CRYPTOGRAPHIC ALGORITHM IMPLEMENTATION AND VALIDATION OF SHA3 AND XTS-AES

by

Nishant Raj & Suraj Kumar
Dept of Computer Science and Engineering
National Institute of Technology Karnataka,
Surathkal

Guide: Prof N Balakrishnan

Supercomputer Education and Research Centre
Indian Institute of Science, Bengaluru

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to everyone who contributed in the completion of this report. I would like to thank The Indian Institute of Science for providing me this opportunity to be a part of the Summer Internship 2019.

I express my sincere gratitude to Professor N. Balakrishnan, Department of Supercomputer Education and Research Centre, IISc Bengaluru who in spite of his busy schedule, spent time to hear and guide me throughout the completion of this project.

I would like to thank research assistant Ms. Swetha H for her help and encouragement in completing this project. I would also like to thank Prof. Alwyn Roshan Pais for providing us this opportunity. I also thank my parents for their unceasing encouragement and support.

This internship was a great learning experience. I will strive to use the gained knowledge in the best possible way and will continue to work on its improvement.

CERTIFICATE

This is to certify that the project entitled '**Cryptographic algorithm implementation and validation of SHA3 and XTS-AES**' submitted to the Indian Institute of Sciences, by ***Nishant Raj and Suraj Kumar***, is accepted as a record of work carried out by them as part of the Summer Internship Programme 2019.

Prof N Balakrishnan
Honorary Professor

Supercomputer Education and Research Centre
Indian Institute of Science

(Guide)

Contents

Abstract

One of the important steps in certifying cryptographic algorithm is Cryptographic Algorithmic Validation Program (CAVP). NIST has enunciated a procedure for carrying out algorithmic validation as implemented for publicly known as well as non-publicly known key ciphers and their known variants. NIST has also provided for each of these algorithms a set of test vectors and expected response. The algorithm that we have chosen for implementation are hash function (SHA3 bit oriented) and block cipher mode of operation (XTS-AES). Their implementation is not publicly available. SHA3 algorithm uses Keccak Permutation. We have implemented the algorithm in Java language on Windows based OS.

This report explains about different hash functions and block cipher modes of operation.

First part explains about the SHA3 which provides integrity for the message, and the implementation of SHA3(Bit Oriented). Further it explains the test methodology which uses the NIST recommended and FIPS approved test vectors for each of the algorithm and the expected response.

Next part explains about the XTS-AES which provides confidentiality for the message, and the implementation of XTS-AES. Further it explains the test methodology which uses the NIST recommended and FIPS approved test vectors for each of the algorithm and the expected response.

INTRODUCTION

Crypto Product (Crypto module) certification consists of two parts, namely Cryptographic Algorithm Validation Program (CAVP) and Cryptographic Module Validation Program under CAVP..Crypto Modules should use only those algorithm implementations that are certified by CAVP. Cryptographic algorithm validation system provides validation testing of FIPS-approved and NIST-Recommended Cryptographic Algorithm .CAVP program various stakeholders are Vendor, CST Lab, Validation authority and Users .Vendor is responsible for implementing cryptographic algorithms as per the Standard, CST laboratory validates different implementation across vendors by using the Cryptographic Algorithm Validation System (CAVS) tool, CAVP Validation authorities (VA) provides the CAVS Tool which is in turn used for validation and User verifies that cryptographic module or a product meets the requirement.

The most critical component of CAVP is CAVS tool. CAVS comprises of validation test suites that verifies the correctness of the vendors implemented algorithm using NIST test vectors. The version 2.0 of the CAVS tool designed and developed at 11Sc is an improvement over the previous version 1.0 of CAVS Tool. IISC implementation of CAVS 2.0 is GUI based tool which is controlled by CST admin and used by Tester.

CST admin’s responsibilities are

- Adding Vendors, IUT for validation algorithms while providing flexibility in selection of any mode or key sizes for any algorithm.
- Assigning particular validation to specific tester
- Keeping track of file repositories
- Maintenance database
- Submitting validation report to certifying authority

Tester's responsibilities are

- Selection of test vectors
- Processing of Vendor's responses
- Generation of error log files
- Generation of validation report

NIST recommended Cryptographic Algorithm

There are five groups of cryptographic algorithms: symmetric key encryption, public key encryption, cryptographic hash functions, message authentication codes, and digital signatures.

We have implemented cryptographic hash functions:

- SHA3 (BIT oriented implementation)
- XTS AES block cipher mode of operation.

1 . CRYPTOGRAPHY ALGORITHM

Cryptography or **cryptology** is the practice and study of techniques for secure communication in the presence of third parties called adversaries. Cryptography is the practice and study of techniques for securing communication and data in the presence of adversaries.

Security Services

- **Authentication**
- **Confidentiality**
- **Integrity**
- **Access Control**
- **Non-repudiation**
- **Availability**

Cryptography Algorithm

- **Symmetric Encryption (AES, DES)**
- **Asymmetric Encryption (RSA)**
- **Hashing Algorithm (SHA3)**

1.1 SHA3

SHA-3 (**Secure Hash Algorithm 3**) is the latest member of the Secure Hash Algorithm family of standards, released by NIST on August 5, 2015. Although part of the same series of standards, SHA-3 is internally different from the MD5-like structure of SHA-1 and SHA-2.

SHA-3 is a subset of the broader cryptographic primitive family **Keccak**. Keccak is based on a novel approach called sponge construction. Sponge construction is based on a wide random function or random permutation, and allows inputting ("absorbing" in sponge terminology) any amount of data, and outputting ("squeezing") any amount of data, while acting as a pseudorandom function with regard to all previous inputs. This leads to great flexibility.

1.2 XTS-AES

Disk encryption is a special case of data at rest protection when the storage medium is a sector-addressable device (e.g., a hard disk).

Disk encryption methods aim to provide three distinct properties:

1. The data on the disk should remain confidential.

2. Data retrieval and storage should both be fast operations, no matter where on the disk the data is stored.
3. The encryption method should not waste disk space (i.e., the amount of storage used for encrypted data should not be significantly larger than the size of plaintext).

XTS – XEX-based tweaked-codebook mode with Ciphertext stealing – in this mode we still execute XOR function between blocks but also add additional tweak key to improve permutation.

2 . SHA3

2.1 INTRODUCTION

The Secure Hash Algorithm (SHA-1) has not yet been “broken.” That is, no one has demonstrated a technique for producing collisions in a practical amount of time. However, because SHA-1 is very similar, in structure and in the basic mathematical operations used, to MD5 and SHA-0, both of which have been broken, SHA-1 is considered insecure and has been phased out for SHA-2. SHA-2, particularly the 512-bit version, would appear to provide unassailable security. However, SHA-2 shares the same structure and mathematical operations as its predecessors, and this is a cause for concern. Because it will take years to find a suitable replacement for SHA-2, should it become vulnerable, NIST decided to begin the process of developing a new hash standard. Accordingly, NIST announced in 2007 a competition to produce the next generation NIST hash function, to be called SHA-3. The winning design for SHA-3 was announced by NIST in October 2012. SHA-3 is a cryptographic hash function that is intended to complement SHA-2 as the approved standard for a wide range of applications.

2.2 Types of SHA3

Instance	Output size *d*	Rate *r* = block size	Capacity *c*	Definition	Security strengths in bits		
					Collision	Preimage	2nd preimage
SHA3-224(*M*)	224	1152	448	Keccak[448](*M* || 01, 224)	112	224	224
SHA3-256(*M*)	256	1088	512	Keccak[512](*M* || 01, 256)	128	256	256
SHA3-384(*M*)	384	832	768	Keccak[768](*M* || 01, 384)	192	384	384
SHA3-512(*M*)	512	576	1024	Keccak[1024](*M* || 01, 512)	256	512	512

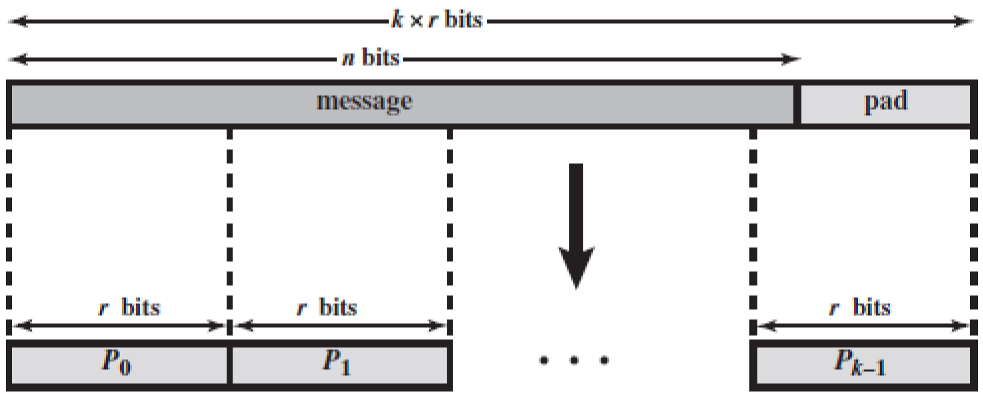
2.3 Implementation

The sponge specification proposes [BERT11] proposes two padding schemes:

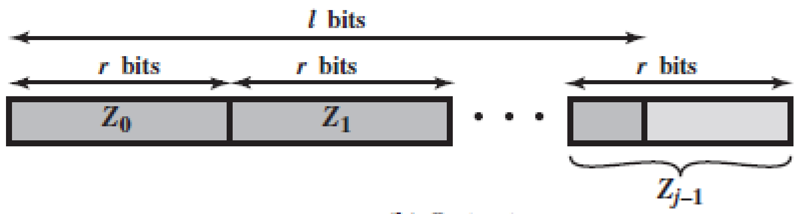
• **Simple padding:** Denoted by pad10*, appends a single bit 1 followed by the minimum number of bits 0 such that the length of the result is a multiple of the block length.

• **Multirate padding:** Denoted by pad10*1, appends a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the length of the result is a multiple of the block length. This is the simplest padding scheme that allows secure use of the same *f* with different rates *r*.

Input / Output



(a) Input



(b) Output

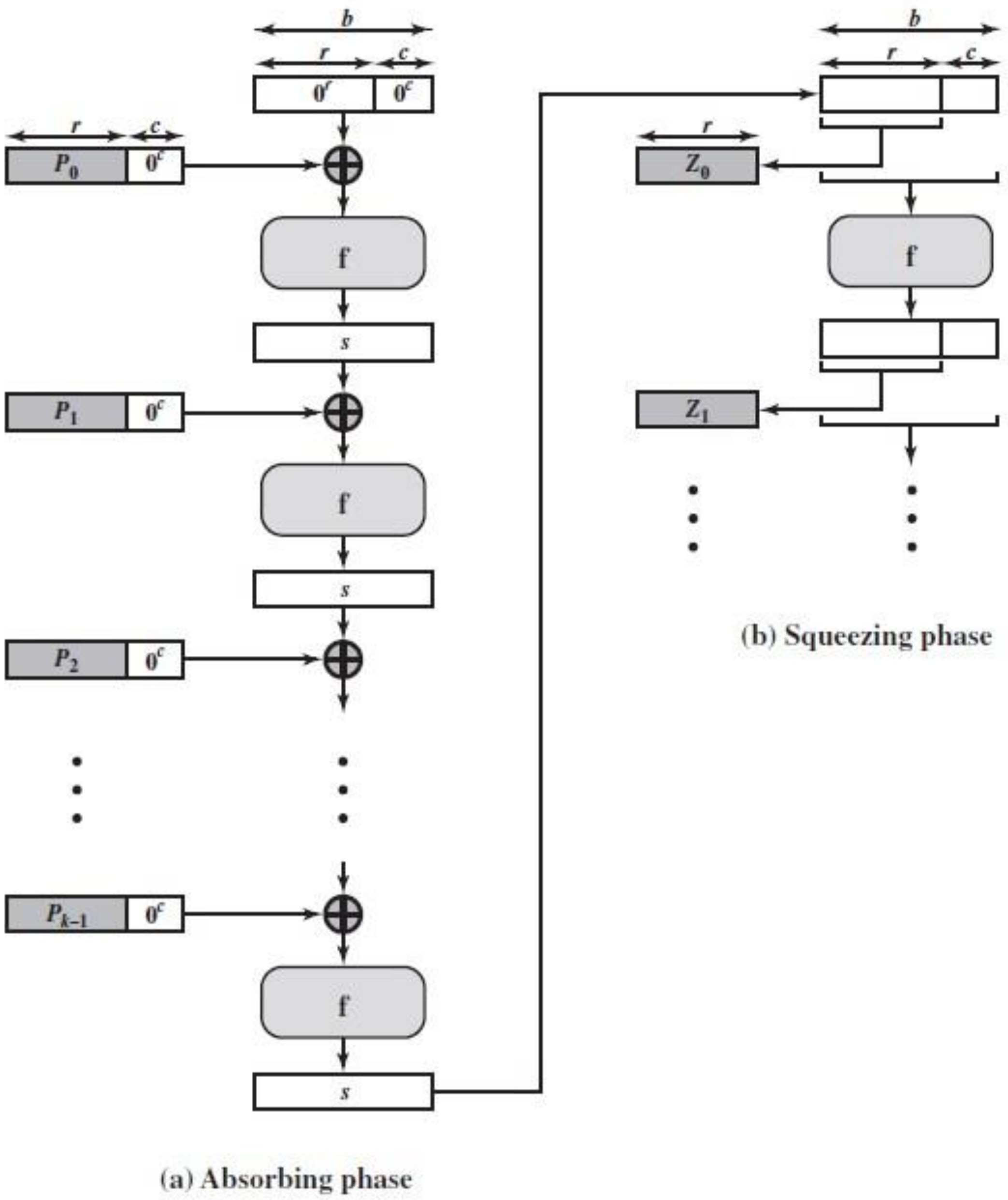
The Sponge Construction

The underlying structure of SHA-3 is a scheme referred to by its designers as a **sponge construction** [BERT07, BERT11]. The sponge construction has the same general structure as other iterated hash functions (Figure 11.8). The sponge function takes an input message and partitions it into fixed-size blocks. Each block is processed in turn with the output of each iteration fed into the next iteration, finally producing an output block. The sponge function is defined by three parameters:

f = the internal function used to process each input block3

r = the size in bits of the input blocks, called the **bitrate**

pad = the padding algorithm



The sponge construction consists of two phases. The **absorbing phase** proceeds as follows: For each iteration, the input block to be processed is padded with zeroes to extend its length from *r* bits to *b* bits. Then, the bitwise XOR of the extended message block and *s* is formed to create a *b*-bit input to the iteration function *f*. The output of *f* is the value of *s* for the next iteration.

If the desired output length ℓ satisfies ℓ … *b*, then at the completion of the absorbing phase, the first ℓ bits of *s* are returned and the sponge construction terminates. Otherwise, the sponge construction enters the **squeezing phase**.

In terms of the sponge algorithm defined above, Keccak[*r*, *c*] is defined as

Keccak[*r*, *c*] Δ SPONGE[Keccak-*f* [*r* + *c*], pad10 * 1, *r*]

Structure of *f*

The application of the five steps can be expressed as the composition of functions:

R = *i* o x o p o r o u

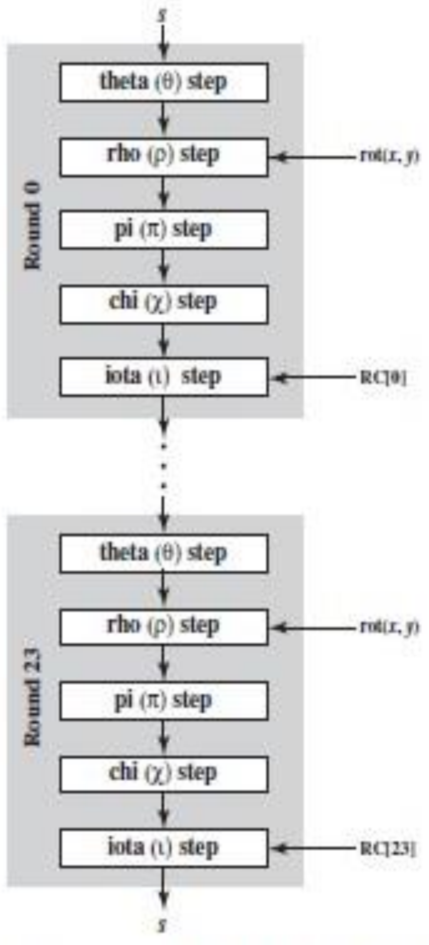
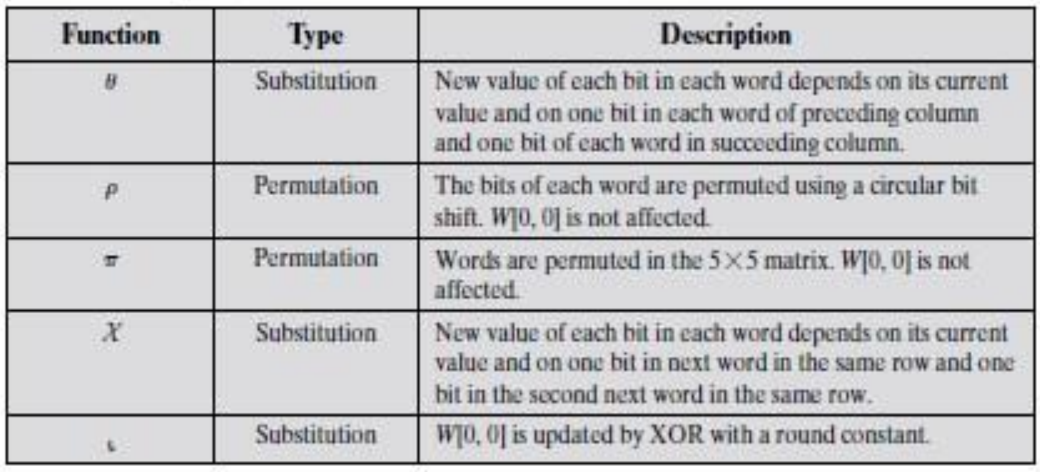


Figure 11.17 SHA-3 Iteration Function *f*

Table 11.6 Step Functions in SHA-3



Function	Type	Description
θ	Substitution	New value of each bit in each word depends on its current value and on one bit in each word of preceding column and one bit of each word in succeeding column.
ρ	Permutation	The bits of each word are permuted using a circular bit shift. *W*[0, 0] is not affected.
π	Permutation	Words are permuted in the 5×5 matrix. *W*[0, 0] is not affected.
χ	Substitution	New value of each bit in each word depends on its current value and on one bit in next word in the same row and one bit in the second next word in the same row.
ι	Substitution	*W*[0, 0] is updated by XOR with a round constant.

Algorithm

```
Keccak[r,c](Mbytes || Mbits) {
  # Padding
  d = 2^|Mbits| + sum for i=0..|Mbits|-1 of 2^i*Mbits[i]
  P = Mbytes || d || 0x00 || … || 0x00
  P = P xor (0x00 || … || 0x00 || 0x80)

  # Initialization
  S[x,y] = 0,                                for (x,y) in (0…4,0…4)

  # Absorbing phase
  for each block Pi in P
    S[x,y] = S[x,y] xor Pi[x+5*y],          for (x,y) such that x+5*y < r/w
    S = Keccak-f[r+c](S)

  # Squeezing phase
  Z = empty string
  while output is requested
    Z = Z || S[x,y],                        for (x,y) such that x+5*y < r/w
    S = Keccak-f[r+c](S)

  return Z
}
```

2.3.1 Byte Oriented Implementation

Input: Is the multiple of 8 bits (1 Byte).

2.3.2 Bit Oriented Implementation

Input: Is of any size.

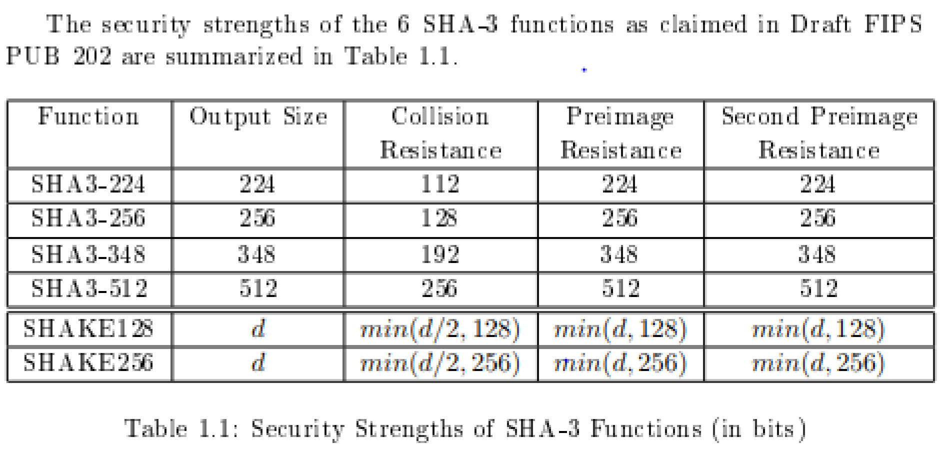
Using padding is to make is the multiple of 8 bits.

Then use the byte-oriented algorithm.

2.4 Security Analysis

Standard applications of hash functions typically require core security properties of collision resistance, preimage resistance, and second preimage resistance. Given a hash function H with an n-bit output, these core security properties are defined below

1. Collision resistance: It should be difficult to find a pair of different messages m1 and m2 such that H(m1) = H(m2).
2. Preimage resistance: Given an arbitrary n-bit value x, it should be difficult to find any message m such that H(m) = x.
3. Second preimage resistance: Given message m1, it should be difficult to find any different message m2 such that H(m1) = H(m2).



The security strengths of the 6 SHA-3 functions as claimed in Draft FIPS PUB 202 are summarized in Table 1.1.

Function	Output Size	Collision Resistance	Preimage Resistance	Second Preimage Resistance
SHA3-224	224	112	224	224
SHA3-256	256	128	256	256
SHA3-348	348	192	348	348
SHA3-512	512	256	512	512
SHAKE128	d	$min(d/2, 128)$	$min(d, 128)$	$min(d, 128)$
SHAKE256	d	$min(d/2, 256)$	$min(d, 256)$	$min(d, 256)$

Table 1.1: Security Strengths of SHA-3 Functions (in bits)

2.5 Results

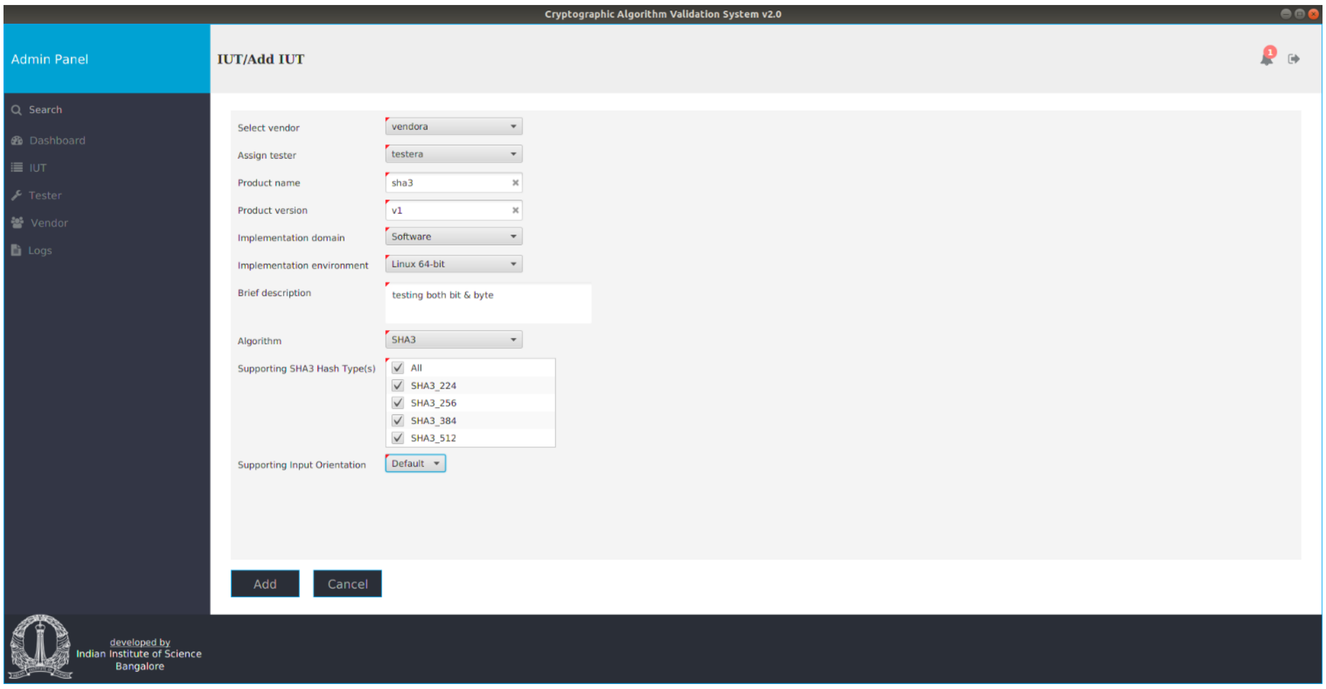
Input size	Output size	Type	Code Working
Any size	**224**	**Short**	**Tested OK**
Any size	**224**	**Long**	**Tested OK**
Any size	**256**	**Short**	**Tested OK**
Any size	**256**	**Long**	**Tested OK**

Any size	**384**	**Short**	**Tested OK**
Any size	**384**	**Long**	**Tested OK**
Any size	**512**	**Short**	**Tested OK**
Any size	**512**	**Long**	**Tested OK**

Test vector for BIT oriented implementation is given by NIST and recommended by FIPS.

It consists of these fields:

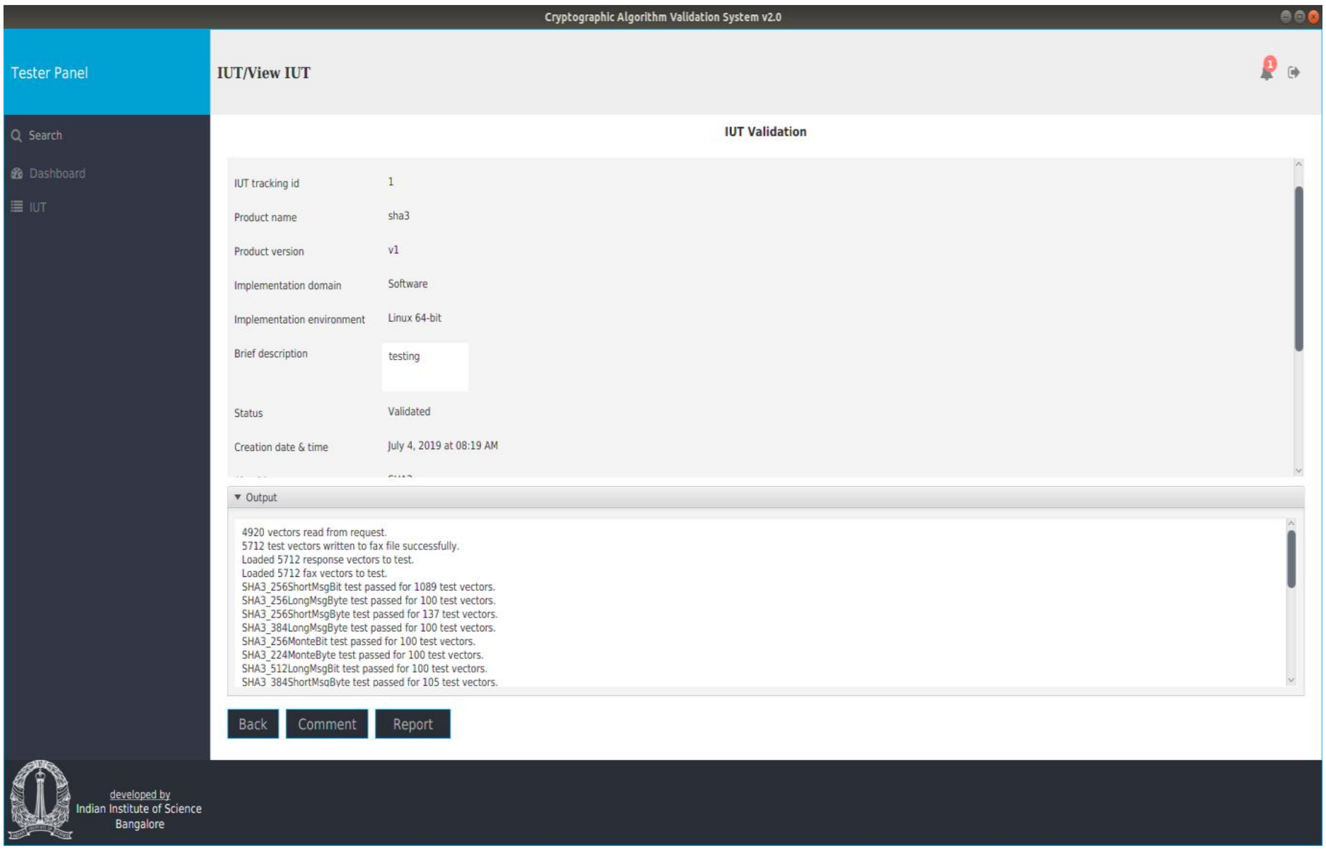
- Len: Length of the message in BITs.
- MSG: Hex input message.
- MD: Message Digest of using HASH algorithm.



IUT/Add IUT

Select vendor	vendora
Assign tester	testera
Product name	sha3
Product version	v1
Implementation domain	Software
Implementation environment	Linux 64-bit
Brief description	testing both bit & byte
Algorithm	SHA3
Supporting SHA3 Hash Type(s)	✓ All ✓ SHA3_224 ✓ SHA3_256 ✓ SHA3_384 ✓ SHA3_512
Supporting Input Orientation	Default

The above snapshots are from actually application running for complete CAVP application.



IUT tracking id	1
Product name	sha3
Product version	v1
Implementation domain	Software
Implementation environment	Linux 64-bit
Brief description	testing
Status	Validated
Creation date & time	July 4, 2019 at 08:19 AM

```
4920 vectors read from request.
5712 test vectors written to fax file successfully.
Loaded 5712 response vectors to test.
Loaded 5712 fax vectors to test.
SHA3_256ShortMsgBit test passed for 1089 test vectors.
SHA3_256LongMsgByte test passed for 100 test vectors.
SHA3_256ShortMsgByte test passed for 137 test vectors.
SHA3_384LongMsgByte test passed for 100 test vectors.
SHA3_256MonteBit test passed for 100 test vectors.
SHA3_224MonteByte test passed for 100 test vectors.
SHA3_512LongMsgBit test passed for 100 test vectors.
SHA3_384ShortMsgByte test passed for 105 test vectors.
```

2.6 Conclusion

After a very long and complex evaluation process, NIST has selected Keccak as the winner of the SHA-3 Competition. It was chosen from a field of five very strong candidates, each of which provides some outstanding performance characteristics and design features. All five finalists appear to provide an adequate security margin, as well.

Keccak has a large security margin, good general performance, excellent efficiency in hardware implementations, and a flexible design. Keccak uses a new “sponge construction” domain extender, which is based on a fixed permutation, that can be readily adjusted to trade generic security strength for throughput, and can generate larger or smaller hash outputs, as required. The Keccak designers have also defined a modified chaining mode for Keccak that provides authenticated encryption. NIST plans to augment the current hash standard, FIPS 180-4, to include the new SHA-3 algorithm, and publish a draft FIPS 180-5 for public review. After the close of the public comment period, NIST will revise the draft standard, as appropriate, in response to the public comments that NIST receives. A final review, approval, and promulgation process will then follow. Additionally, NIST may consider standardizing additional constructions based on the Keccak permutation, such as an authenticated-encryption mode, in the future. Any such future standardization efforts will be conducted in consultation with the public, the Keccak design team and the larger cryptographic research community.

3 . XTS-AES

3.1 Introduction

This standard defines the XTS-AES tweakable block cipher and its use for encryption of sector-based storage. XTS-AES is a tweakable block cipher that acts on data units of 128 bits or more and uses the AES block cipher as a subroutine. The key material for XTS-AES consists of a data encryption key (used by the AES block cipher) as well as a "tweak key" that is used to incorporate the logical position of the data block into the encryption. XTS-AES is a concrete instantiation of the class of tweakable block ciphers described in Rogaway. a the XTS-AES addresses threats such as copy-and-paste attack, while allowing parallelization and pipelining in cipher implementations.
Tweak value: The 128-bit value used to represent the logical position of the data being encrypted or decrypted with XTS-AES.

3.1.1 Data Units and Tweaks

The data unit size shall be at least 128 bits. Data unit should be divided into 128-bit blocks. Last part of the data unit might be shorter than 128 bits. The number of 128-bit blocks in the data unit shall not exceed 2128 – 2. The number of 128-bit blocks should not exceed 220. Each data unit is assigned a tweak value that is a non-negative integer. The tweak values are assigned consecutively, starting from an arbitrary nonnegative integer. When encrypting a tweak value using AES, the tweak is first converted into a little-endian byte array. For example, tweak value 123456789a16 corresponds to byte array 9a16,7816,5616,3416,1216.

3.1.2 Multiplication by a primitive element α

The encryption procedure (see 5.3) and decryption procedure (see 5.4) use multiplication of a 16-byte value (the result of AES encryption or decryption) by j-th power of α, a primitive element of GF(2128). The input value is first converted into a byte array a0[k], k = 0,1,...,15. In particular, the 16-byte result of AES encryption or decryption is treated as a byte array, where a0[0] is the first byte of the AES block.

This multiplication is defined by the following procedure:

Input: j is the power of α byte array a0[k], k = 0,1,...,15

Output: byte array aj[k], k = 0,1,...,15

The output array is defined recursively by the following formulas where i is iterated from 0 to j:

$a_{i+1}[0] \leftarrow (2\ (a_i[0] \bmod 128)) \oplus (135 \lfloor a_i[15]/128 \rfloor)$

$a_{i+1}[k] \leftarrow (2\ (a_i[k] \bmod 128)) \oplus \lfloor a_i[k-1]/128 \rfloor$, k = 1,2,…,15

3.1.3 Acronyms and abbreviations

The following symbols are used in equations and figures:

$\oplus$ Bit-wise exclusive-OR operation =

$\otimes$ Modular multiplication of two polynomials over the binary field GF(2), modulo x 128 + x 7 + x 2 + x + 1, where GF stands for Galois Field

α A primitive element of GF(2128) that corresponds to polynomial x (i.e., 0000…0102), where GF stands for Galois Field

• Assignment of a value to a variable

| Concatenation (e.g., if K1 = 0012 and K2 = 1010102, then K1|K2 = 0011010102)

// Start of a comment. Comment ends at end of line

$\lfloor x \rfloor$ Floor of x (e.g., $\lfloor 7/3 \rfloor$ = 2)

3.2 Types of XTS-AES

1) **Tweak value as hex string with**
   1. **AES218**
   2. **AES256**
2) **Tweak value as decimal value with**
   1. **AES128**
   2. **AES256**

3.3 Implementation

Encryption and Decryption algorithm are as follows:

3.3.1 XTS-AES encryption procedure

3.3.1.1 XTS-AES-blockEnc procedure, encryption of a single 128-bit block

The XTS-AES encryption procedure for a single 128-bit block is modeled with Equation (1).

C ← XTS-AES-blockEnc(Key, P, i, j)

where

Key is the 256 or 512bit XTS-AES key

P is a block of 128 bits (i.e., the plaintext)

i is the value of the 128-bit tweak (see 5.1)

j is the sequential number of the 128-bit block inside the data unit

C is the block of 128 bits of ciphertext resulting from the operation

The key is parsed as a concatenation of two fields of equal size called Key1 and Key2 such that:
Key = Key1 | Key2.

The ciphertext shall then be computed by the following or an equivalent sequence of steps:

1) $T \leftarrow$ AES-enc(Key2 , i) $\otimes$ αj

2) $PP \leftarrow P \oplus T$

3) $CC \leftarrow$ AES-enc(Key1 , PP)

4) $C \leftarrow CC \oplus T$

AES-enc(K,P) is the procedure of encrypting plaintext P using AES algorithm with key K, according to FIPS-197. The multiplication and computation of power in step 1) is executed in GF(2128), where α is the primitive element defined in 4.2 (see 5.2)

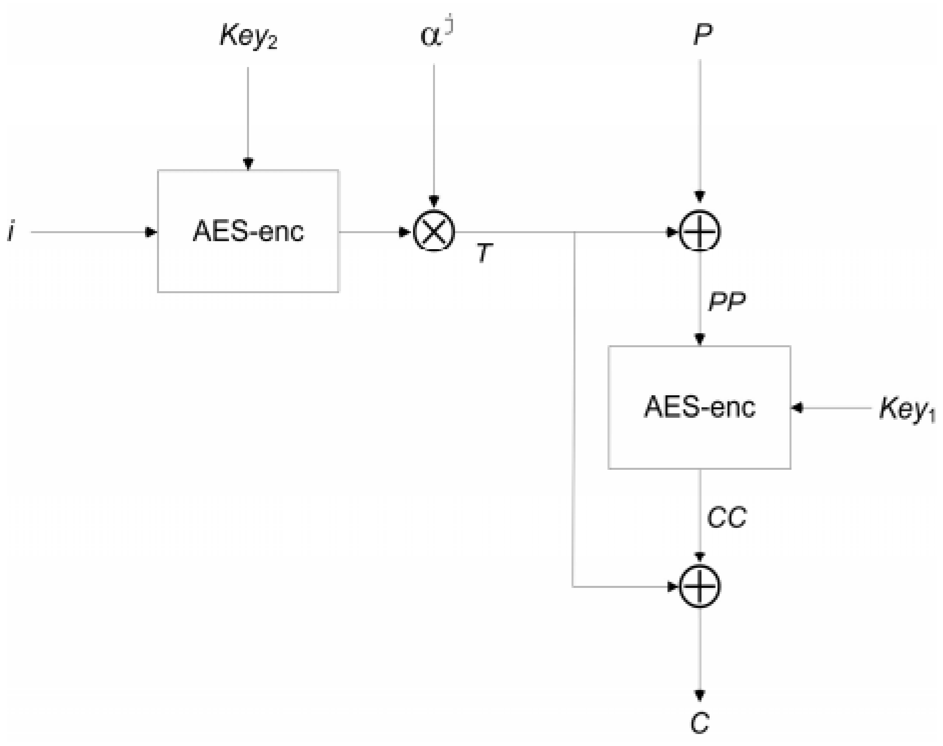


Figure 1— Diagram of XTS-AES blockEnc procedure

3.3.1.2 XTS-AES encryption of a data unit

The XTS-AES encryption procedure for a data unit of plaintext of 128 or more bits is modeled with Equation (2).

$C \leftarrow$ XTS-AES-Enc (Key, P, i)

where

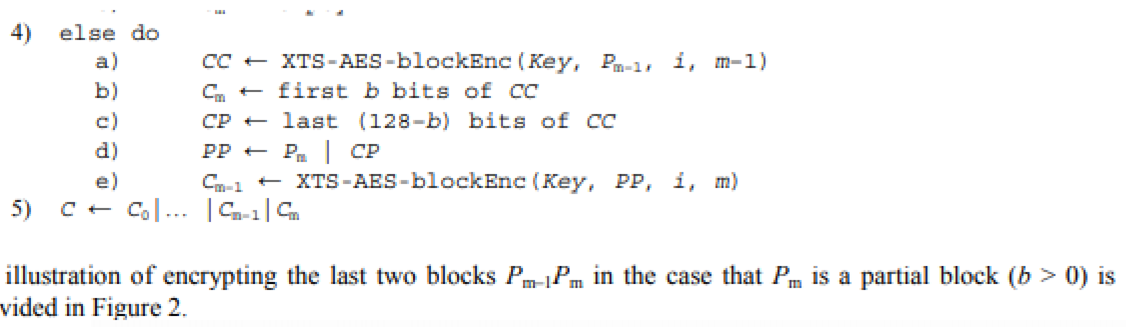
Key	is the 256 or 512bit XTS-AES key
P	is the plaintext
i	is the value of the 128-bit tweak (see 5.1)
C	is the ciphertext resulting from the operation, of the same bit-size as P

The plaintext data unit is first partitioned into m + 1 blocks, as follows:

$P = P0 |\dots |Pm-1|Pm$

where m is the largest integer such that 128m is no more than the bit-size of P, the first m blocks P0,…, Pm−1 are each exactly 128 bits long, and the last block Pm is between 0 and 127 bits long (Pm could be empty, i.e., 0 bits long). The key is parsed as a concatenation of two fields of equal size called Key1 and Key2 such that: Key = Key1 | Key2. The ciphertext C is then computed by the following or an equivalent sequence of steps:

```
1)  for q ← 0 to m-2 do
    a) Cq ← XTS-AES-blockEnc(Key, Pj, i, q)
2) b ← bit-size of Pm
3) if b = 0 then do
        a) Cm–1 ← XTS–AES-blockEnc(Key, Pm–1, i, m–1)
        b) Cm ← empty 4) else do a) CC ← XTS-AES-blockEnc(Key, Pm–1, i, m–1) b) Cm
```



```
4)  else do
      a)      CC ← XTS-AES-blockEnc(Key, $P_{m-1}$, i, m-1)
      b)      $C_m$ ← first b bits of CC
      c)      CP ← last (128-b) bits of CC
      d)      PP ← $P_m$ | CP
      e)      $C_{m-1}$ ← XTS-AES-blockEnc(Key, PP, i, m)
5)  C ← $C_0$|… |$C_{m-1}$|$C_m$
```

An illustration of encrypting the last two blocks $P_{m-1}P_m$ in the case that P_m is a partial block ($b > 0$) is provided in Figure 2.

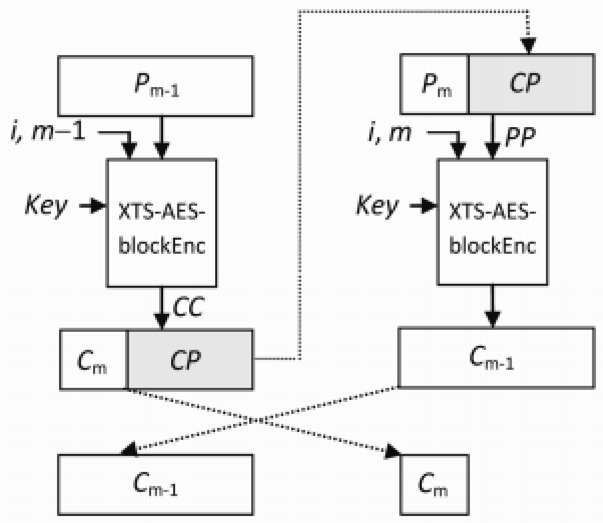


Figure 2—XTS-AES encryption of last two blocks when last block is 1 to 127 bits

3.3.2 XTS-AES decryption procedure

3.3.2.1 XTS-AES-blockDec procedure, decryption of a single 128-bit block

The XTS-AES decryption procedure of a single 128-bit block is modeled with Equation (3).

P ← XTS-AES-blockDec(Key, C, i, j)

where

Key	is the 256 or 512-bit XTS-AES key
C	the 128-bit block of ciphertext
i	is the value of the 128-bit tweak (see 5.1)
j	is the sequential number of the 128-bit block inside the data unit
P	is the 128-bit block of plaintext resulting from the operation

The key is parsed as a concatenation of two fields of equal size called Key_1 and Key_2 such that: $Key = Key_1 \mid Key_2$. The plaintext shall then be computed by the following or an equivalent sequence of steps (see Figure 3):

1) $T \leftarrow \text{AES-enc}(Key_2\ ,\ i)\ \otimes \alpha^j$
2) $CC \leftarrow C \oplus T$
3) $PP \leftarrow \text{AES-dec}(Key_1\ ,\ CC)$
4) $P \leftarrow PP \oplus T$

AES-dec(K,C) is the procedure of decrypting ciphertext C using AES algorithm with key K, according to FIPS-197. The multiplication and computation of power in step 1) is executed in GF(2^{128}), where α is the primitive element defined in 4.2 (see 5.2).

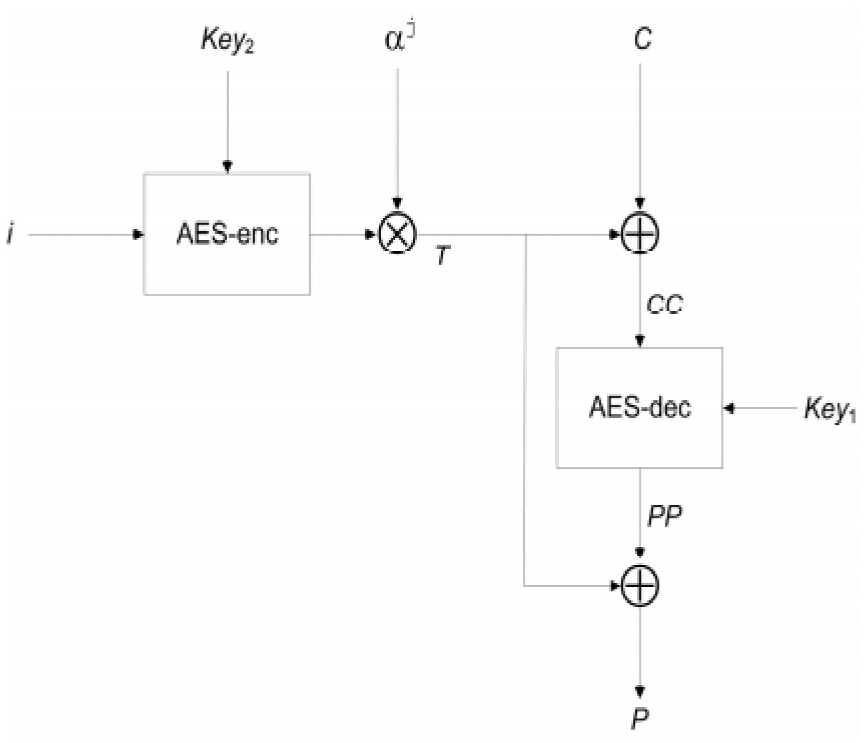


Figure 3—Diagram of XTS-AES blockDec procedure

3.3.2.2 XTS-AES decryption of a data unit

The XTS-AES decryption procedure for a data unit ciphertext of 128 or more bits is modeled with Equation (4).

$P \leftarrow$ XTS-AES-Dec(Key, C, i)

where

Key is the 256 or 512-bit XTS-AES key

C is the ciphertext corresponding to the data unit

i is the value of the 128-bit tweak

P is the plaintext data unit resulting from the operation, of the same bit-size as C

The ciphertext is first partitioned into m + 1 blocks as follows:

$C = C0 |\ldots |Cm-1|C$

where m is the largest integer such that 128m is no more than the bit-size of C, the first m blocks C0,…, Cm−1 are each exactly 128 bits long, and the last block Cm is between 0 and 127 bits long (Cm could be empty, i.e., 0 bits long). The key is parsed as a concatenation of two fields of equal size called Key1 and Key2 such that:

Key = Key1 | Key2.

The plaintext P is then computed by the following or an equivalent sequence of steps:

```
1)  for $q$ ← 0 to $m$-2 do
     a)      $P_q$ ← XTS-AES-blockDec($Key$, $C_j$, $i$, $q$)
2)  $b$ ← bit-size of $C_m$
3)  if $b$ = 0 then do
     b)      $P_{m-1}$ ← XTS-AES-blockDec($Key$, $C_{m-1}$, $i$, $m$-1)
     c)      $P_m$ ← empty
4)  else do
     d)      $PP$ ← XTS-AES-blockDec($Key$, $C_{m-1}$, $i$, $m$)
     e)      $P_m$ ← first b bits of $PP$
     f)      $CP$ ← last (128-b) bits of PP
     g)      $CC$ ← $C_m$ | $CP$
     h)      $P_{m-1}$ ← XTS-AES-blockDec($Key$, $CC$, $i$, $m$-1)
5)  $P$ ← $P_0$ |… | $P_{m-1}$ | $P_m$
```

The decryption of the last two blocks $C_{m-1}C_m$ in the case that C_m is a partial block ($b > 0$) is illustrated in Figure 4.

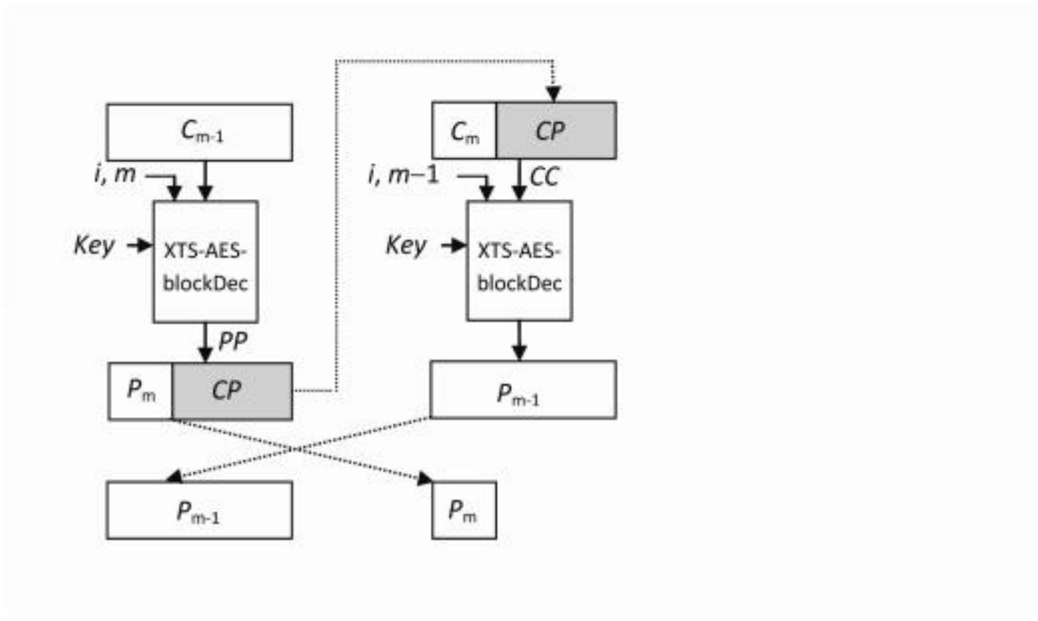
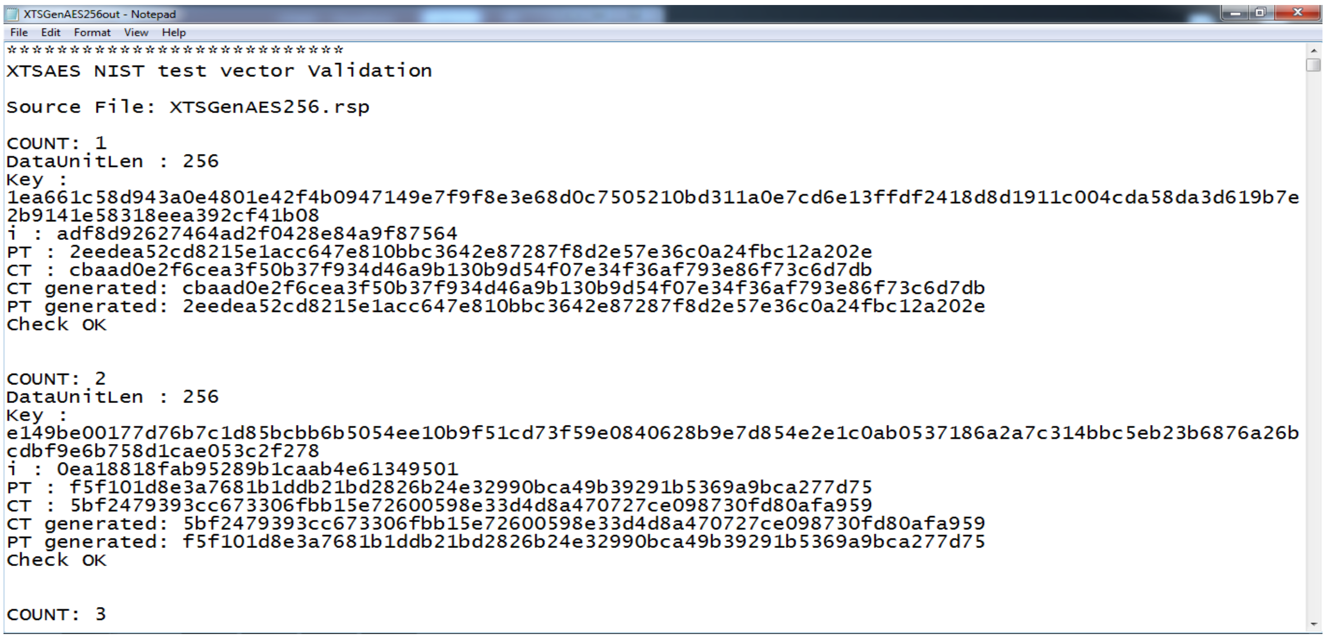


Figure 4—XTS-AES decryption of last two blocks when last block is 1 to 127 bits

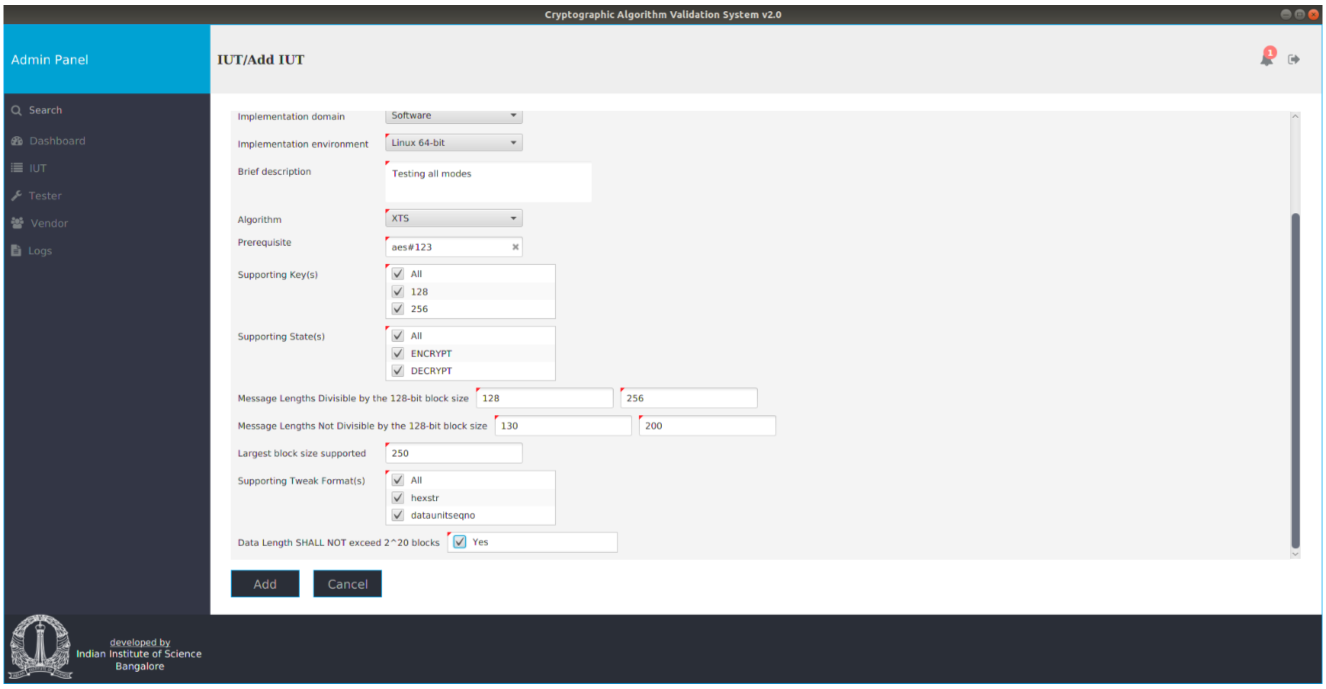
3.4 Results

Following is the screenshot of one of the many output file generated after validation of the NIST test vectors.



```
******************************
XTSAES NIST test vector Validation

Source File: XTSGenAES256.rsp

COUNT: 1
DataUnitLen : 256
Key :
1ea661c58d943a0e4801e42f4b0947149e7f9f8e3e68d0c7505210bd311a0e7cd6e13ffdf2418d8d1911c004cda58da3d619b7e
2b9141e58318eea392cf41b08
i : adf8d92627464ad2f0428e84a9f87564
PT : 2eedea52cd8215e1acc647e810bbc3642e87287f8d2e57e36c0a24fbc12a202e
CT : cbaad0e2f6cea3f50b37f934d46a9b130b9d54f07e34f36af793e86f73c6d7db
CT generated: cbaad0e2f6cea3f50b37f934d46a9b130b9d54f07e34f36af793e86f73c6d7db
PT generated: 2eedea52cd8215e1acc647e810bbc3642e87287f8d2e57e36c0a24fbc12a202e
Check OK


COUNT: 2
DataUnitLen : 256
Key :
e149be00177d76b7c1d85bcbb6b5054ee10b9f51cd73f59e0840628b9e7d854e2e1c0ab0537186a2a7c314bbc5eb23b6876a26b
cdbf9e6b758d1cae053c2f278
i : 0ea18818fab95289b1caab4e61349501
PT : f5f101d8e3a7681b1ddb21bd2826b24e32990bca49b39291b5369a9bca277d75
CT : 5bf2479393cc673306fbb15e72600598e33d4d8a470727ce098730fd80afa959
CT generated: 5bf2479393cc673306fbb15e72600598e33d4d8a470727ce098730fd80afa959
PT generated: f5f101d8e3a7681b1ddb21bd2826b24e32990bca49b39291b5369a9bca277d75
Check OK


COUNT: 3
```

The following snapshots are from actually application running for complete CAVP application.

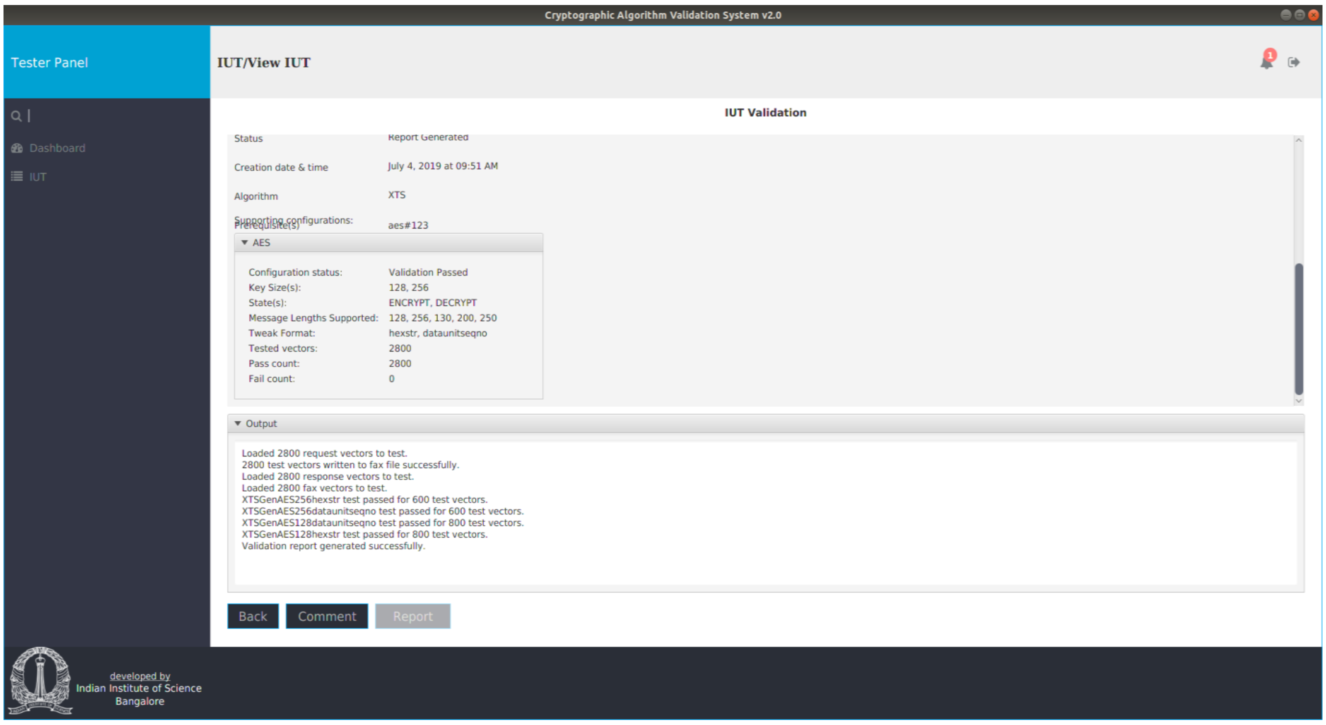


Implementation domain	Software
Implementation environment	Linux 64-bit
Brief description	Testing all modes
Algorithm	XTS
Prerequisite	aes#123
Supporting Key(s)	✓ All ✓ 128 ✓ 256
Supporting State(s)	✓ All ✓ ENCRYPT ✓ DECRYPT
Message Lengths Divisible by the 128-bit block size	128 / 256
Message Lengths Not Divisible by the 128-bit block size	130 / 200
Largest block size supported	250
Supporting Tweak Format(s)	✓ All ✓ hexstr ✓ dataunitseqno
Data Length SHALL NOT exceed 2^20 blocks	✓ Yes

According to the NIST test vector following are the parameters analyzed and tested on the implemented algorithms:

AES key size	Input Size	Tweak Hex String or Data Sequence Number
128	128, 256, 130, 200 and 356	Hex String
256	128, 256, 130, 200 and 356	Hex String
128	128, 256, 130, 200 and 356	Data Sequence Number
256	128, 256, 130, 200 and 356	Data Sequence Number

We have tested on all the above parameters specified.



Status	Report Generated
Creation date & time	July 4, 2019 at 09:51 AM
Algorithm	XTS
Supporting configurations:	
Prerequisite(s)	aes#123

AES

Configuration status:	Validation Passed
Key Size(s):	128, 256
State(s):	ENCRYPT, DECRYPT
Message Lengths Supported:	128, 256, 130, 200, 250
Tweak Format:	hexstr, dataunitseqno
Tested vectors:	2800
Pass count:	2800
Fail count:	0

Output

```
Loaded 2800 request vectors to test.
2800 test vectors written to fax file successfully.
Loaded 2800 response vectors to test.
Loaded 2800 fax vectors to test.
XTSGenAES256hexstr test passed for 600 test vectors.
XTSGenAES256dataunitseqno test passed for 600 test vectors.
XTSGenAES128dataunitseqno test passed for 800 test vectors.
XTSGenAES128hexstr test passed for 800 test vectors.
Validation report generated successfully.
```

3.5 CONCLUSION

In conclusion, while no encryption mode is perfect in all situations, there is compelling evidence that XTS-AES has strong industry support and provides good protection of stored data in situations where data expansion (for authentication codes) and performing two passes over the data (for wide-block encryption) is not possible. Overall, I recommend that NIST accept XTS-AES (as defined in IEEE Std 1619-2007) as an Approved Mode of Operation with the following changes:

- Allow 'one-key' variants of XTS-AES, where the same 128-bit or 256-bit key is used within both AES ciphers.
- Prohibit data unit sizes larger than 220 blocks.
- Clarify that the ordering of the blocks in ciphertext stealing is a logical ordering, and can be mapped to a different physical ordering (i.e., it is possible to swap the order of the ciphertext stealing blocks)

REFERENCES

[1] NIST FIPS 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

[2] Description of Known Answer Test (KAT) and Monte Carlo Test (MCT) for SHA-3 Candidate Algorithm Submissions.: Revision 3: February 20, 2008

[3] FIPS 180-3, Secure Hash Standard, Revision expected to be published in 2008.

[4] The Secure Hash Algorithm 3 Validation System (SHA3VS): April 7, 2016 Original: January 29, 2016.

[5] The XTS-AES Tweakable Block Cipher: An Extract from IEEE Std 1619-2007 Extracted from IEEE Std 1619-2007, published 18 April 2008.

[6] XTS: A Mode of AES for Encrypting Hard Disk

[7] Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices: NIST Special Publication 800-38E, January, 2010

[8] The XTS-AES Validation System (XTSVS): Updated: September 5, 2013, Previously Updated: