Software Evaluation for Second Round Candidates in NIST Lightweight Cryptography

Ryota Hira[1], Tomoaki Kitahara[1], Daiki Miyahara[1],
Yuko Hara-Azumi[2], Yang Li[1], and Kazuo Sakiyama[1]

[1] The University of Electro-Communications, Chofu, Tokyo 182-8585, Japan
[2] Tokyo Institute of Technology, Fuchu, Tokyo 183-8538, Japan

Abstract. Lightweight cryptography algorithms are increasing in value because they can enhance security under limited resources. National Institute of Standards and Technology is working on standardising lightweight authenticated encryption with associated data. Thirty-two candidates are included in the second round of the NIST selection process, and their specifications differ with respect to various points. Therefore, for each algorithm, the differences in specifications are expected to affect the algorithm's performance. This study aims to facilitate the selection and design of those algorithms according to the usage scenarios. For this purpose, we investigate and compare the 32 lightweight cryptography algorithm candidates using specifications and software implementations. The results indicate that latency and memory usage depend on parameters and nonlinear operations. In terms of memory usage, a difference exists in ROM usage, but not in the RAM usage from our experiments using ARM platform. We also discovered that the data size to be processed efficiently differs according to the padding scheme, mode of operation, and block size.

Keywords: Lightweight Cryptography · Authenticated Encryption with Associated Data · Block Cipher · Sponge Construction.

1 Introduction

With the conception of the Internet of Things (IoT), resource-constrained devices, such as sensor nodes, in-vehicle devices, and medical devices, can connect to the network. Cryptography must be applied to these devices to protect the data flowing over the Internet. However, conventional cryptographic standards are unsuitable for use in these devices because of their high cost. Therefore, a secure and implementable lightweight cryptography algorithm is required to protect data and privacy.

Accordingly, numerous institutions have focused on the research and development of lightweight cryptography [4, 12, 1, 6, 9]. National Institute of Standards

[2] A preliminary version of this paper was presented at the 38th Symposium on Cryptography and Information Security (SCIS 2021) [5]. In this study, we perform additional experiments on the finalists.

and Technology (NIST) has been working on standardising lightweight cryptography algorithms [10]. The standardisation process of NIST was initiated in July 2015, and lightweight cryptography algorithms were solicited in August 2018. In April 2019, 56 algorithms were published as Round-1 candidates. Furthermore, in August 2019, 32 algorithms were accepted as Round-2 candidates; 24 candidates were excluded because of their security vulnerability [14]. In March 2021, 10 finalists were selected from the 32 Round-2 candidates [13]; the finalists are currently being evaluated.

This study aims to implement lightweight cryptography algorithms in software for resource-limited embedded systems, and to facilitate the selection and design of cryptography algorithms according to the usage scenarios. For this purposes, we clarify the influence of cryptographic specifications on software implementation. We conduct a survey based on the specifications and software implementation performance of NIST Round-2 and finalist candidates, and compare the results. The contributions of this work are: (1) We discovered that latency varies depending on cryptographic parameter: nonce and block size. (2) In terms of memory usage, we explored the details of ROM usage in nonlinear operations. (3) We varified the size of the data to be processed efficiently varies depending on the block size, padding scheme, and mode of operation.

The remainder of this paper is organised as follows. Section 2 summarises the lightweight cryptography and authenticated encryption with associated data (AEAD). Section 3 describes the survey items and results of the study based on the specifications. Section 4 describes the investigation methodology and results of software implementation performance. Then, discusses the results of the survey on the Round-2 candidates. In Sect. 5, we perform additional experiments on the finalist candidates, and discuss the obtained results. Section 6 concludes the paper.

2 Prelimiaries

2.1 Lightweight Cryptography Algorithm

A lightweight cryptography algorithm is a class of cryptography algorithms that is used in environments with limited computational resources, memory size, and power. CRYPTREC recommends the lightweight cryptographic primitives, i.e., block cipher, stream cipher, hush function, as well as message authentication code (MAC) and authenticated encryption. The circuit size, power consumption, and latency in hardware implementations and memory size in software implementations are evaluated typical performance indicators of lightweight cryptography algorithms [3]. The lightweight cryptography algorithm, which is expected to be used in small devices, such as home appliances, medical equipment, and robots, that are connected to the Internet.

[2] A preliminary version of this paper was presented at the 38th Symposium on Cryptography and Information Security (SCIS 2021) [5]. In this study, we perform additional experiments on the finalists.

2.2 Authenticated Encryption with Associated Data

AEAD ciphers simultaneously provide confidentiality through encryption and integrity through message authentication [2]. The encryption takes the plaintext, associated data, nonce, and key as the input, and outputs the ciphertext and authentication tag. Associated data are authenticated with plaintext, and are not encrypted for integrity. Nonce is a disposable random number that is used to enhance the security of encryption under the same key. The decryption process takes the ciphertext, associated data, nonce, key, and authentication tag as the input, and outputs the plaintext if the authentication is successful. If the authentication fails, the plaintext is not output.

There are two main types of AEAD modes of operation selected by the NIST: block cipher and sponge modes [13]. The components used in the state transformation process differ for these two modes. The former uses a block cipher, and the latter uses a permutation; however, both modes use the XOR operation and padding functions.

1. Block Cipher-based Candidates
   The block cipher-based type of AEAD is realised through the block cipher mode of operation that repeatedly uses block ciphers. This type of AEAD requires a key schedule because the block cipher generally uses round keys.
2. Sponge-based Candates
   The sponge-based type of AEAD is realised via a sponge mode of operation that repeatedly uses permutations. This type of AEAD uses the permutation as a key, and does not require a key schedule to generate the round key. However, it is expedient to use a large permutation to increase security [15].

3 Specification-level Survey

Based on the specifications of each candidate submitted to the NIST and sample code of each algorithm in C language, we investigated the AEAD modes, recommended parameters, and operations of building blocks (block cipher or permutation) in cryptographic processing. In this study, we investigated 32 candidates: ACE, ASCON, COMET, DryGASCON, Elephant, ESTATE, ForkAE, GIFT-COFB, Gimli, Grain-128AEAD, HyENA, ISAP, KNOT, LOTUS-AEAD and LOCUS-AEAD, mixFeed, ORANGE, Oribatida, PHOTON-Beetle, Pyjamask, Romulus, SAEAES, Saturnin, SKINNY-AEAD, SPARKLE, SPIX, SpoC, Spook, Subterranean2.0, SUNDAE-GIFT, TinyJambu, WAGE, and Xoodyak. Some candidates have multiple members with different parameters and operations. We investigated 89 members from the 32 candidates. The submitter of each algorithm represents a representative member of the candidates, having multiple members. We refer to these representative members as primary members.

We summarise the results of the survey in table 6 as an appendix. The primary member of each candidate is marked with an asterisk. In the table entry, AEAD type 'B' refers to a block cipher-based candidate, and AEAD type 'S' to a sponge-based candidate. Because SAEAES has some members whose block size differs between the plaintext and associated data, the block size for the associated data is described in parentheses in table 6.

3.1 Types of AEAD

As mentioned in Sect. 2.2, there are two main types of AEAD: block cipher- and sponge-based. Different types of AEAD require different building blocks and parameters, which might change the performance.

We discovered 14 block cipher-based candidates with 51 members and 16 sponge-based candidates with 34 members. The remaining two candidates were classified as others: Grain-128AEAD and Elephant. The former is based on a stream cipher, and the latter is based on permutation with a non-sponge structure.

Some candidates had other characteristics. Spook is a sponge-based candidate that uses permutation Shadow-512 for state update; however, it also uses block cipher Clyde-128 for the state initialisation and finalisation of the AEAD. TinyJambu is classified as a block cipher-based candidate; however, it uses a keyed NFSR for the state transformation process, which is unique compared to other general block cipher operations.

3.2 Parameters

AEAD has five parameters: key, block (rate), state, tag, and nonce sizes.

1. Keys
   The key size is crucial to the security of a cipher. If the key size is small, the total number of keys decreases, and a brute-force attack becomes possible. Therefore, it is necessary to select a key size that is sufficiently resilient to brute-force attacks. The NIST submission requirements need the key size to be 128 bits or more.
2. Block size
   The block size affects the security and processing time. This value influences the amount of data processed securely when encryption is performed under the same key. The block size also refers to the number of bits of plaintext and ciphertext generated per block cipher or permutation. Therefore, it can increase or decrease the processing time.
3. State size
   The state size indicates the size of the block cipher or permutation used in the cryptographic process. This parameter determines the size of temporary variables during the encryption or decryption process, and thus, increases or decreases the RAM usage to store the intermediate state. The state size also affects the processing time because it is related to the amount of computation required to update the state.
4. Tag size
   The tag size determines the size of the authentication tag. This size affects the processing time when generating the authentication tag. According to the NIST submission requirements, the tag size should be 64 bits or more.

5. Nonce size
   The nonce size determines the size of random numbers used in the encryption. The time required to generate and supply random numbers from outside the cipher varies depending on the size of the random numbers used for the encryption. Additionally, the time required to process random numbers in the cipher differs, thus influencing the processing time. Therefore, this parameter affects the execution time. According to the NIST submission requirements, the nonce size needs to be 96 bits or more.

Table 1 and 2 present the mean, median, and mode of the parameters of the primary member of each candidate.

Table 1: Parameters of block cipher-based AEAD

	Block	Key	State	Tag	Nonce
Mean	121.1	137.1	146.3	128.0	118.1
Median	128.0	128.0	128.0	128.0	124.0
Mode	128	128	128	128	128

Table 2: Parameters of sponge-based AEAD

	Block	Key	State	Tag	Nonce
Mean	127.0	148.0	325.3	128.0	137.1
Median	128.0	128.0	289.5	128.0	128.0
Mode	128	128	256	128	128

Table 1 and 2 indicate that the state size differs between the block cipher- and sponge-based candicates. This difference is probably due to the use of a large permutation in sponge-based candidates to increase security, as described in Section 2.2. No differences exist in the other parameters depending on the types of AEAD.

3.3 Operations

Two types of state transformation operations are used in cryptography algorithms: linear and nonlinear. Nonlinear operations are necessary because linear cryptography algorithms are insecure against cryptanalysis. However, nonlinear operations are generally more expensive to implement than linear operations. Therefore, small nonlinear operations, which are less expensive and have weak nonlinearity, are commonly used. Generally, nonlinearity is spread over the entire state by applying linear operations after applying these small nonlinear operations. In this study, we investigate nonlinear operations that have high implementation costs.

There are three main types of nonlinear operations: modular operation, S-box, and Boolean function. They differ based on the optimal implementation environment and strength of nonlinearity [11].

S-boxes can be constructed in two ways; one method stores a lookup table in memory, which is a table with corresponding input and output values, and the other method calculates a programme using the Boolean function. Therefore, the amount of memory usage and execution time differ depending on the configuration method, even if the same S-box is used.

For the primary member of each candidate, 1 candidate used the modular operation, 13 candidates used the S-box, and 18 candidates used the Boolean function for nonlinearity. For the candidates using S-boxes, the algebraic number of bits in the S-box is listed in table 6. The candidates using S-boxes, which were calculated with the Boolean function, were counted as candidates using the Boolean function, and the algebraic number of bits of the S-boxes is presented in parentheses in table 6.

Table 6 demonstrates that the candidates, constructing S-boxes using lookup tables, are mostly used in block cipher-based AEAD, and that they use 4-, 7-, and 8-bits S-boxes. For the sponge-based AEAD, many candidates use the Boolean function or S-boxes calculated using the Boolean function.

4 Investigation of Software Implementation Performance

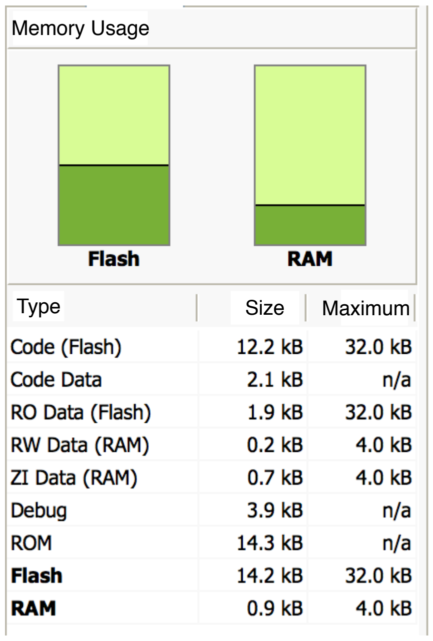
4.1 Measurement Method

We conducted experiments to evaluate the software performance of each candidate. We implemented the sample codes in C language for each candidate submitted to the NIST, and investigated their latency and memory usage. The programmes used in this measurement are reference implementations of each algorithm.

We used the test vectors prepared by the NIST. The test vectors comprised 33×33 or 1089 combinations of plaintext and associated data by default; each data size ranged from 0 to 32 bytes, varying 1 byte at a time. The key and nonce sizes of each test vector are fixed. In this measurement, we used 289 of the prepared test vectors to reduce the time required for the measurement. The test vectors comprised 17×17 or 289 combinations of plaintext and associated data, each data size ranged from 0 to 32 bytes, varying 2 bytes at a time.

The execution and development environment is as follows.

- Execution environment: Mbed LPC1114FN28 [8]
  - CPU: 32 bit ARM cortex-M0 core
  - Clock frequency: 48 MHz
  - Flash memory: 32 kB
  - RAM: 4 kB
- IDE: Mbed Online Compiler
  - Easy to prepare the development environment
  - Presents the memory usage when compiling



Type	Size	Maximum
Code (Flash)	12.2 kB	32.0 kB
Code Data	2.1 kB	n/a
RO Data (Flash)	1.9 kB	32.0 kB
RW Data (RAM)	0.2 kB	4.0 kB
ZI Data (RAM)	0.7 kB	4.0 kB
Debug	3.9 kB	n/a
ROM	14.3 kB	n/a
Flash	14.2 kB	32.0 kB
RAM	0.9 kB	4.0 kB

Fig. 1: Analysis results of memory usage acquired from Mbed online compiler (example screenshot, partially rewritten in English)

We measured the execution time with two timers in the programme: one for encryption, and the other for decryption. The values obtained are the sum of the latencies for encryption or decryption for the 289 test vectors.

The memory usage was obtained using the Mbed online compiler [7]. Figure 1 presents the results of the Mbed online compiler, which details the memory usage when employing the compiling programme. Figure 1 also shows the analysis results of the comparison programme (mentioned later). Here, the flash memory is divided into code and read-only (RO) data. The code stores the programme code, and the RO data stores the RO constant data. The RAM stores data that may be rewritten during the programme operation. These data are divided into read-write (RW) and zero-initialised (ZI) data.

4.2 Results

We summarise the results of the survey in table 7 as an appendix. The first member of each candidate is marked with an asterisk.We created another identical programme that performs everything expect the cryptographic processing unit, to compare the amount of memory usage for the cryptographic processing unit of the programme. We have included this result to the second row of table 7 as a programme for comparison.

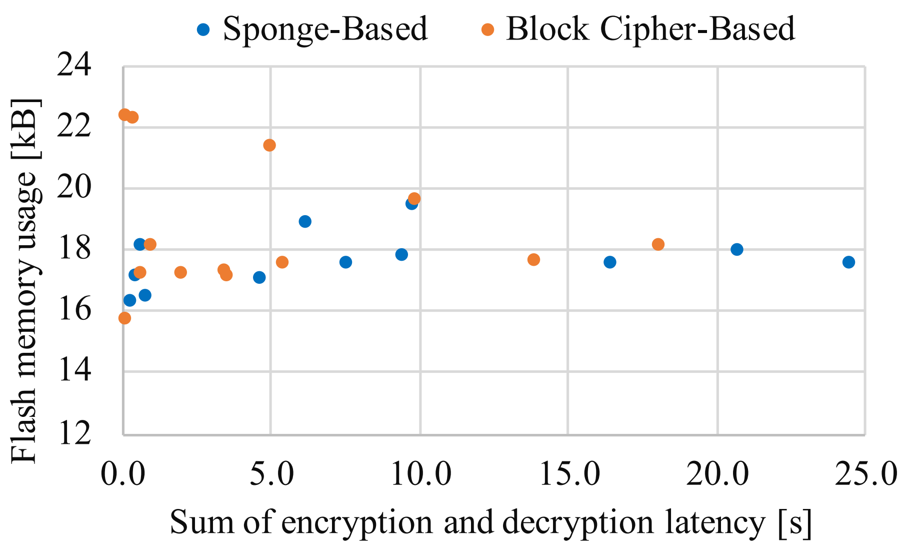


Fig. 2: Relationship between latency and flash memory usage

We only discussed the amount of flash memory usage because there was no difference in the RAM usage for each candidate.Owing to the difference in the number of members for each candidate, the mean and median of the implementation performance for all members were inconsistent and biased. Therefore, we only discussed the primary member.

Figure 2 illustrates the software implementation performance, and table 3 lists the mean and median of the results. Figure 2 presents an excerpt of the measurement results, excluding data with extremely high latency and memory usage (candidates with flash memory usage exceeding 24 kB and latency exceeding 25 s).

3 shows that the block cipher-based AEAD tends to have lower latency and higher flash memory usage than the sponge-based AEAD. However, figure 2 shows that the type of AEAD does not have a decisive influence on the software implementation performance, because some candidates have no difference in performance, although they are different types of AEAD.

Table 3: Latency and flash memory usage

		Sum of latency	Flash memory usage
Block cipher base	Mean	5.0	18.7
	Median	3.5	17.9
Sponge base	Mean	7.9	18.4
	Median	6.2	17.5

4.3 Analysis of Measurement Results

We performed an analysis based on the results. The results of the parameters, nonlinear operations, and performance, used in the analysis, are those listed in table 6 and 7.

Table 4 presents the obtained correlation coefficients of all 89 members for implementation performance and each parameter, and that for implementation performance and cost of the nonlinear operation. The cost of the nonlinear operations differs from that of the S-boxes, modular operation, and Boolean function. In the case of the S-boxes, the cost of the nonlinear operations is the algebraic number of bits per output value stored in the lookup table. For the modular operation and Boolean function, the cost of the nonlinear operation is set to zero, because no lookup table is required.

In each section, we describe the trends that emerged in the performance of all candidates and characteristics of a specific candidates.

Table 4: Correlation coefficients between the parameters and size of nonlinear operations and the implementation performance

	Latency	Flash memory Usage	Code Data	RO Data
Block size	0.143	-0.236	-0.107	-0.330
Key size	-0.127	0.061	-0.033	0.210
State size	0.087	-0.156	-0.022	-0.316
Tag size	-0.162	-0.052	0.018	-0.162
Nonce size	-0.081	-0.075	-0.072	-0.033
Non-liner Operation	0.006	0.359	0.130	0.577

Parameters on Implementation Performance Table 4 shows that no significant correlation exists between the parameters and implementation performance. However, there is a weak negative correlation between the RO data and block and state sizes. Therefore, as the block or state sizes increase, the amount of RO data usage decreases. The sponge-based AEAD tends to have a larger state size, and uses less flash memory than the block cipher-based AEAD. Therefore, the relationship between the state size and RO data might be due to the characteristics of the AEAD types. According to Section 3.2, the block size of the primary members does not depend on the type of AEAD. However, when we calculated the average block size of 89 members, the block cipher- and sponge-based AEAD were 105.4 and 127.0, respectively. This suggests that the relationship between the block size and RO data exists because of the types of AEAD.

Then, we discuss the performance of members with different parameters within the same candidate. ASCON, PHOTON-Beetle, and SAEAES have several members with different block sizes, and exhibit no significant differences in other parameters or operations. For example, based on table 5, photonbeetleaead128rate12v1 and photonbeetleaead128rate32v1 have block sizes of 128 and 32 bits, respectively. Their encryption and decryption latencies are approximately 12 and 30 s, respectively. Similarly, saeaes128a120t128v1 and saeaes128a64t128v1 have different block sizes of 120 and 64 bits for associated data, and the former has lower latency. For ascon128av12 and ascon128v12, the latency of the former is lower with a larger block size, although the difference between their number of rounds of permutation is two. Therefore, members with a larger block size tend to have lower latencies for different members of the same candidate.

Table 5: Correlation coefficients between the parameters and size of nonlinear operations and the implementation performance

Member	Block size	Encryption time(seconds)
ascon128av12	128	0.153
ascon128v12	64	0.183
* photonbeetleaead128rate128v1	128	12.251
photonbeetleaead128rate32v1	32	29.655
saeaes128a120t128v1	64(120)	0.099
saeaes128a64t128v1	64(64)	0.111

SUNDAE-GIFT has four members with different nonce sizes. The latencies for nonce sizes of 0, 64, 96, and 128 bits are 2.41, 2.65, 2.76, and 2.87 s, respectively. Thus, these results indicate that, when the nonce size is larger, the latency is higher. Therefore, the size of the nonce size affects the latency.

Operations on Implementation Performance For reference, when S-boxes are implemented using a lookup table, a 4-bit S-box requires 64 bits (0.0078 kB) of memory usage, and an 8-bit S-box requires 2048 bits (0.25 kB).

Table 4 shows that the correlation coefficient between the cost of nonlinear operations and flash memory usage is 0.359, indicating a weak positive correlation. Additionally, the correlation coefficient with the RO data is 0.577, indicating a positive correlation. Therefore, the cost of the nonlinear operations affects the flash memory usage, especially RO data usage.

COMET and ESTATE have several members with different operations, and demonstrate no significant differences in the other parameters. COMET128AESV1 and COMET128CHAMV1 are based on the block ciphers AES and CHAM, respectively. The nonlinear operations used in each cipher are 8-bit S-boxes for AES and modular addition for CHAM. The implementation performance of the two members differs in terms of the RO data and latency. Moreover, estatetweaes128v1, using 8-bit S-boxes for TweAES, and estatetwegift128v1, using 4-bit S-boxes for TweGIFT, differ with respect to the RO data and latency. Therefore, the operations used in the algorithm affect the memory usage and latency.

All members of SAEAES and comet128aesv1 have more RO data than the other candidates. Their sample programmes include four arrays of 256 4-byte data for lookup tables of size 4 kB. These data are supposed to be used to speed up the cryptographic process; in fact, the latency of these candidates is low. Therefore, the latency can be reduced by storing large data in the memory beforehand.

Other Factors on Implementation Performance There was no difference in RW, ZI, and their sum for all 89 members of the 32 candidates. However, elephant160v1 and elephant176v1 as well as all members of mixFeed had more RW data than the other candidates. We examined the sample codes of the two candidates, and discovered that the array data stored in the lookup table did not have a const modifier to treat the constant as RO data in the C language. This might have caused the lookup table to be stored in the RAM as RW data. After adding the const modifier to their data, they were stored in flash memory as RO data.

Elephant160v1 and elephant176v1, isapk128v20, and Oribatida had higher latency than the other candidates. In the Elephant specification, elephant160v1 and elephant176v1 were hardware-oriented members with more rounds in the substitution function at 80 and 90. Therefore, they are unsuitable for the software, and their latency is considerably high. Elephant200v1—a software member—had 18 rounds of substitution functions, and the latency was approximately 10 s. The reason for the high latency of isapk128v20 and Oribatida has not been discovered.

5 Experiment for use in specific scenarios

5.1 Measurement Method

We conducted experiment to investigate the relationship between the data size and latency for 28 members of the 10 finalists. From this experiment, we can discover the data size to be processed efficiently and can use each candidates in specific scenarios. We used the reference C implementation code of the 10 finalists for the experiments. The execution and development environments are the same as those described in Section 4. We implemented all members of each candidate in the software, and measured the latency for different data sizes. In this study, the data size was the sum of the plaintext and AD lengths. For example, when processing 8 bytes of the plaintext and 8 bytes of the AD, the data size is 16 bytes.

The test vectors comprised $33 \times 33 \, \text{or} \, 1089$ combinations of plaintext and AD; each data size ranged from 0 to 32 bytes, varying 1 byte at a time. However, for some of the candidates (romulus-m and xoodyak), we used more test vectors ($100 \times 100 \, \text{or} \, 10000$) to investigate them in detail.

Unlike the experiments conducted in Section 4, we measured the latency for the encryption/decryption of each test vector separately. Therefore, for one candidate (or one member), we obtain the latency for 1089 (or 10,000) vectors. We compared the latency for each data size to determine what could be processed efficiently.

5.2 Results

We summarised the results in a heat map as an appendix in Section C. All data used to create the figures are latency taken for encryption. The vertical axis represents the plaintext length, and the horizontal axis represents the AD length. For each candidate, we created two diagrams based on the experimental results. One diagram depicts the heat mapping of raw-data, where white indicates the minimum latency and black the maximum latency (hereafter, referred to as 'raw-data'). The other illustrates the heat mapping of the obtained latency divided by the data size in bytes (hereafter referred to as 'divided-data'). For example, the latency obtained by processing 8 bytes of plaintext and 8 bytes of AD is divided by data size $8 + 8 \, \text{or} \, 16$. The divided-data express the latency per 1 byte, and allow the data size to be compared, which can be processed efficiently. In this figure, the 10th percentile is coloured white, and the 90th percentile is coloured black to enhance readability.

5.3 Analysis of Data Size and Latency

In this section, we analyse the results of the experiments described in Section 6. The analysis is based on the raw- and divided-data. From the previous analysis, we consider the relationship between the specification of the cipher and the latency. The latter analysis provides the data size, which can be processed efficiently.

Analysis with Raw-data From the results for each candidate, the latency increased as the processed data size increased. Furthermore, the latency increased step-by-step after a certain number of bytes for the AD and plaintext lengths of the processed data. These numbers of bytes were based on the block size of the AEAD, and the latency increased as the data size exceeded integer multiples of the block size. The candidates that used the stream cipher as building blocks (Grain-128AEAD) had the same characteristics if we consider the block size as 1 bit or 1 byte. All the candidates showed a stepwise increase in latency based on the block size. However, the latency increased in different ways depending on the padding scheme and mode of operation. We describe the effect of the padding scheme and mode of operation in the following.

Effect of Padding Scheme In the AEAD mode, a single encryption block obtains a ciphertext of bit length equal to that of plaintext, equal to block size r. Therefore, when encrypting a plaintext of length r, the ciphertext should be obtained with only one encryption block. However, depending on the padding scheme, the number of encryption blocks may increase. Next, we summarise the candidates wherein the number of encryption blocks varies depending on the padding scheme.

The data sizes, where the characteristics of the padding scheme appear, are 0 bytes and an integer multiple of r bytes. First, we describe the process when the data size is 0 bytes. In such a case, some candidates perform padding, and process the data as a single block. However, some candidates do not process the data, and skip the cryptographic process. We classify the candidates as follows. Here, Proc_AD and Proc_PT refer to the series of operations for AD and PT, respectively.

1. Does not perform Proc_PT and Proc_AD
   Grain128-AEAD, PHOTON-Beetle, Sparkle, TinyJambu
2. Performs Proc_PT and does not perform Proc_AD
   Ascon
3. Does not perform Proc_PT and performs Proc_AD
   Elephant, GIFT-COFB, ISAP, Romulus
4. Performs Proc_PT and Proc_AD
   Xoodyak

In terms of performance, the candidate that does not process when the data size is zero has low latency, and is more efficient. Performing cryptographic operations only on the padded data is inefficient.

Subsequently, we describe the process when the data size is approximately an integer multiple of r bytes. Padding is applied to the last block of data when it is less than the block size, making the data size equal to the block size. The input, which is an integer multiple of the block size, is divided into multiple blocks of r bytes, and encrypted.

The finalist candidates use two major padding schemes. One pads only number of bits less than the block size. If the data size x is $(n-1)r < x < nr$, then the data size becomes nr (for $n = 1, 2, 3, ...$) after the padding process. Therefore, the input data of size x can be processed using n blocks. We refer to this method as padding scheme 1 in this study.

In the other padding scheme, when the data size is equal to the block size, i.e. nr bytes, an additional block of padded data is added. Therefore, if the data size is nr bytes, the number of blocks becomes $(n+1)$. We refer to this method as padding scheme 2 in this paper.

The padding schemes for each candidate are summarised below.

1. Padding scheme 1
   GIFT-COFB, PHOTON-Beetle, Romulus, Sparkle, TinyJambu, Xoodyak
2. Padding scheme 2
   ASCON, Elephant, ISAP

In Elephant, the nonce is padded to AD, which changes the data size that can be processed in one block. Because Grain128-aead uses a stream cipher as a building block, we cannot determine the effect of padding.

Effect of Mode of Operation The NIST candidates use various cryptographic modes of operation. The different modes of operation affect the manner latency changes and memory usage. In this section, we discuss the latency.

First, We describe the candidates whose latency changes in steps of 2 bytes, depending on the mode of operation. The candidates are Elephant and ISAP; their modes of operation are Enc-then-Mac. The Enc-then-Mac mode encrypts the plaintext, and generates an authentication tag from the ciphertext. Both candidates encrypt the plaintext, and generate tags from the AD and ciphertext. At this time, padding scheme 1 is used for the plaintext, and padding scheme 2 is used for the AD and ciphertext. Therefore, depending on the data size, the number of blocks in the plaintext and ciphertext blocks differ, resulting in a latency change of 2 bytes. Specifically, from $(n-1)r+1$ to $nr-1$ bytes, both the plaintext and ciphertext blocks are processed with n blocks. With nr bytes, the plaintext block is processed with n encryption blocks; however, the ciphertext block requires $n+1$ blocks. Therefore, the latency changes once. Furthermore, when processing $nr+1$ bytes of input data, boththe plaintext and ciphertext blocks require $n+1$ blocks of processing. This changes the latency again. As described above, Elephant and ISAP have a latency change of 2 bytes.

Subsequently, we introduce candidates for which the time taken to process the PT and AD are different. These candidates have different latency increments when the plaintext and AD blocks are increased. For Elephant and ISAP, the latency increment is larger when the number of plaintext blocks is increased compared to when the number of AD blocks is increased. This is because, when the number of plaintext blocks increases, the number of ciphertext blocks required to generate the tag also increases.

TinyJambu has a similar feature. TinyJambu increases the latency when the number of plaintext blocks increases, compared to when the number of AD blocks increases. This is because the number of rounds of the function used to update the state differs. TinyJambu uses 1024 and 640 rounds of functions to update the state of the PT blocks and AD blocks, respectively. Therefore, the latency increases, especially when the number of PT blocks increases.

Moreover, when ASCON increases the number of AD blocks, the latency increases more than when it increases the number of PT blocks. However, we did not learn the cause of this from the specification.

Finally, we introduce a characteristic candidate. It is a member of Romulus, romulus-m. This candidate differs significantly from the other finalist in the manner the latency changes. Generally, the latency of the other candidates increases as the AD length and plaintext length exceed an integer multiple of the block size. The latency is related to the data size and varies regularly, and the diagram is grid-like. However, the latency of romulus-m varies irregularly, and the diagram shows a shifted grid. For the plaintext length, the latency changes every 16 bytes, which is the block size. However, for the AD length, there was no clear relationship between the block length and latency. The reason for this is not clear from the specification.

Analysis with Divided-data The figures of divided-data shows that the latency per byte decreases as the data size increases. This is due to the initialisation and finalisation of the state, which is always performed regardless of the data size. The processing time for the AD and PT varies depending on the data size. However, the state initialisation and finalisation take almost constant time, regardless of the data size. Therefore, the latency per byte decreases as the data size (divisor) increases.

Additionally, the diagram of the divided-data is in the form of a grid, similar to the raw-data. Within the same grid (block), when the data size is larger, the latency per byte is smaller. This is because, when the number of blocks is the same, the latency is almost constant. The latency per byte changes only depending on the data length, which is the divisor. Therefore, when the data size is larger, processing within the same block is more efficient.

6 Conclusion

In this study, we surveyed and compared second-round candidates from the NIST lightweight cryptography project. We also conducted additional experiments on the finalist candidates to investigate the relationship between data size and latency.

Based on specifications, the state size differs between block cipher- and sponge-based AEAD. Several candidates used S-boxes and Boolean functions for the nonlinear operations of primary members, and only one candidate used modular operations as the nonlinear operations of primary members. According to the results of the software implementation, the RAM usage for each candidate did not differ. Additionally, the block cipher-based AEAD tends to have a lower latency and higher flash memory usage than the sponge-based AEAD. However, it is unclear whether this is due to the type of AEAD in this study.

We have investigated the relationship between the specifications and software implementation performance. Consequently, differences in parameters and operations cause differences in software implementation performance among different members of the same candidate. Based on the results above, the cost of nonlinear operations and the amount of RO data usage had the strongest correlation.

The experimental results on finalist candidates show that the latency of the AEAD increases step-by-step with the block size, as the data size increases. Moreover, the manner in which the latency changes for each candidate differs owing to the padding scheme and mode of operation. We can select algorithms that suit usage scenarios, and use them efficiently by understanding the specifications of the cryptography algorithms.

In this study, there was no difference in the amount of RAM usage. This is because the Mbed online compiler displayed memory usage in units of 0.1 kB, and there was no difference below 0.1 kB. Therefore, we will conduct a detailed investigation of RAM usage. In this paper, we focused on the latency and memory usage of the software implementation. In future, we will also investigate the security of each cipher against side-channel and fault attacks.

Acknowledgements This work was supported by JST AIP Acceleration Research under Grant JPMJCR20U2 and JSPS KAKENHI Grant Numbers JP20H00590 and JP18H05289.

References

1. Buchanan, W.J., Li, S., Asif, R.: Lightweight cryptography methods. Journal of Cyber Security Technology **1**(3-4), 187–201 (2017). https://doi.org/10.1080/23742917.2017.1384917, https://doi.org/10.1080/23742917.2017.1384917
2. CRYPTREC: Angou gijyutu tyousa wg (keiryou angou) houkokusyo (March 2015), https://www.cryptrec.go.jp/exreport/cryptrec-ex-2406-2014.pdf
3. CRYPTREC: Cryptographic technology guideline (lightweight cryptography) (June 2017), https://www.cryptrec.go.jp/report/cryptrec-gl-2003-2016jp.pdf
4. Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., Uhsadel, L.: A survey of lightweight-cryptography implementations. IEEE Design Test of Computers **24**(6), 522–533 (2007). https://doi.org/10.1109/MDT.2007.178
5. Hira, R., Li, Y., Hara-Azumi, Y., Sakiyama, K.: Survey for software implementation of the second round candidates in the nist lightweight cryptography. In: 2021 Symposium on Cryptography and Information Security (January 2021)
6. Manifavas, C., Hatzivasilis, G., Fysarakis, K., Rantos, K.: Lightweight cryptography for embedded systems – a comparative analysis. In: Garcia-Alfaro, J., Lioudakis, G., Cuppens-Boulahia, N., Foley, S., Fitzgerald, W.M. (eds.) Data Privacy Management and Autonomous Spontaneous Security. pp. 333–349. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54568-9_21
7. arm MBED: Build with the online compiler, https://os.mbed.com/docs/mbed-os/v6.5/quick-start/build-with-the-online-compiler.html
8. arm MBED: mbed lpc1114fn28, https://os.mbed.com/platforms/LPC1114FN28/
9. Mohajerani, K., Haeussler, R., Nagpal, R., Farahmand, F., Abdulgadir, A., Kaps, J.P., Gaj, K.: Fpga benchmarking of round 2 candidates in the nist lightweight cryptography standardization process: Methodology, metrics, tools, and results. IACR Cryptol. ePrint Arch. **2020**, 1207 (2020)
10. NIST: Submission requirements and evaluation criteria for the lightweight cryptography standardization process (August 2018), https://csrc.nist.gov/CSRC/media/Projects/Light weight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf
11. Sakiyama, K., Sasaki, Y., Li, Y.: Security of Block Ciphers: From Algorithm Design to Hardware Implementation, chap. Introduction to Block Ciphers, pp. 1–26. Wiley-IEEE Press (2015). https://doi.org/10.1002/9781118660027.ch1
12. Thakor, V.A., Razzaque, M.A., Khandaker, M.R.A.: Lightweight cryptography algorithms for resource-constrained iot devices: A review, comparison and research opportunities. IEEE Access **9**, 28177–28193 (2021). https://doi.org/10.1109/ACCESS.2021.3052867
13. Turan, M.S., McKay, K., Chang, D., Calik, C., Bassham, L., Kang, J., Kelsey, J.: Status report on the second round of the

nist lightweight cryptography standardization process (July 2021), https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8369.pdf
14. Turan, M.S., McKay, K.A., Calik, C., Chang, D., Bassham, L.: Status report on the first round of the nist lightweight cryptography standardization process (October 2019), https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8268.pdf
15. Yasuda, K., Sasaki, Y.: Cryptographic hash functions: How should we deal with the critical situation? IEICE Fundamentals Review **4**(1), 57–67 (2010), https://www.jstage.jst.go.jp/article/essfr/4/1/4_1_57/_pdf

A Survey of Specification

B Result of the Software Implementation Study

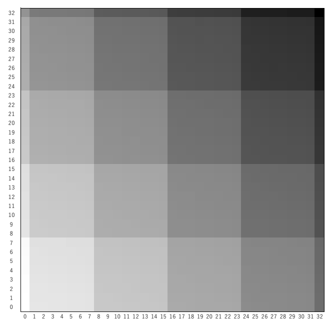
C Latency Heat Map of the Finalist Candidates

Table 6: Result of specification study

Algorithm	Member	Types of AEAD	Block size	Key size	State size	Tag size	Nonce size	Non-liner operation
ace	* aceae128v1	S	64	128	320	128	128	Boolean function
ascon	ascon128av12	S	128	128	320	128	128	Boolean function(5bit)
	* ascon128v12	S	64	128	320	128	128	Boolean function(5bit)
	ascon80pqv12	S	64	160	320	128	128	Boolean function(5bit)
comet	* comet128aesv1	B	128	128	256	128	128	S-box(8bit)
	comet64chamv1	B	64	128	196	64	120	modular operarion
	comet64speckv1	B	64	128	196	64	120	modular operarion
	comet128chamv1	B	128	128	256	128	128	modular operarion
dygascon	* drygascon128	S	128	320	448	128	128	Boolean function
	drygascon256	S	unknown	unknown	unknown	unknown	unknown	Boolean function
elephant	elephant200v1	other	200	128	200	128	96	Boolean function
	* elephant160v1	other	160	128	160	64	96	S-box(8bit)
	elephant176v1	other	176	128	176	64	96	S-box(8bit)
estate	sestatetweaes128v1	B	128	128	128	128	128	S-box(8bit)
	* estatetweaes128v1	B	128	128	128	128	128	S-box(8bit)
	estatetwegift128v1	B	128	128	128	128	128	S-box(4bit)
forkae	paefforkskinnyb128t256n112v1	B	128	128	128	128	112	S-box(8bit)
	saefforkskinnyb128t192n56v1	B	128	128	128	128	56	S-box(8bit)
	saefforkskinnyb128t256n120v1	B	128	128	128	128	120	S-box(8bit)
	* paefforkskinnyb128t288n104v1	B	128	128	128	128	104	S-box(8bit)
	paefforkskinnyb64t192n48v1	B	64	128	64	64	48	S-box(4bit)
	paefforkskinnyb128t192n48v1	B	128	128	128	128	48	S-box(8bit)
giftcofb	* giftcofb128v1	B	128	128	128	128	128	Boolean function(4bit)
gimli	* gimli24v1	S	128	256	384	128	128	Boolean function
grain-128-aead	* grain128aead	other	-	128	256	96	96	Boolean function
hyena	* hyenav1	B	128	128	128	128	96	S-box(4bit)
isap	isapa128av20	S	64	128	320	128	128	Boolean function(5bit)
	isapa128v20	S	64	128	320	128	128	Boolean function(5bit)
	* isapk128av20	S	144	128	400	128	128	Boolean function
	isapk128v20	S	144	128	400	128	128	Boolean function
knot	* knot128v1	S	64	128	256	128	128	Boolean function(4bit)
	knot192	S	96	192	384	192	192	Boolean function(4bit)
	knot256	S	128	256	512	256	256	Boolean function(4bit)
	knot128v2	S	192	128	384	128	128	Boolean function(4bit)
lotus/locus	twegift64locusaeadv1	B	64	128	64	64	128	S-box(4bit)
	* twegift64lotusaeadv1	B	64	128	64	64	128	S-box(4bit)
mixfeed	* mixfeed	B	128	128	128	128	120	S-box(8bit)
orange	* orangezestv1	S	256	128	256	128	128	S-box(4bit)
oribatida	oribatida192v12	S	96	128	192	96	64	Boolean function
	* oribatida256v12	S	128	128	256	128	128	Boolean function
photonbeete	* photonbeetleaead128rate128v1	S	128	128	256	128	128	S-box(4bit)
	photonbeetleaead128rate32v1	S	32	128	256	128	128	S-box(4bit)
pyjamask	* pyjamask128aeadv1	B	128	128	128	128	96	Boolean function (4bit)
	pyjamask96aeadv1	S	96	128	96	96	64	Boolean function(3bit)
rumulus	romulusn3v12	B	128	128	128	128	96	S-box(8bit)
	romulusm3v12	B	128	128	128	128	96	S-box(8bit)
	romulusn2v12	B	128	128	128	128	96	S-box(8bit)
	* romulusn1v12	B	128	128	128	128	128	S-box(8bit)
	romulusm1v12	B	128	128	128	128	128	S-box(8bit)
	romulusm2v12	B	128	128	128	128	96	S-box(8bit)
saeaes	saeaes128a120t64v1	B	64(120)	128	128	64	120	S-box(8bit)
	saeaes128a120t128v1	B	64(120)	128	128	128	120	S-box(8bit)
	saeaes128a64t64v1	B	64(64)	128	128	64	120	S-box(8bit)
	* saeaes128a64t128v1	B	64(64)	128	128	128	120	S-box(8bit)
	saeaes256a120t128v1	B	64(120)	256	128	128	120	S-box(8bit)
	saeaes192a120t128v1	B	64(120)	192	128	128	120	S-box(8bit)
	saeaes256a64t64v1	B	64(64)	256	128	64	120	S-box(8bit)
	saeaes256a64t128v1	B	64(64)	256	128	128	120	S-box(8bit)
	saeaes192a64t64v1	B	64(64)	192	128	64	120	S-box(8bit)
	saeaes192a64t128v1	B	64(64)	192	128	128	120	S-box(8bit)
saturnin	* saturninctrcascadev2	B	256	256	256	256	160	Boolean function(4bit)
	saturninshortv2	B	128	256	256	128	128	Boolean function(4bit)
sbterranean	subterraneanv1	S	32	128	257	128	unknown	Boolean function
skinnyaead	* skinnyaeadtk3128128v1	B	128	128	128	128	128	S-box(8bit)
	skinnyaeadtk296128v1	B	128	128	128	128	96	S-box(8bit)
	skinnyaeadtk29664v1	B	128	128	128	64	96	S-box(8bit)
	skinnyaeadtk312864v1	B	128	128	128	64	128	S-box(8bit)
	skinnyaeadtk396128v1	B	128	128	128	128	96	S-box(8bit)
	skinnyaeadtk39664v1	B	128	128	128	64	96	S-box(8bit)
sparkle	schwaemm128128v1	S	128	128	256	128	128	modular operarion
	schwaemm256128v1	S	256	128	384	128	256	modular operarion
	schwaemm192192v1	S	192	192	384	192	192	modular operarion
	schwaemm256256v1	S	256	256	512	256	256	modular operarion
spix	* spix128v1	S	64	128	256	128	128	Boolean function
spoc	* spoc128sliscplight256v1	S	128	128	256	128	128	Boolean function(64bit)
	spoc64sliscplight192v1	S	64	128	192	64	128	Boolean function(64bit)
spook	spook128mu512v1	S	256	128	512	128	128	Boolean function(4bit)
	* spook128su512v1	S	256	128	512	128	128	Boolean function(4bit)
	spook128mu384v1	S	128	128	384	128	128	Boolean function(4bit)
	spook128su384v1	S	128	128	384	128	128	Boolean function(4bit)
sundae	sundaegift0v1	B	128	128	128	128	0	Boolean function(4bit)
	sundaegift64v1	B	128	128	128	128	64	Boolean function(4bit)
	* sundaegift96v1	B	128	128	128	128	96	Boolean function(4bit)
	sundaegift128v1	B	128	128	128	128	128	Boolean function(4bit)
tinyjambu	* tinyjambu128	B	32	128	128	64	94	Boolean function
	tinyjambu256	B	32	256	128	64	96	Boolean function
	tinyjambu192	B	32	192	128	64	96	Boolean function
wage	* wageae128v1	S	64	128	259	128	128	S-box(7bit)
xoodyak	* xoodyakv1	S	128	128	384	128	unknown	Boolean function

Table 7: Software evaluation study

Algorithm	Member	Encryption time(seconds)	Decryption time(seconds)	RAM usage(kB)	RW Data(kB)	ZI Data(kB)	ROM usage(kB)	Code(kB)	RO Data(kB)
Comparison	-	0.003	0.003	0.9	0.2	0.7	14.1	12.2	1.9
ace	* aceae128v1	8.261	8.263	0.9	0.2	0.7	17.5	15.4	2.1
ascon	ascon128av12	0.153	0.155	0.9	0.2	0.7	30.6	28.6	2.0
	* ascon128v12	0.183	0.185	0.9	0.2	0.7	31.4	29.4	2.0
	ascon80pqv12	0.185	0.188	0.9	0.2	0.7	31.3	29.3	2.0
comet	* comet128aesv1	0.240	0.242	0.9	0.2	0.7	22.2	15.9	6.3
	comet64chamv1	0.591	0.593	0.9	0.2	0.7	17.4	15.4	2.0
	comet64speckv1	0.594	0.597	0.9	0.2	0.7	17.6	15.5	2.0
	comet128chamv1	1.233	1.235	0.9	0.2	0.7	17.7	15.8	2.0
drygascon	* drygascon128	3.803	3.806	0.9	0.2	0.7	17.5	15.6	2.0
	drygascon256	10.988	10.993	0.9	0.2	0.7	17.5	15.5	2.0
elephant	elephant200v1	11.095	11.095	0.9	0.2	0.7	17.0	14.9	2.1
	* elephant160v1	309.186	309.187	1.9	1.2	0.7	16.4	14.4	2.0
	elephant176v1	362.768	362.769	1.9	1.2	0.7	16.4	14.4	2.0
estate	sestatetweaes128v1	0.310	0.312	0.9	0.2	0.7	17.2	14.7	2.5
	* estatetweaes128v1	0.346	0.348	0.9	0.2	0.7	17.2	14.7	2.5
	estatetwegift128v1	10.212	10.213	0.9	0.2	0.7	17.2	14.9	2.3
forkae	paefforkskinnyb128t256n112v1	1.209	1.690	0.9	0.2	0.7	20.9	18.3	2.6
	saefforkskinnyb128t192n56v1	1.210	1.692	0.9	0.2	0.7	20.7	18.1	2.6
	saefforkskinnyb128t256n120v1	1.211	1.694	0.9	0.2	0.7	20.7	18.1	2.6
	* paefforkskinnyb128t288n104v1	1.992	3.045	0.9	0.2	0.7	21.3	18.7	2.6
	paefforkskinnyb64t192n48v1	2.464	3.770	1.1	0.2	0.9	21.5	19.4	2.1
	paefforkskinnyb128t192n48v1	1.205	1.689	0.9	0.2	0.7	20.9	18.4	2.6
giftcofb	* giftcofb128v1	1.806	1.807	0.9	0.2	0.7	17.1	15.1	2.0
gimli	* gimli24v1	0.441	0.444	0.9	0.2	0.7	16.4	14.4	2.0
grain-128-aead	* grain128aead	23.854	23.831	0.9	0.2	0.7	17.8	15.8	2.0
hyena	* hyenav1	6.966	6.967	0.9	0.2	0.7	17.6	15.2	2.3
isap	isapa128av20	2.791	2.793	0.9	0.2	0.7	16.5	14.5	2.0
	isapa128v20	11.414	11.416	0.9	0.2	0.7	16.5	14.5	2.0
	* isapk128av20	46.794	46.796	0.9	0.2	0.7	16.6	14.5	2.2
	isapk128v20	394.973	394.975	0.9	0.2	0.7	16.6	14.5	2.2
knot	* knot128v1	unknown	unknown	1.0	0.2	0.7	16.8	14.8	2.0
	knot192	unknown	unknown	1.0	0.3	0.7	17.6	15.6	2.0
	knot256	unknown	unknown	1.0	0.3	0.7	17.9	15.9	2.0
	knot128v2	unknown	unknown	1.0	0.3	0.7	17.3	15.3	2.0
lotus/locus	twegift64locusaeadv1	8.876	8.929	0.9	0.2	0.7	18.2	15.9	2.2
	* twegift64lotusaeadv1	9.048	9.049	0.9	0.2	0.7	18.1	15.9	2.2
mixfeed	* mixfeed	1.041	1.041	1.1	0.4	0.7	17.2	15.2	2.0
orange	* orangezestv1	10.373	10.373	0.9	0.2	0.7	17.9	15.8	2.1
oribatida	oribatida192v12	176.166	176.179	0.9	0.2	0.7	19.2	16.7	2.5
	* oribatida256v12	286.61	286.613	0.9	0.2	0.7	19.0	16.5	2.5
photonbeete	* photonbeetleaead128rate128v1	12.251	12.253	0.9	0.2	0.7	17.5	15.4	2.1
	photonbeetleaead128rate32v1	29.655	29.658	0.9	0.2	0.7	17.6	15.5	2.1
pyjamask	* pyjamask128aeadv1	1.754	1.754	0.9	0.2	0.7	17.3	15.3	2.0
	pyjamask96aeadv1	1.738	1.723	0.9	0.2	0.7	17.4	15.4	2.0
rumulus	romulusn3v12	3.214	3.216	1.0	0.2	0.7	19.5	16.9	2.6
	romulusm3v12	4.185	4.188	1.0	0.2	0.7	19.7	17.1	2.6
	romulusn2v12	4.943	4.944	1.0	0.2	0.7	19.7	17.1	2.6
	* romulusn1v12	4.959	4.962	1.0	0.2	0.7	19.6	17.0	2.6
	romulusm1v12	6.126	6.128	1.0	0.2	0.7	19.5	17.0	2.6
	romulusm2v12	6.376	6.378	1.0	0.2	0.7	19.8	17.2	2.6
saeaes	saeaes128a120t64v1	0.098	0.099	0.8	0.2	0.6	21.5	15.5	6.1
	saeaes128a120t128v1	0.099	0.101	0.9	0.2	0.7	22.3	16.3	6.0
	saeaes128a64t64v1	0.110	0.112	0.9	0.2	0.7	22.4	16.3	6.0
	* saeaes128a64t128v1	0.111	0.113	0.9	0.2	0.7	22.3	16.3	6.0
	saeaes256a120t128v1	0.130	0.133	0.9	0.2	0.7	23.4	17.4	6.0
	saeaes192a120t128v1	0.141	0.143	0.9	0.2	0.7	23.0	16.9	6.0
	saeaes256a64t64v1	0.147	0.149	0.9	0.2	0.7	23.5	17.4	6.0
	saeaes256a64t128v1	0.148	0.150	0.9	0.2	0.7	23.4	17.4	6.0
	saeaes192a64t64v1	0.155	0.157	0.9	0.2	0.7	23.0	17.0	6.0
	saeaes192a64t128v1	0.156	0.158	0.9	0.2	0.7	23.0	16.9	6.0
saturnin	* saturninctrcascadev2	0.527	0.528	0.9	0.2	0.7	18.1	16.1	2.0
	saturninshortv2	unknown	unknown	0.9	0.2	0.7	19.9	17.9	2.0
sbterranean	subterraneanv1	3.117	3.120	0.9	0.2	0.7	18.8	15.5	3.3
	* skinnyaeadtk3128128v1	unknown	unknown	0.9	0.2	0.7	20.6	17.9	2.7
	skinnyaeadtk296128v1	unknown	unknown	0.9	0.2	0.7	20.7	18.0	2.7
	skinnyaeadtk29664v1	unknown	unknown	0.9	0.2	0.7	20.9	18.3	2.7
	skinnyaeadtk312864v1	unknown	unknown	0.9	0.2	0.7	20.8	18.1	2.7
	skinnyaeadtk396128v1	unknown	unknown	0.9	0.2	0.7	20.6	17.9	2.7
	skinnyaeadtk39664v1	unknown	unknown	0.9	0.2	0.7	20.8	18.1	2.7
sparkle	schwaemm128128v1	0.221	0.222	0.9	0.2	0.7	16.9	14.9	2.0
	schwaemm256128v1	0.269	0.269	0.9	0.2	0.7	17.1	15.1	2.0
	schwaemm192192v1	0.333	0.313	0.9	0.2	0.7	17.0	15.0	2.0
	schwaemm256256v1	0.353	0.354	0.9	0.2	0.7	17.2	15.1	2.0
spix	* spix128v1	4.734	4.736	0.9	0.2	0.7	17.8	15.7	2.1
spoc	* spoc128sliscplight256v1	2.371	2.371	0.9	0.2	0.7	17.0	14.9	2.1
	spoc64sliscplight192v1	2.813	2.812	0.9	0.2	0.7	17.0	14.9	2.1
spook	spook128mu512v1	0.353	0.359	0.9	0.2	0.7	18.0	15.9	2.2
	* spook128su512v1	0.354	0.359	0.9	0.2	0.7	18.1	15.9	2.2
	spook128mu384v1	0.429	0.435	0.9	0.2	0.7	18.1	15.9	2.2
	spook128su384v1	0.429	0.435	0.9	0.2	0.7	18.1	15.9	2.2
sundae	sundaegift0v1	2.411	2.414	0.9	0.2	0.7	17.5	15.5	2.0
	sundaegift64v1	2.657	2.660	0.9	0.2	0.7	17.5	15.5	2.0
	* sundaegift96v1	2.765	2.768	0.9	0.2	0.7	17.5	15.5	2.0
	sundaegift128v1	2.874	2.877	0.9	0.2	0.7	17.5	15.5	2.0
tinyjambu	* tinyjambu128	0.114	0.115	0.9	0.2	0.7	15.7	13.7	2.0
	tinyjambu256	0.128	0.129	0.9	0.2	0.7	15.7	13.7	2.0
	tinyjambu192	1.338	1.339	0.9	0.2	0.7	15.7	13.7	2.0
wage	* wageae128v1	4.938	4.945	0.9	0.2	0.7	19.4	16.8	2.6
xoodyak	* xoodyakv1	0.173	0.174	0.9	0.2	0.7	16.3	14.3	2.0


(a) raw-data

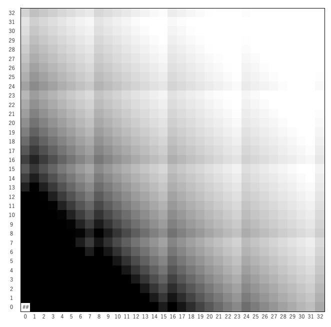
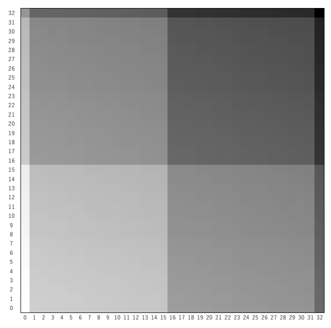

(b) divided-data

Fig. 3: ascon80


(a) raw-data

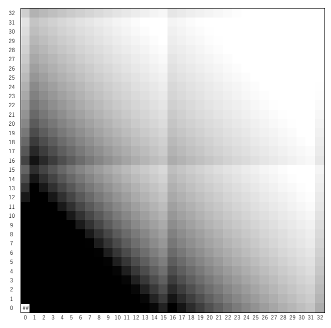
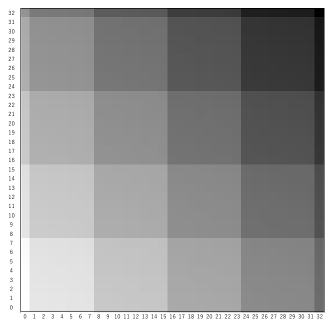

(b) divided-data

Fig. 4: ascon128_a


(a) raw-data

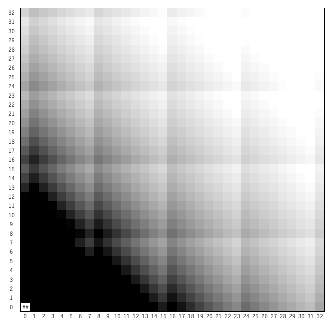
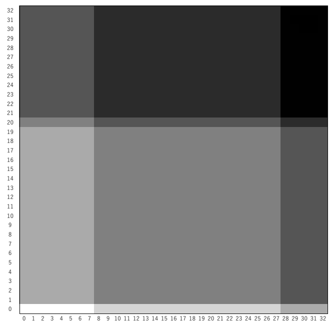

(b) divided-data

Fig. 5: ascon128


(a) raw-data

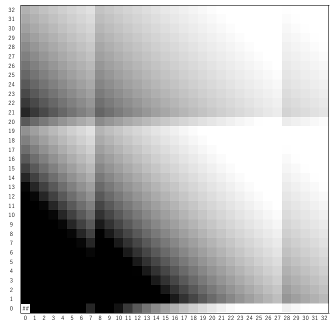
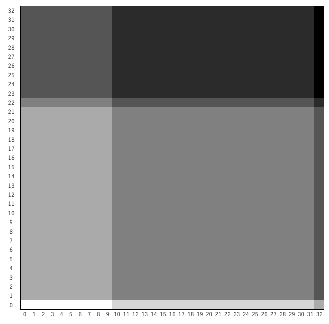

(b) divided-data

Fig. 6: elephant160


(a) raw-data

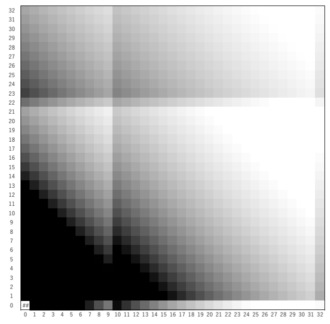
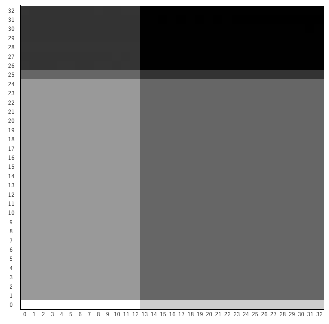

(b) divided-data

Fig. 7: elephant176


(a) raw-data

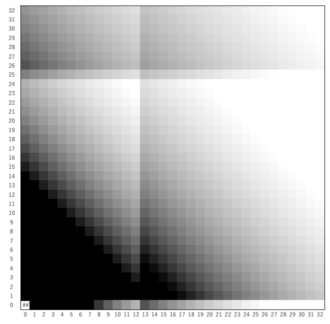
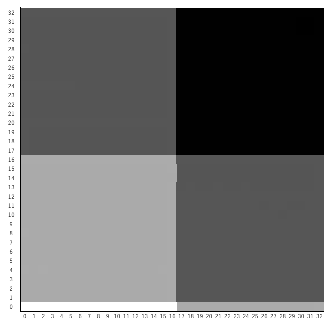

(b) divided-data

Fig. 8: elephant200


(a) raw-data

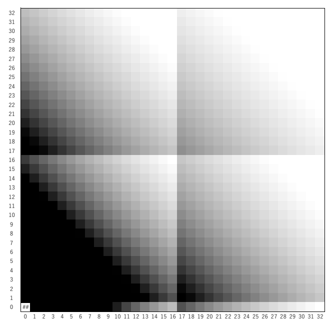
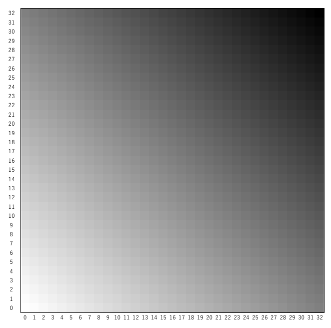

(b) divided-data

Fig. 9: giftcofb


(a) raw-data

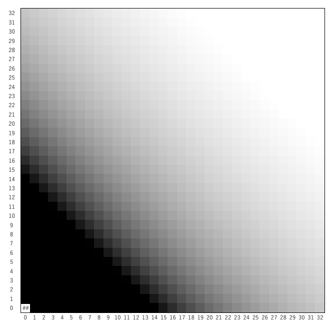
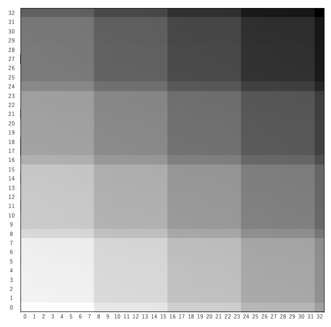

(b) divided-data

Fig. 10: grain128aead


(a) raw-data

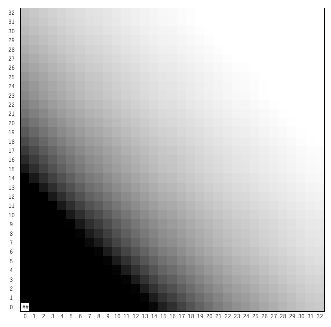
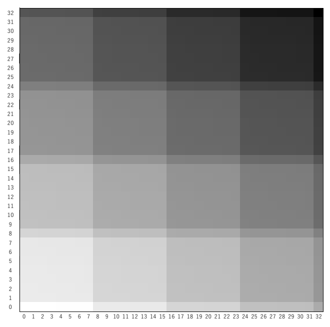

(b) divided-data

Fig. 11: isapa128_a


(a) raw-data

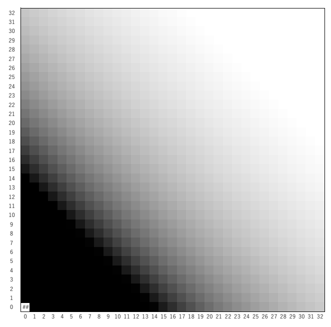
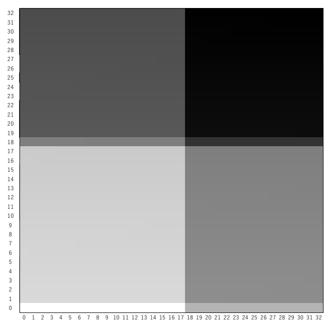

(b) divided-data

Fig. 12: isapa128


(a) raw-data

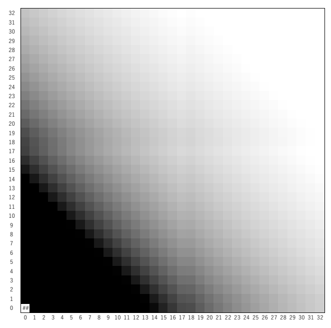
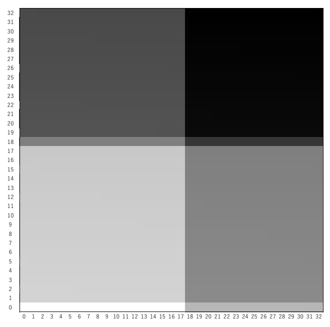

(b) divided-data

Fig. 13: isapk128_a


(a) raw-data

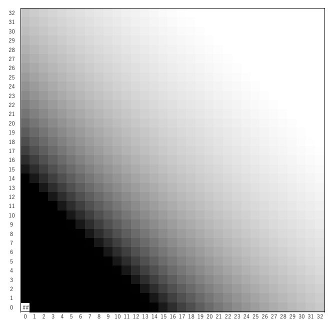
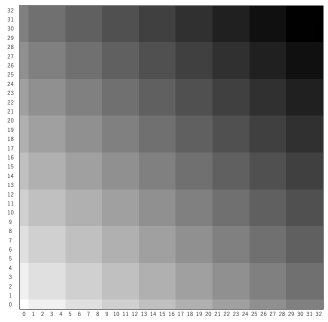

(b) divided-data

Fig. 14: isapk128_a


(a) raw-data

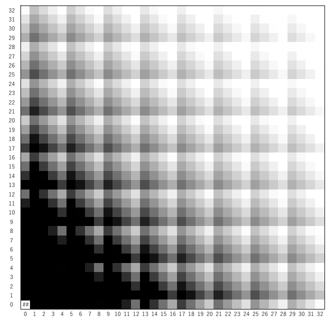
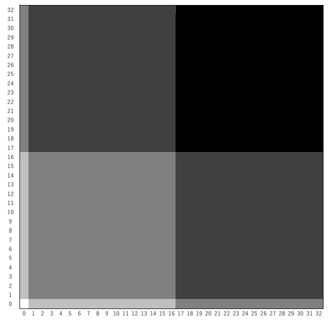

(b) divided-data

Fig. 15: photonbeetle128_32


(a) raw-data

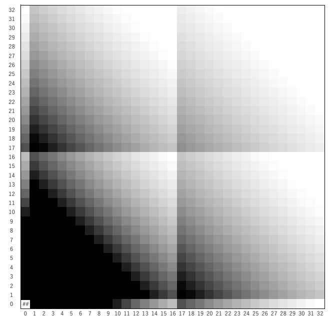
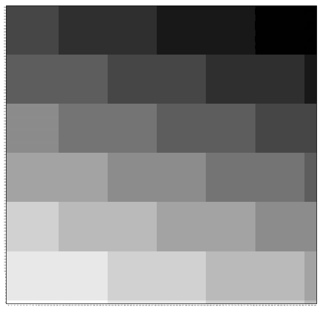

(b) divided-data

Fig. 16: photonebeetle128_128


(a) raw-data

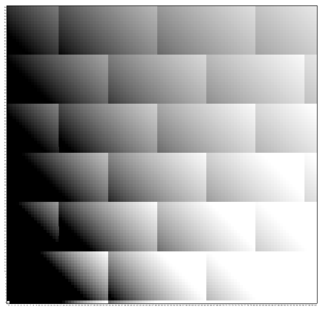
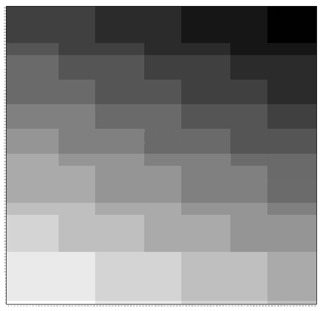

(b) divided-data

Fig. 17: romurus_m1


(a) raw-data

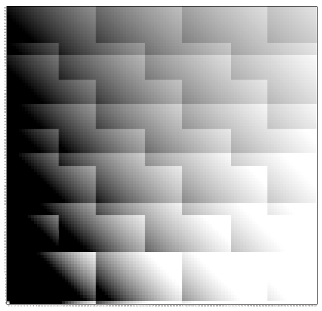
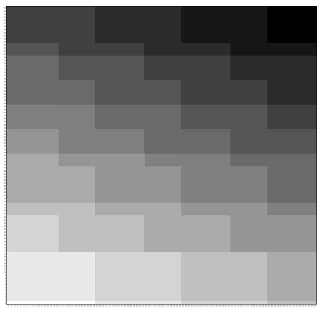

(b) divided-data

Fig. 18: romurus_m2


(a) raw-data

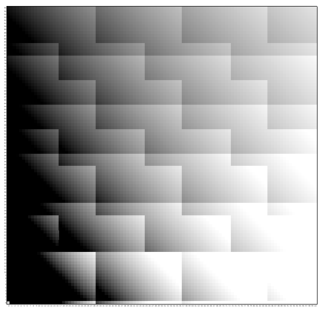
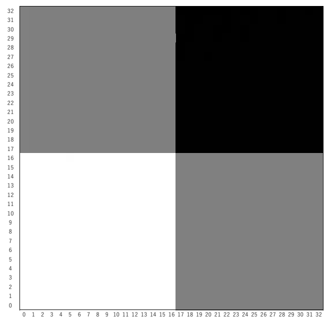
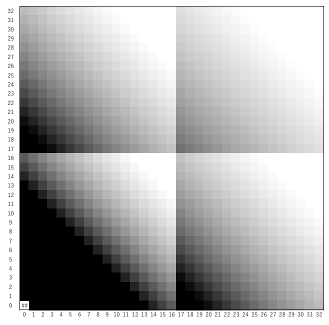

(b) divided-data

Fig. 19: romurus_m3

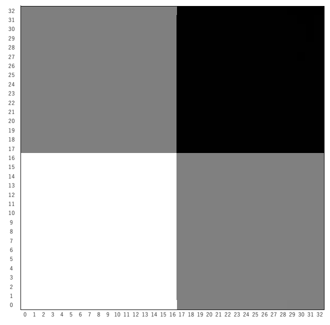


(a) raw-data



(b) divided-data

Fig. 20: romuruls_n1


(a) raw-data

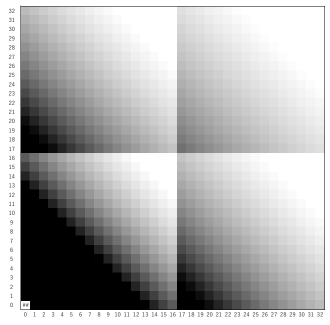
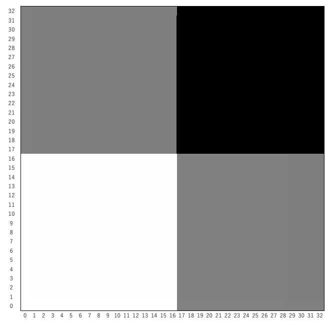

(b) divided-data

Fig. 21: romulus_n2


(a) raw-data

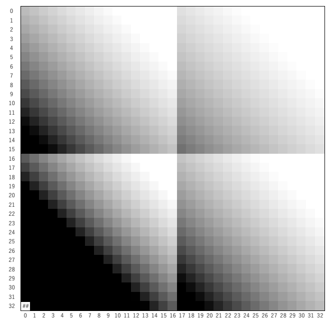
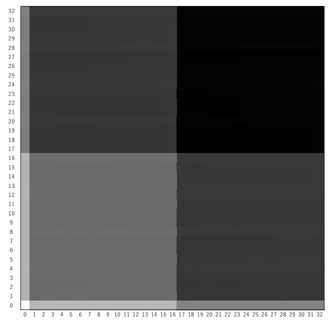

(b) divided-data

Fig. 22: romulusu_n3


(a) raw-data

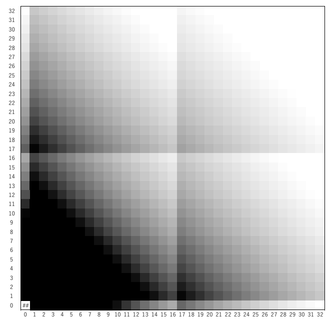
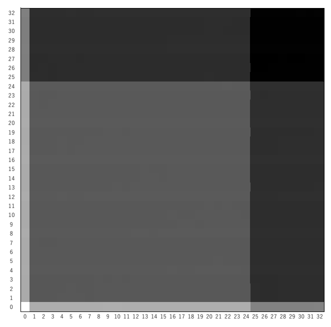

(b) divided-data

Fig. 23: schwaemm128_128


(a) raw-data

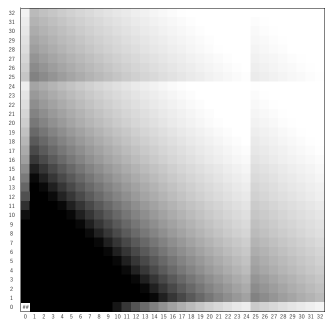
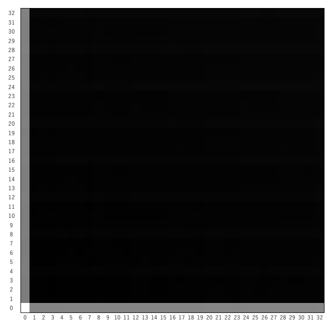

(b) divided-data

Fig. 24: schwaemm192_192


(a) raw-data

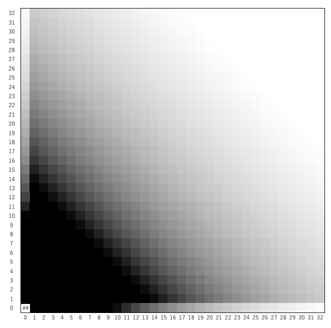
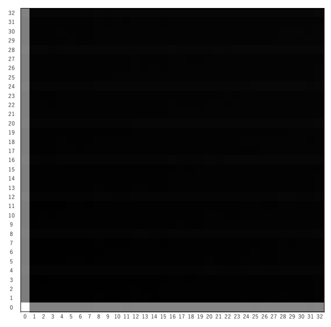

(b) divided-data

Fig. 25: schwaemm256_128


(a) raw-data

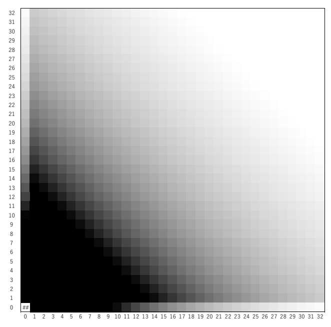
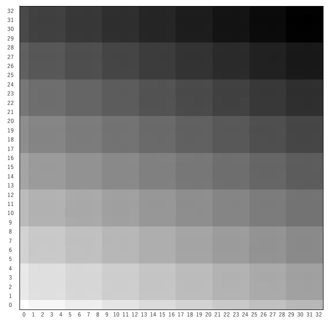

(b) divided-data

Fig. 26: schwaemm256_256


(a) raw-data

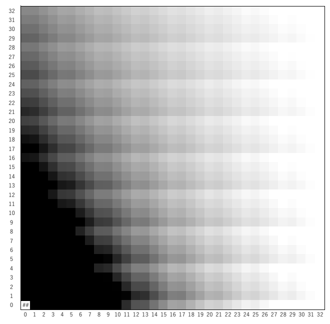
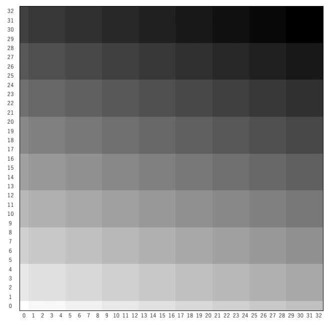

(b) divided-data

Fig. 27: tinyjambu128


(a) raw-data

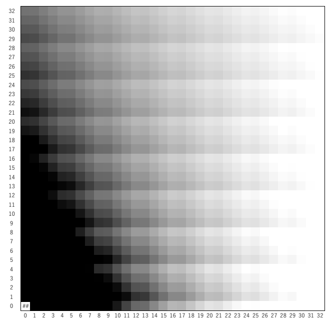
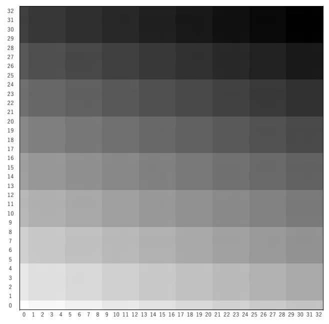

(b) divided-data

Fig. 28: tinyjambu192


(a) raw-data

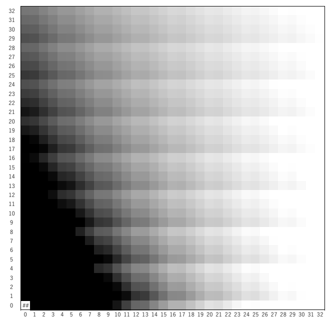
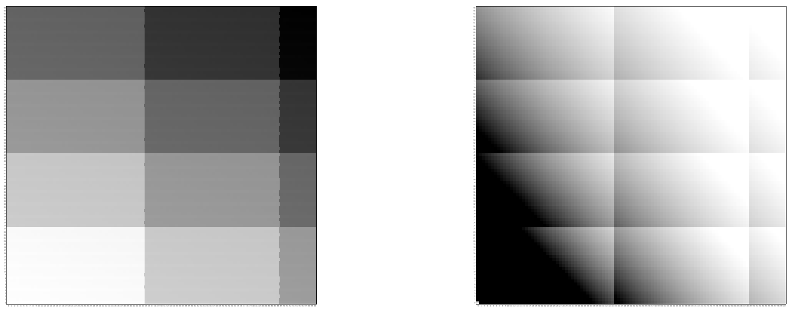

(b) divided-data

Fig. 29: tinyjambu256



(a) raw-data **(b)** divided-data

Fig. 30: xoodyak