit sliced HANUMAN and APE and the reference implementations are the same. HANUMAN80-BS can encrypt data at speeds of about 89 cycles per byte (152 for HANUMAN120-BS), while APE80-BS does so at 93 cycles per byte (151 for APE120-BS).

Since we have shown an implementation of the underlying PRIMATEs permutation that uses bit slicing, we have also shown that it is possible to make an implementation of the PRIMATEs permutation which is resistant to cache-timing attacks, as bit-sliced implementations run in constant time on the same input-sizes regardless of the content.

Finally, we have argued for how the PRIMATEs permutation can be further nearly doubled in speed by using the AVX-512 instruction set instead, which is due for the consumer-market in the second half of 2017. The current bit-sliced implementations uses the AVX2 instruction set.

Bibliography

[1] LearnCryptography.com. *Caesar Cipher*.
https://learncryptography.com/classical-encryption/caesar-cipher

[2] John Daemen and Vincent Rijmen. *AES Proposal: Rijndael*. September, 1999.
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.640

[3] T. Dierks, E. Rescorla. *The Transport Layer Security (TLS) Protocol*. August, 2008.
https://tools.ietf.org/html/rfc5246

[4] NIST. *Secure Hashing, Approved Algorithms*.
http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html

[5] NIST. *NIST's policy on hash functions*.
http://csrc.nist.gov/groups/ST/hash/policy.html

[6] ISO/IEC. *ISO/IEC 19772:2009*. February, 2009.
http://www.iso.org/iso/catalogue_detail.htm?csnumber=46345

[7] S. Kanno, M. Kanda. *AES Galois Counter Mode (GCM) Cipher Suites for TLS*. September, 2011.
https://tools.ietf.org/html/rfc6367

[8] Emilia Käsper and Peter Schwabe. *Faster and Timing-Attack Resistant AES-GCM*. Cryptology ePrint Archive, Report 2009/129.
http://eprint.iacr.org/2009/129

[9] Ko Stoffelen. *Optimizing S-box Implementations for Several Criteria using SAT Solvers*. Cryptology ePrint Archive, Report 2016/198.
http://eprint.iacr.org/2016/198

[10] Andreeva, et al. *PRIMATEs v1.02: Submission to the CAESAR Competition*. September 8, 2014.
http://primates.ae/wp-content/uploads/primatesv1.02.pdf

[11] Eli Biham. *A Fast New DES Implementation in Software*. Proceedings of the 4th International Workshop on Fast Software Encryption. FSE '97.
http://dl.acm.org/citation.cfm?id=647932.757246

[12] Matthew Kwan. *Bitslice DES*.
http://www.darkside.com.au/bitslice/

[13] The Mootely Fool. *Intel Corp. Confirms First 10-Nanometer Product on Track for 2017 Introduction*.
http://www.fool.com/investing/general/2016/02/16/intel-corp-confirms-first-10-nanometer-product-on.aspx

[14] Mark Buxton (Intel). *Haswell New Instruction Descriptions Now Available!*. June 13, 2011.
https://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/

[15] Intel Intrinsics Guide.
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[16] GCC optimization flags. Options That Control Optimization,
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[17] Dobraunig, et al. *Ascon: A Family of Authenticated Encryption Algorithms*.
http://ascon.iaik.tugraz.at/

[18] Jean-Philippe Aumasson, Philipp Jovanovic, Samuel Neves. *NORX*.
https://norx.io/

A PRIMATEs Tables

This appendix contains the S-box, shift rows indices, round-constants and mix columns matrix for the 120-bit PRIMATEs permutation.

A.1 S-box

x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S(x)	1	0	25	26	17	29	21	27	20	5	4	23	14	18	2	28
x	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
S(x)	15	8	6	3	13	7	24	16	30	9	31	10	22	12	11	19

Table A.1: The 5-bit S-box of PRIMATEs in decimal.

A.2 Round-Constants

Round	1	2	3	4	5	6	7	8	9	10	11	12
p_1	01	02	05	0a	15	0b	17	0e	1d	1b	16	0c
p_2	18	11	03	07	0f	1f						
p_3	1e	1c	19	13	06	0d						
p_4	18	11	03	07	0f	1f	1e	1c	13	19	06	0f

Table A.2: Round-constants used by the PRIMATEs permutations. Round-constants are in hexadecimal.

A.3 Mix Columns Matrix

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 15 & 9 & 9 & 15 & 2 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 18 & 2 & 2 & 18 \end{bmatrix}$$

Figure A.1: The multiplication matrix used for the mix columns operation. Multiplication is done over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

A.4 Shift Rows Indices

PRIMATEs-80					
Row	0	1	2	3	4
Left-shifts	0	1	2	4	7

Table A.3: The amount each row is left rotated during the *shift rows* operation for PRIMATEs-80.

PRIMATEs-120							
Row	0	1	2	3	4	5	6
Left-shifts	0	1	2	3	4	5	7

Table A.4: The amount each row is left rotated during the *shift rows* operation for PRIMATEs-120.

B Variant Mix Column Matrices

There is room for optimization in the Mix Columns transformation, since it applies the same matrix seven times. Instead of performing the computation:

$$State \times A1 \times A1 \times A1 \times A1 \times A1 \times A1 \times A1$$

with the same matrix, $A1$, one can for instance multiply the matrices together beforehand into $A7$, and then perform;

$$State \times A7$$

instead. This appendix contains all the different possible matrices that could be created, by multiplying matrix $A1$ together for PRIMATEs-80 and PRIMATEs-120. Section 7.2.4 contains the analysis for which matrix – or combination of matrices – that was the most efficient to use.

B.1 Mix Columns

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 15 & 9 & 9 & 15 & 2 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 18 & 2 & 2 & 18 \end{bmatrix}$$

Figure B.1: Matrix $A1$ suggested by the PRIMATEs team. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 15 & 9 & 9 & 15 & 2 \\ 2 & 5 & 28 & 29 & 27 & 23 & 11 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 18 & 2 & 2 & 18 \\ 18 & 8 & 19 & 3 & 11 \end{bmatrix}$$

Figure B.2: Matrix $A2$, found by $A2 = A1 \times A1$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 15 & 9 & 9 & 15 & 2 \\ 2 & 5 & 28 & 29 & 27 & 23 & 11 \\ 11 & 20 & 3 & 5 & 4 & 29 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 18 & 2 & 2 & 18 \\ 18 & 8 & 19 & 3 & 11 \\ 11 & 5 & 30 & 5 & 20 \end{bmatrix}$$

Figure B.3: Matrix $A3$, found by $A3 = A1 \times A1 \times A1$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 15 & 9 & 9 & 15 & 2 \\ 2 & 5 & 28 & 29 & 27 & 23 & 11 \\ 11 & 20 & 3 & 5 & 4 & 29 & 1 \\ 1 & 9 & 27 & 10 & 12 & 11 & 31 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 18 & 2 & 2 & 18 \\ 18 & 8 & 19 & 3 & 11 \\ 11 & 5 & 30 & 5 & 20 \\ 20 & 1 & 8 & 19 & 15 \end{bmatrix}$$

Figure B.4: Matrix $A4$, found by $A4 = A1 \times A1 \times A1 \times A1$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 15 & 9 & 9 & 15 & 2 \\ 2 & 5 & 28 & 29 & 27 & 23 & 11 \\ 11 & 20 & 3 & 5 & 4 & 29 & 1 \\ 1 & 9 & 27 & 10 & 12 & 11 & 31 \\ 31 & 26 & 29 & 7 & 22 & 24 & 16 \end{bmatrix} \quad \begin{bmatrix} 1 & 18 & 2 & 2 & 18 \\ 18 & 8 & 19 & 3 & 11 \\ 11 & 5 & 30 & 5 & 20 \\ 20 & 1 & 8 & 19 & 15 \\ 15 & 1 & 31 & 22 & 6 \end{bmatrix}$$

Figure B.5: Matrix $A5$, found by $A5 = A1 \times A1 \times A1 \times A1 \times A1$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 15 & 9 & 9 & 15 & 2 \\ 2 & 5 & 28 & 29 & 27 & 23 & 11 \\ 11 & 20 & 3 & 5 & 4 & 29 & 1 \\ 1 & 9 & 27 & 10 & 12 & 11 & 31 \\ 31 & 26 & 29 & 7 & 22 & 24 & 16 \\ 16 & 26 & 17 & 25 & 3 & 29 & 29 \end{bmatrix}$$

Figure B.6: Matrix $A6$ for PRIMATEs-120, found by $A6 = A1 \times A1 \times A1 \times A1 \times A1 \times A1$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$.

$$\begin{bmatrix} 1 & 2 & 15 & 9 & 9 & 15 & 2 \\ 2 & 5 & 28 & 29 & 27 & 23 & 11 \\ 11 & 20 & 3 & 5 & 4 & 29 & 1 \\ 1 & 9 & 27 & 10 & 12 & 11 & 31 \\ 31 & 26 & 29 & 7 & 22 & 24 & 16 \\ 16 & 26 & 17 & 25 & 3 & 29 & 29 \\ 29 & 15 & 16 & 31 & 23 & 9 & 2 \end{bmatrix}$$

Figure B.7: Matrix $A7$ for PRIMATEs-120, found by $A7 = A1 \times A1 \times A1 \times A1 \times A1 \times A1 \times A1$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$.

B.2 Inverse Mix Columns

$$\begin{bmatrix} 2 & 15 & 9 & 9 & 15 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 18 & 2 & 2 & 18 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure B.8: Matrix $A1^{-1}$ found by computing the inverse of $A1$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 11 & 23 & 27 & 29 & 28 & 5 & 2 \\ 2 & 15 & 9 & 9 & 15 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 11 & 3 & 19 & 8 & 18 \\ 18 & 2 & 2 & 18 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure B.9: Matrix $A2^{-1}$, found by $A2^{-1} = A1^{-1} \times A1^{-1}$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 1 & 29 & 4 & 5 & 3 & 20 & 11 \\ 11 & 23 & 27 & 29 & 28 & 5 & 2 \\ 2 & 15 & 9 & 9 & 15 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 20 & 5 & 30 & 5 & 11 \\ 11 & 3 & 19 & 8 & 18 \\ 18 & 2 & 2 & 18 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Figure B.10: Matrix $A3^{-1}$, found by $A3^{-1} = A1^{-1} \times A1^{-1} \times A1^{-1}$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 31 & 11 & 12 & 10 & 27 & 9 & 1 \\ 1 & 29 & 4 & 5 & 3 & 20 & 11 \\ 11 & 23 & 27 & 29 & 28 & 5 & 2 \\ 2 & 15 & 9 & 9 & 15 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 15 & 19 & 8 & 1 & 20 \\ 20 & 5 & 30 & 5 & 11 \\ 11 & 3 & 19 & 8 & 18 \\ 18 & 2 & 2 & 18 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure B.11: Matrix $A4^{-1}$, found by $A4^{-1} = A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1}$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 16 & 24 & 22 & 7 & 29 & 26 & 31 \\ 31 & 11 & 12 & 10 & 27 & 9 & 1 \\ 1 & 29 & 4 & 5 & 3 & 20 & 11 \\ 11 & 23 & 27 & 29 & 28 & 5 & 2 \\ 2 & 15 & 9 & 9 & 15 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 6 & 22 & 31 & 1 & 15 \\ 15 & 19 & 8 & 1 & 20 \\ 20 & 5 & 30 & 5 & 11 \\ 11 & 3 & 19 & 8 & 18 \\ 18 & 2 & 2 & 18 & 1 \end{bmatrix}$$

Figure B.12: Matrix $A5^{-1}$, found by $A5^{-1} = A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1}$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$. Left is for PRIMATEs-120. Right is for PRIMATEs-80.

$$\begin{bmatrix} 29 & 26 & 3 & 25 & 17 & 26 & 16 \\ 16 & 24 & 22 & 7 & 29 & 26 & 31 \\ 31 & 11 & 12 & 10 & 27 & 9 & 1 \\ 1 & 29 & 4 & 5 & 3 & 20 & 11 \\ 11 & 23 & 27 & 29 & 28 & 5 & 2 \\ 2 & 15 & 9 & 9 & 15 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure B.13: Matrix $A6^{-1}$ for PRIMATEs-120, found by $A6^{-1} = A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1}$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$.

$$\begin{bmatrix} 2 & 9 & 23 & 31 & 16 & 15 & 29 \\ 29 & 26 & 3 & 25 & 17 & 26 & 16 \\ 16 & 24 & 22 & 7 & 29 & 26 & 31 \\ 31 & 11 & 12 & 10 & 27 & 9 & 1 \\ 1 & 29 & 4 & 5 & 3 & 20 & 11 \\ 11 & 23 & 27 & 29 & 28 & 5 & 2 \\ 2 & 15 & 9 & 9 & 15 & 2 & 1 \end{bmatrix}$$

Figure B.14: Matrix $A7^{-1}$ for PRIMATEs-120, found by $A7^{-1} = A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1} \times A1^{-1}$ over $\mathbb{F}_{2^5} \cong \mathbb{F}_2[x]/(x^5 + x^2 + 1)$.

C S-box ANF Formulas

This appendix contains the ANF formulas, which we made from the PRIMATEs S-box. We used these to create a PRIMATEs S-box, which uses only the logic operations AND and XOR.

$$
\begin{aligned}
y0 &= 1 + x0 + x0x2 + x3 + x1x4 \\
y1 &= x0x1 + x2x3 + x4 + x0x4 + x2x4 \\
y2 &= x0x2 + x1x2 + x3 + x4 + x0x4 + x3x4 \\
y3 &= x1 + x0x2 + x1x2 + x1x3 + x2x3 + x4 \\
y4 &= x1 + x2 + x1x2 + x3 + x0x3 + x1x4 + x2x4
\end{aligned}
$$

Figure C.1: The ANF formulas used to create the PRIMATEs S-box for bit slicing

$$
\begin{aligned}
y0 &= 1 + x0 + x1 + x2 + x1x2 + x0x3 + x1x3 + x0x1x3 + x2x3 + \\
&\quad x0x2x3 + x1x4 + x0x1x4 + x0x2x4 + x1x2x4 + x3x4 + x2x3x4 \\
y1 &= x1 + x2 + x0x2 + x1x2 + x2x3 + x0x2x3 + x4 + x0x4 + \\
&\quad x0x2x4 + x0x3x4 + x1x3x4 \\
y2 &= x1 + x0x1 + x1x2 + x1x3 + x2x3 + x1x2x3 + x4 + x1x4 + \\
&\quad x0x1x4 + x2x4 + x0x2x4 + x0x3x4 \\
y3 &= x1 + x0x1 + x2 + x0x1x2 + x0x3 + x1x2x3 + x0x1x4 + \\
&\quad x0x2x4 + x1x2x4 + x0x3x4 + x1x3x4 \\
y4 &= x0x1 + x1x2 + x0x1x2 + x3 + x0x1x3 + x4 + x0x4 + \\
&\quad x1x4 + x0x1x4 + x2x4 + x0x2x4 + x1x2x4 + x3x4
\end{aligned}
$$

Figure C.2: The ANF formulas used to create the inverse PRIMATEs S-box for bit slicing