2019-06

RECONSTRUCTION OF SATELLITE DECRYPTION AND DATA HANDLING PROCESSES

Gilley, Joseph

Monterey, CA; Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

RECONSTRUCTION OF SATELLITE DECRYPTION AND DATA HANDLING PROCESSES

by

Joseph Gilley

June 2019

Thesis Advisor: James H. Newman
Second Reader: Giovanni Minelli

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2019	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE RECONSTRUCTION OF SATELLITE DECRYPTION AND DATA HANDLING PROCESSES	**5. FUNDING NUMBERS**
6. AUTHOR(S) Joseph Gilley	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

This study examines how to reconstruct a satellite decryption process from source information. It examines how cryptographic algorithms are implemented in software and what software components are required to access data in a useable format. This examination enabled the reverse engineering and reconstruction of the decoding processes utilized by the three PropCubes: Merryweather, Fauna, and Flora. Transmitted data from these PropCubes was analyzed to verify the validity of the developed decryption and data handling Python scripts. A concept of operations for implementing the reconstructed decryption and data handling processes in real-time is discussed in this research.

14. SUBJECT TERMS encryption, decryption, Advanced Encryption Standard, Galois/Counter Mode, data processing, satellite, CubeSat, PropCube, Merryweather, Flora, Fauna	**15. NUMBER OF PAGES** 85
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

RECONSTRUCTION OF SATELLITE DECRYPTION AND DATA HANDLING PROCESSES

Joseph Gilley
Lieutenant, United States Navy
BS, BA, University of Buffalo, 2012

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SPACE SYSTEMS OPERATIONS

from the

NAVAL POSTGRADUATE SCHOOL
June 2019

Approved by: James H. Newman
Advisor

Giovanni Minelli
Second Reader

James H. Newman
Chair, Department of Space Systems Academic Group

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This study examines how to reconstruct a satellite decryption process from source information. It examines how cryptographic algorithms are implemented in software and what software components are required to access data in a useable format. This examination enabled the reverse engineering and reconstruction of the decoding processes utilized by the three PropCubes: Merryweather, Fauna, and Flora. Transmitted data from these PropCubes was analyzed to verify the validity of the developed decryption and data handling Python scripts. A concept of operations for implementing the reconstructed decryption and data handling processes in real-time is discussed in this research.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. DESCRIPTION OF DATA SETS UTILIZED....2
1. Anatomy of a Message....2
2. NPS KISS Decoding Process....3
B. METHODOLOGY....5
C. THESIS OVERVIEW....6

II. RECONSTRUCTION OF PROPCUBE'S DECRYPTION PROCESS....7
A. ADVANCED ENCRYPTION STANDARD 128....7
1. AES Algorithm....10
2. Key Expansion....14
3. Decryption....16
B. GALOIS/COUNTER MODE....21
1. Authenticated Encryption....22
2. Authenticated Decryption....28
C. PYTHON CRYPTOGRAPHY LIBRARY....30

III. RECONSTRUCTION OF PROPCUBE'S DATA HANDLING PROCESS FOR INTERPRETATION OF DECRYPTED DATA....33
A. DATA EXPRESSION INTERFACE CONTROL DOCUMENT....33
B. DETERMINING MESSAGE DATA....35
C. INTERPRETING THE MESSAGE....38

IV. FUTURE WORK AND CONCLUSION....45
A. FUTURE WORK....45
1. Real-Time Concept of Operations....45
2. Reconstruct ARSFTP_DATA1 Messages....46
3. Uplink Data Handling and Encryption....46
B. CONCLUSION AND RECOMMENDATIONS....47

APPENDIX A. PROPCUBE DECRYPTOR PYTHON SCRIPT....51

APPENDIX B. DATA PARSE PYTHON CODE....57

LIST OF REFERENCES....65

INITIAL DISTRIBUTION LIST....67

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. Data Transport Protocol. Adapted from [3]. 3

Figure 2. Flora and Merryweather Message Data Format. Adapted from [3]. 3

Figure 3. Fauna Message Data Format. Adapted from [3]. 3

Figure 4. KISS Log File Example 5

Figure 5. Example 128-bit Data Block Treated as a 4 x 4 Matrix. Adapted from [8]. 8

Figure 6. Multiplication Algorithm. Source: [8]. 9

Figure 7. AES-128 Encryption Algorithm. Source: [8]. 11

Figure 8. S-box: Substitution Values for the xy Byte in Hexadecimal Format. Source: [7]. 12

Figure 9. ByteSub Layer. Adapted from [7], [8]. 12

Figure 10. ShiftRow Layer. Source: [7]. 13

Figure 11. Matrix Multiplication for MixColumn Layer. Adapted from [7], [8]. 13

Figure 12. AddRoundKey Layer. Adapted from [8]. 14

Figure 13. Key Matrix 15

Figure 14. Recursive Equations For Round Key Generation, where $\nmid$ means does not divide and | means divides. Adapted from [8]. 15

Figure 15. Example of $T(W(i-1))$ Transformation. Adapted from [8]. 16

Figure 16. AES-128 Decryption Algorithm. Adapted from [8]. 17

Figure 17. Inverse S-box: Substitution Values for the xy Byte in Hexadecimal Format. Source: [7]. 18

Figure 18. Matrix Multiplication for InvMixColumn Layer. Adapted from [7], [8]. 18

Figure 19. Linear InvMixColumns Result. Adapted from [8]. 20

Figure 20. GCM Authenticated Encryption on a PropCube Packet. Adapted from [3], [9]. 23

Figure 21. Diagram of Authenticated Encryption. Source: [9]. 25

Figure 22. Definition of X_i for the GHASH Operation. Source: [9]. 28

Figure 23. Diagram of Authenticated Decryption. Source: [9] 29

Figure 24. Summary of Decryption Operations. Adapted from [9]. 30

Figure 25. Example Decryption Code. Source: [13]. 31

Figure 26. Example of a Decrypted Flora Ciphertext Produced by "propcube_decrypt.py" Script. 32

Figure 27. ARSFTP_METADATA Message Format. Adapted from [3], [4]. 34

Figure 28. IP Header Format. Source: [14]. 36

Figure 29. Logic Flow Diagram for "data_parse.py" Script 38

Figure 30. ARSFTP_DATA1 Message Format. Adapted from [3], [4]. 39

Figure 31. Python Code for Conversions From "data_parse.py" 40

Figure 32. SM_STATUS_PART1 Message Format. Adapted from [3], [4]. 41

Figure 33. Plaintext From Flora Interpreted by "data_parse.py" Script 42

Figure 34. Command to Utilize Reconstructed Processes 43

Figure 35. Example AES-128 in GCM Encryption Code. Source: [13]. 47

Figure 36. Example AES-128 in CTR Encryption Code. Source: [13]. 49

LIST OF TABLES

Table 1. KISS Special Character List. Adapted from [6]. 4

Table 2. AES-128 Encryption Algorithm. Adapted from [8]. 19

Table 3. AES-128 Decryption Algorithm. Adapted from [8]. 19

Table 4. Modified AES-128 Decryption Algorithm. Adapted from [8]. 19

Table 5. Further Modified AES-128 Decryption Algorithm. Adapted from [8]. 20

Table 6. Final AES-128 Decryption Algorithm. Adapted from [8]. 21

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AAD	Additional authenticated data
AEAD	Authenticated Encryption with Authenticated Data
AES	Advanced Encryption Standard
ARSFTP	Amateur Radio Satellite File Transport Protocol
CSV	comma-separated values
CTR	Counter Mode
CubeSat	cube satellite
FEND	Frame End
FESC	Frame Escape
GCM	Galois/Counter Mode
ICD	Interface Control Document
IV	Initialization Vector
IP	Internet Protocol
KISS	"Keep it simple, stupid" communication protocol
MAC	Message Authentication Code
MC3	Mobile CubeSat Command and Control
NIST	National Institute of Standards and Technology
NPS	Naval Postgraduate School
PropCube	Picosats Realizing Orbital Propagation Calibrations using Beacon Emitters
TFEND	Transposed Frame End
TFESC	Transposed Frame Escape
UDP	User Datagram Protocol

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. James Newman, and my second reader, Dr. Giovanni Minelli, for their patience and support of this thesis. Without their guidance and knowledge I would not have been able to complete this work.

I would also like to thank Lara Magallanes and Noah Weitz for letting me continuously disrupt their workday with my questions about PropCube operations.

Thank you to Jim Horning for his help in creating the Python scripts. Without the help of Jim I would still be looking at a computer screen.

Finally, a giant debt of gratitude is owed to my husband, Carlos Guash Jr., for his inexhaustible patience and support while I did my research.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

The Space Systems Academic Group (SSAG) at the Naval Postgraduate School (NPS) operates three satellites called Picosats Realizing Orbital Propagation Calibrations Using Beacon Emitters (PropCube) [1], [2]. The cube satellites (CubeSats), Merryweather, Flora, and Fauna, were designed by Tyvak and are operated by the Mobile CubeSat Command and Control (MC3) network at NPS [1]–[3]. The NPS node originates commands for the spacecraft and processes data received from PropCube utilizing Tyvak-created source code. Formerly, the source code that enabled the ground site to decrypt and interpret the received data was unavailable to NPS operators; therefore, these processes were carried out by software that was not explicitly known. Without explicit knowledge of the decryption process and data handling process, the SSAG is not able to customize processes to optimize operations.

Upon acquisition of PropCube, Tyvak provided documentation that outlined specific, standardized data protocols that are utilized for data handling and security [3], [4]. These processes are primarily standardized protocols that are available to the public at large. Through examining these standards, this thesis reconstructs the Advanced Encryption Standard 128 (AES-128) decryption and data handling processes of PropCube and identifies the key information required to do so.

The reconstruction of the decryption process and data handling process are critical for customizing operations. PropCube transmits data in a time constrained, noisy, radio-frequency environment at 914 MHz [2]. Sporadic radio-frequency noise routinely increases the link bit error rate during PropCube's transmissions, which reduces the probability the ground station will receive the transmitted data [2]. The Tyvak-created processes enable access to downlinked data after a PropCube pass has occurred. To ensure the desired data reaches the MC3 ground station, multiple requests for the same data are made by the spacecraft operators, which increase the probability of successful reception of the desired data. In recreating the decryption and data handling processes, the SSAG can potentially alter the processes to enable real-time access to PropCube data being transmitted. Such modifications could increase efficiency in operations by reducing the number of requests

made for specific data. Real-time access to data could give operators actionable information to enable improved operations. In knowing what data is being received as it is received, flexible decisions can be made by the operator or using automated scripts during a single pass to improve data reception.

A. DESCRIPTION OF DATA SETS UTILIZED

The methodology of this work draws on previous work conducted by students and members of the SSAG [2], [5]. This previous work created data sets that were utilized in reconstructing both the decryption process and the data handling process. These data sets provided a means for validation of the two developed processes, enabling this research.

1. Anatomy of a Message

To decrypt and interpret received messages, the structure of the message must be understood. The structure of the message dictates everything from what system is supposed to receive the message to how data is supposed to be accessed by the receiving system. In the Tyvak UHF Space-to-Ground Interface Control Document (ICD) [3], the structure of a message transport protocol is explicitly delineated. The standard AX.25 Amateur Packet-radio Link Layer Protocol is utilized with a unique fixed callsign assigned to each spacecraft [2], [3], [5]. The standard Internet Protocol (IP) header and User Datagram Protocol (UDP) are used. Each spacecraft is assigned its own unique, static IP address [3]. The IP, UDP, and message data are sent encrypted utilizing AES-128 [3]. Utilizing the initialization vector (IV) sent as the first 12 bytes of the AX.25 frame, as depicted in Figure 1, and the Tyvak-provided keys, it is possible to decrypt the received data. Reconstruction of this decryption process will be discussed in Chapter II of this thesis.

The message data format, shown in Figures 2 and 3, depends on the spacecraft and the specific message being received. Flora and Merryweather have an identical format, while Fauna, having been launched two years after the other two spacecraft, has a slightly modified message data format [2], [3]. The message identification indicates what type of message is being received by the ground station from the spacecraft. Utilizing the Data Expression Space-to-Ground ICD [4], it is possible to parse the data appropriately so that it can be interpreted.

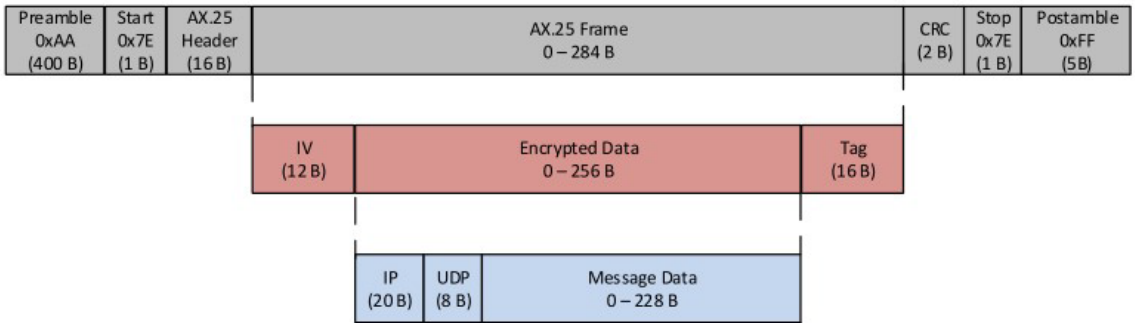

Figure 1. Data Transport Protocol. Adapted from [3].

Figure 2. Flora and Merryweather Message Data Format. Adapted from [3].

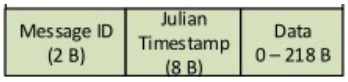

Figure 3. Fauna Message Data Format. Adapted from [3].

2. NPS KISS Decoding Process

Jan Roehrig outlines the AX.25 protocol and its implementation for MC3 and PropCube in his thesis [5]. Each message transmitted by the spacecraft begins with a preamble of alternating ones and zeros, as shown in Figure 1, so that the ground receiver and satellite may synchronize [2], [3], [5]. The following Start byte, 0x 7E, seen in Figure 1, announces the beginning of the data transmission. This same flag, 0x 7E, also denotes the end of each data transmission for that set of packets [2], [3], [5]. Appended to the transmitted data is a 16-bit cyclic redundancy check (CRC) value that is utilized for error detection [2], [3], [5]. Once the ground station receives the data, this CRC value is calculated based on the data downlinked [2], [5]. The ground station's calculated value is checked against the transmitted CRC value. If these values match, then it is likely the data received is identical to the data transmitted [2], [5]. This means the data can be successfully

decoded, and it is logged into a “Keep it simple, stupid” (KISS) packet [2], [5]. KISS packets include the AX.25 Header and AX.25 Frame data from Figure 1.

In KISS mode, at both the beginning and end, each packet is framed by 0x C0, the Frame End (FEND) special character byte to delimit the transmitted data [5], [6]. If the data in the packet contains the FEND special character, it is replaced by a two-byte special-character sequence, 0x DB, the Frame Escape (FESC) and, 0x DC, the Transposed Frame End (TFEND) [5], [6]. Additionally, if the FESC character is contained in the packet’s data, it is replaced by a two-byte special-character sequence of FESC and, 0x DD, the Transposed Frame Escape (TFESC) [5], [6]. The special-characters and their hex representation are in Table 1.

Table 1. KISS Special Character List. Adapted from [6].

Special Character	Description	Hex Representation
FEND	Frame End	C0
FESC	Frame Escape	DB
TFEND	Transposed Frame End	DC
TFESC	Transposed Frame Escape	DD

Figure 4 is an example of a KISS log file from Flora. It contains multiple KISS packets, each of which contain their own AX.25 Header, which is sent unencrypted, and AX.25 Frame, which, with the exception of the IV and authentication tag, is sent encrypted. In Figure 4, the bytes underlined in red are inserted FEND characters in adherence to the KISS protocol, the bytes underlined in green comprise the AX.25 header, the bytes underlined in purple comprise the IV, and the bytes underlined in blue comprise the authentication tag. The bytes not underlined are encrypted data. The bytes boxed in red is an example of when the FEND character is replaced by a FESC, TFEND sequence, while the bytes boxed in green are examples of when the FESC character is replaced by a FESC, TFESC sequence.

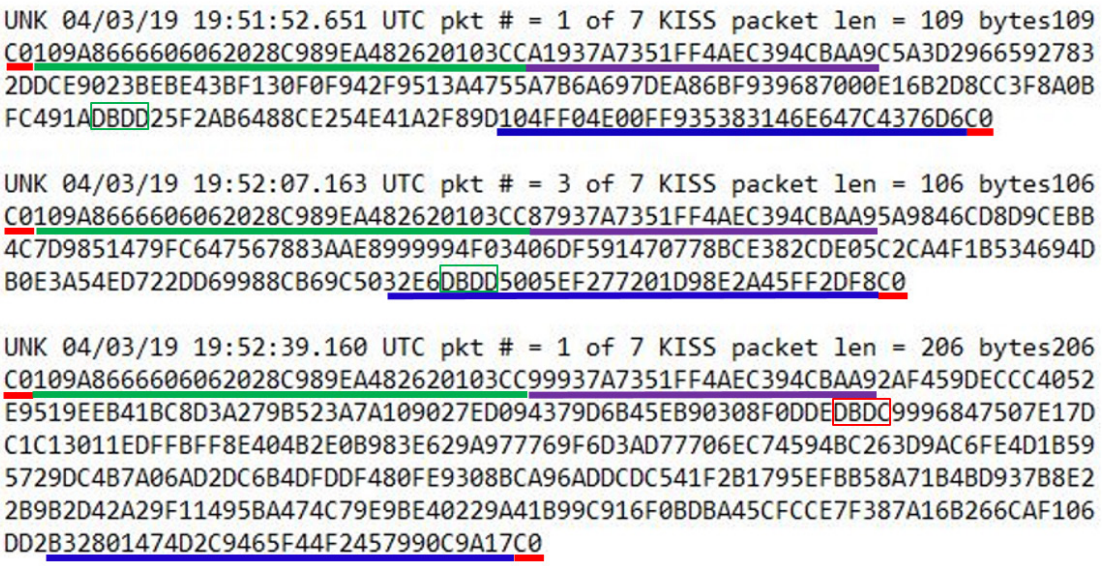

Figure 4. KISS Log File Example

The KISS data packets must be unframed prior to decryption. That is, the beginning and end C0 bytes must be removed, any sequenced DB DC bytes in the data must be replaced by C0, and any sequenced DB DD bytes in the data must be replaced by DB [5], [6]. The process of unframing packets is handled in the "propcube_decrypt.py" script, as seen in Appendix A of this thesis, enabling the decryption process to act only on the appropriate data.

B. METHODOLOGY

This thesis first reconstructs PropCube's decryption process, which enables access to transmitted data. It then reconstructs the data handling process, which enables the decrypted data to be interpreted and displayed in a human readable format. To verify the reconstructed processes, this thesis conducts selected analysis of the current KISS log file data set. KISS files that had accessible corresponding decrypted and interpreted data were selected to test the processes developed in this thesis. The validity of the two reconstructed processes was verified through a comparison between the data sets. Through an examination of the data set and provided spacecraft documentation, a trial-and-error approach was utilized to reconstruct PropCube's decryption and data handling processes.

Tyvak's provided executable software processes log data to .pcap files maintained by the ground station. These files contain network packet data. This data is decrypted, but not yet interpreted. These files allowed for the created decryption process to be validated prior to developing the data handling process. The Tyvak software processes decrypted data, which is then interpreted and maintained in separate log files. These log files are also accessible to the ground station. These files allowed for the created data interpretation process to be validated.

C. THESIS OVERVIEW

This thesis documents the information necessary to successfully reconstruct PropCube's decryption and data interpretation processes.

Chapter II reconstructs PropCube's decryption process. PropCube utilizes the AES-128 algorithm and its employment of the Galois/Counter Mode (GCM) of operation to provide confidentiality and authentication of transmitted data [3]. The operations utilized in the algorithms are explained, and the inputs necessary from a KISS packet are identified. Chapter II discusses how the "propcube_decrypt.py" script, found in Appendix A of this thesis, was developed and verified.

Chapter III discusses the reconstruction of the data handling process for interpretation of decrypted data. It discusses the development of the "data_parse.py" script, found in Appendix B. In parsing the decrypted data as described in the Data Expression ICD, some, but not all, received data packets are able to be accessed and interpreted. This chapter outlines which messages are able to be reconstructed, which are not, and why.

Finally, Chapter IV outlines future work that could build on this project. Chapter IV also summarizes the results of this thesis, reiterating the key information required to reconstruct PropCube's decryption process and enable the ground station to interpret the downlinked data.

II. RECONSTRUCTION OF PROPCUBE'S DECRYPTION PROCESS

Utilizing the KISS log files generated by the Mobile CubeSat Command and Control (MC3) network as test data, this thesis reconstructs and outlines how PropCube implements the Advance Encryption Standard (AES) 128 in Galois/Counter Mode (GCM). This chapter provides an overview of the cryptographic and authentication functions used by PropCube, a knowledge of which is necessary to reconstruct the decryption process. With an understanding of the AES-128 in GCM cryptosystem, open source Python libraries are leveraged in the construction of the "propcube_decrypt.py" script shown in Appendix A. The Python libraries provide a simple and secure implementation of the cryptosystem, including the decryption process. The development of this decryption script was a critical step to enabling the interpretation of transmitted PropCube data.

A. ADVANCED ENCRYPTION STANDARD 128

To enable decryption, the AES algorithm must be understood. The National Institute of Standards and Technology (NIST) adapted the Rijndael algorithm, created by Joan Daemen and Vincent Rijmen, as AES [7], [8]. The NIST document "Announcing the Advanced Encryption Standard (AES)" outlines, with examples, the adapted algorithm [7]. AES is a *symmetric key* algorithm, meaning both communicating parties know the secret cipher key [8]. AES can be utilized with cipher keys of one of three lengths; 128, 192 or 256 bits [7]. PropCube uses a 128-bit secret cipher key, referred to as AES-128, which provides data confidentiality [3], [7], [8]. This work will refer to the Tyvak-provide secret cipher key as the *original key*. Data confidentiality provides a means to stop third party actors from being able to read transmitted data [8].

AES is a *block cipher* which breaks the plaintext into blocks of fixed length and encrypts each block individually [8]. AES-128 acts on a fixed 128-bit block length [7], [8]. The specific 128-bit block that AES-128 acts on depends on the mode of operation. Since PropCube utilizes GCM as its mode of operation, AES-128 is applied to a unique 128-bit counter block [9]. So it is the counter block that is being encrypted by AES-128, not a

plaintext block. The encrypted counter block is then combined with the plaintext block to produce the ciphertext [9]. By encrypting the unique counter block, rather than the plaintext, a unique ciphertext is always produced, even when the same message data is repeatedly transmitted [8]. This protects the data's confidentiality against chosen plaintext attacks that could otherwise be used by a third party to gain access to encrypted data [8]. How counter blocks are determined and GCM is employed by PropCube is described in depth in Section B of this chapter.

The 128 input bits are treated as 16 bytes, which are arranged into a 4 x 4 matrix [8]. The first column of the matrix is the first four bytes of the block, the second column is the next four bytes, and so on [8]. For example, if the 128 bits were grouped into the 16 bytes, $b_0 b_1 b_2 b_3 \ldots b_{12} b_{13} b_{14} b_{15}$, then the array, is as presented in Figure 5.

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$

Figure 5. Example 128-bit Data Block Treated as a 4 x 4 Matrix. Adapted from [8].

We count the rows from top to bottom as follows: 0, 1, 2, and 3; similarly the columns from left to right are 0, 1, 2, and 3.

The operations performed on our elements by the AES algorithm are performed over the established field [7], [8]. As described in "Announcing the Advanced Encryption Standard (AES)," each byte in a matrix is treated as an element of the finite field $\mathbb{F}\left(2^8\right)$. The irreducible polynomial $x^8 + x^4 + x^3 + x^1 + 1$ is used to construct the finite field in AES [7], [8]. For more background information on finite fields, the curious reader should see Chapter 3 Section 11 in reference [8]. In $\mathbb{F}\left(2^8\right)$, each byte represents a unique polynomial of the form $b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x^1 + b_0$, where the

coefficient b_n is the bit in n^{th} place of the byte [8]. For example, 10101010 represents the polynomial $x^7 + x^5 + x^3 + x^1$.

In this finite field, a bitwise XOR function, also denoted as $\oplus$, is utilized on the elements as the addition operation [8]. When matrices are added, the XOR operation is performed on bytes of corresponding entries [8]. When matrices are multiplied, polynomial multiplication modulo $x^8 + x^4 + x^3 + x^1 + 1$ is performed on the elements in the matrices [8]. Two polynomials are multiplied, and the product is divided by the base polynomial; the remainder is the polynomial element desired. For example:

$$(x^7 + x^5 + x^3 + x^1)(x^2) = x^9 + x^7 + x^5 + x^3$$
$$\equiv x^7 + x^4 + x^3 + x^2 + x^1 (\text{mod}\, x^8 + x^4 + x^3 + x^1 + 1)$$

First, the polynomials are multiplied. Since the product is of degree 9, it is larger than the base polynomial, so it must be divided by the base polynomial. By performing standard polynomial division and dividing $x^9 + x^7 + x^5 + x^3$ by $x^8 + x^4 + x^3 + x^1 + 1$ the result would be $x^7 - x^4 + x^3 - x^2 - x^1$. However, this operation is being performed in a binary field, which means a coefficient of -1 is the same as the coefficient 1 since $-1 \equiv 1 (\text{mod}\, 2)$. This results in the $x^7 + x^4 + x^3 + x^2 + x^1 (\text{mod}\, x^8 + x^4 + x^3 + x^1 + 1)$ congruency.

In general, this polynomial multiplication can be performed on bits by considering multiplication by the polynomial x^1 [8]. Let $p(x)$ be the polynomial element contained in $\mathbb{F}(2^8)$ on which we are performing multiplication, and let $p(x)$ be in bit representation. The algorithm, as seen in Figure 6, can be utilized to multiply by x^1.

1. Shift left and append a 0 as the last bit.
2. If the first bit is 0, stop.
3. If the first bit is 1, XOR with 100011011.

Figure 6. Multiplication Algorithm. Source: [8].

If the first bit is 0, as in step two, the polynomial is less than degree eight after being multiplied by x^1 so there is no need to reduce the product [8]. To multiply by higher powers than x^1, as in the example above, the polynomial is multiplied by x^1 multiple times [8]. To apply the algorithm to the previous example, let $p(x) = x^7 + x^5 + x^3 + x^1$. This polynomial is represented in bits as 10101010. Since $p(x)$ is being multiplied by x^2 it is appended with two 0s rather than one, thus $10101010 \rightarrow 1010101000$. This begins with 1, so step three is followed as seen below:

$$\begin{array}{r} 1010101000 \\ \oplus\ 100011011 \\ \hline 010011110 \end{array}$$

This yields a leading 0, so step two is followed and the algorithm stops. If there was a leading 1 a second step three from the algorithm would have been performed on the result of the first XOR with the trailing 0 brought down [8]. There are at most as many XOR operations performed as the degree of x being multiplied [8]. So in this example there could have been at most two XOR operations performed. Had $p(x)$ been multiplied by x^3, there could have been at most three XOR operations performed, and so on [8]. The resulting polynomial 010011110 is equivalent to $x^7 + x^4 + x^3 + x^2 + x^1$, which matches the answer given initially. This algorithm is further extendable as stated in reference [8], "multiplication by an arbitrary polynomial can be accomplished by multiplying by the various powers of X appearing in that polynomial, then adding (i.e. XORing) the results".

1. AES Algorithm

AES-128 uses the original key and 128-bit data blocks as inputs to generate a ciphertext output [7]. AES-128 utilizes the operations outlined in the previous section as building blocks for encryption and decryption. AES-128 encryption/decryption consists of ten rounds performed upon each 128-bit block of data [8]. Each round of encryption utilizes four transformations called layers [8]. These layers are the ByteSub transformation, the ShiftRow transformation, the MixColumn transformation, and the AddRoundKey transformation. Utilizing these four layers, the encryption algorithm is shown in Figure 7,

where the $W(i)$ for $i = 0,1,2,...,43$ comprise the round keys utilized in the AddRoundKey layer. The $W(i)$, referred to as words, are defined in Section A.2 of this chapter.

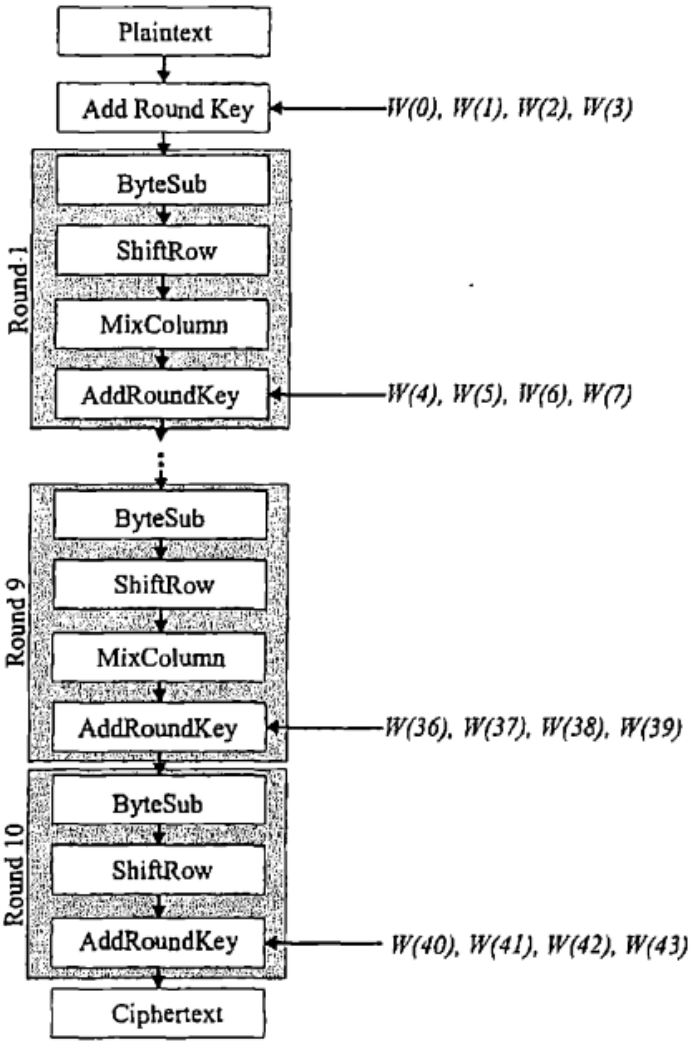

Figure 7. AES-128 Encryption Algorithm. Source: [8].

a. The ByteSub Transformation

This transformation performs a non-linear byte substitution utilizing a substitution table called an *S-box* [7]. This operation acts on each of the 16 bytes independently [7]. This layer, implemented with a lookup table for performance, is non-linear to protect the

cryptosystem from differential and linear cryptanalysis attacks [8]. For example, the hexadecimal byte 7D is substituted with the hexadecimal byte FF, circled in Figure 8.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 8. S-box: Substitution Values for the xy Byte in Hexadecimal Format. Source: [7].

A new 4 x 4 matrix is the output after the ByteSub Layer is applied to the data block matrix from Figure 5. The new 4 x 4 matrix is the right matrix shown in Figure 9.

$$\begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix} \rightarrow \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix}$$

Figure 9. ByteSub Layer. Adapted from [7], [8].

The interested reader can read the mathematical description used to construct the S-box in Chapter 5 Section 2 of reference [8].

b. The ShiftRow Transformation

As seen in Figure 10, this transformation cyclically shifts the bytes in the rows left [7]. Each row shifts its row number of bytes, so that row 0 does not shift, row 1 shifts each byte left one, and so on [8]. This layer causes data diffusion over multiple rounds [8].

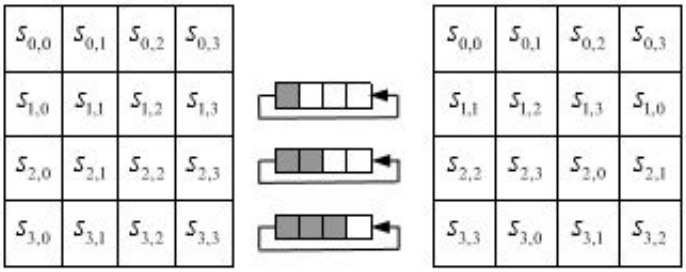

Figure 10. ShiftRow Layer. Source: [7].

c. The MixColumn Transformation

This transformation mixes the data of the columns to produce new columns [7]. Like ShiftRow, this layer also causes a diffusion of the data over multiple rounds [8]. This is accomplished with the following matrix multiplication, shown in Figure 11, where the left matrix is the MixColumn matrix and the right matrix is the state of the initial data block inputs after the ShiftRow transformation.

$$\begin{pmatrix} 00000010 & 00000011 & 00000001 & 00000001 \\ 00000001 & 00000010 & 00000011 & 00000001 \\ 00000001 & 00000001 & 00000010 & 00000011 \\ 00000011 & 00000001 & 00000001 & 00000010 \end{pmatrix} \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{pmatrix}$$

Figure 11. Matrix Multiplication for MixColumn Layer. Adapted from [7], [8].

The MixColumn matrix is invertible [8]. Its inverse matrix is utilized in decryption [8].

d. AddRoundKey

For each round of encryption/decryption a *round key* is used [8]. Since there are ten rounds of encryption, there are ten round keys, in addition to the original key [8]. The generation of the ten round keys from the original is referred to as *key expansion* [7]. The AddRoundKey transformation utilizes the XOR operation to apply a round key to the current state of the matrix [7]. In Figure 12, the matrix with entries s_i is the current state of matrix being encrypted, while the matrix with entries k_i is the round key being applied. How a round key is obtained is described in section A.2 of this chapter. These matrices are added by performing the XOR operation on corresponding bytes, $s_i \oplus k_i$, to produce the output byte [8]. This operation has a 0^{th} round, in addition to the ten rounds, which uses the original key [8].

$$\begin{pmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{pmatrix} \oplus \begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}$$

Figure 12. AddRoundKey Layer. Adapted from [8].

2. Key Expansion

In order to use the AES-128 algorithm, we must have a secret 128-bit original key known to both parties. From the original key, the ten round keys are recursively generated [7]. These round keys are then used during the AddRoundKey layer of the AES algorithm. Merryweather, Flora, and Fauna each have a distinct original key. Their keys are contained in an individual "AES GCM Keychain File" which was provided by Tyvak [3].

The original 128-bit key is grouped into 16 bytes, each designated k_n, and arranged into a 4 x 4 matrix in the same way as the data block [8]. The four byte columns, denoted

$W(i)$, are referred to as *words* [7], [8]. Let $k_0k_1k_2k_3...k_{12}k_{13}k_{14}k_{15}$ be some key. Then the key matrix is as seen in Figure 13.

$$\begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}$$

Figure 13. Key Matrix

Let $W(0)$ be the first column, $W(1)$ be the second column, $W(2)$ be the third column, and $W(3)$ be the forth column [8]. These four words are the original key, which is utilized during round 0 of the AES algorithm. The recursive equations, shown in Figure 14, are used to derive $W(4)$ through $W(43)$; the ten round keys, from the original four words.

$$W(i) = \begin{cases} W(i-4) \oplus W(i-1), if\ 4 \nmid i \\ W(i-4) \oplus T(W(i-1)), if\ 4 \mid i \end{cases}$$

Figure 14. Recursive Equations For Round Key Generation, where $\nmid$ means does not divide and | means divides. Adapted from [8].

In Figure 14, $\nmid$ means does not divide and | means divides. When four divides *i,* the operation $T(W(i-1))$ transforms $W(i-1)$, as seen in Figure 15, by first performing a cyclical shift up of the elements in the column vector, with the top entry of the vector becoming the bottom entry [7], [8]. Each byte is then substituted using the ByteSub Transformation described in the AES Algorithm section of this thesis [7], [8]. A round constant is then calculated by $r(i) = 00000010^{(i-4)/4}$, where 00000010 is a polynomial element of $\mathbb{F}(2^8)$ [8]. This round constant is added, as defined by the XOR operation,

with the 0 entry of the column vector [8]. A complete example of key expansion can be found in Appendix A of the NIST publication, which outlines AES implementation, reference [7].

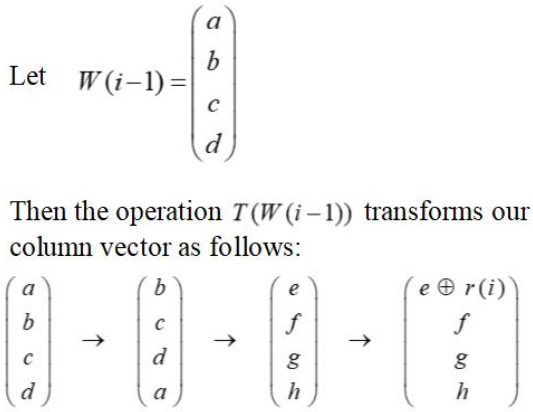

Figure 15. Example of $T(W(i-1))$ Transformation. Adapted from [8].

3. Decryption

Each of the layers has an inverse which allows the process to be undone to decrypt the block [7], [8]. These layers are the InvByteSub transformation, InvShiftRow transformation, InvMixColumn transformation, and AddRoundKey transformation. It is then useful to reorder the layers of the decryption process to mirror the layers of the encryption [8]. This reordering allows for the construction of a single piece of hardware that can carry out both the encryption and decryption process. PropCube uses a software driven solution for encryption and decryption, but the AES algorithm is designed to meet both software and hardware implementation requirements [7]. This reordering creates a new operation InvAddRoundKey [8]. Figure 16 shows the AES-128 decryption algorithm.

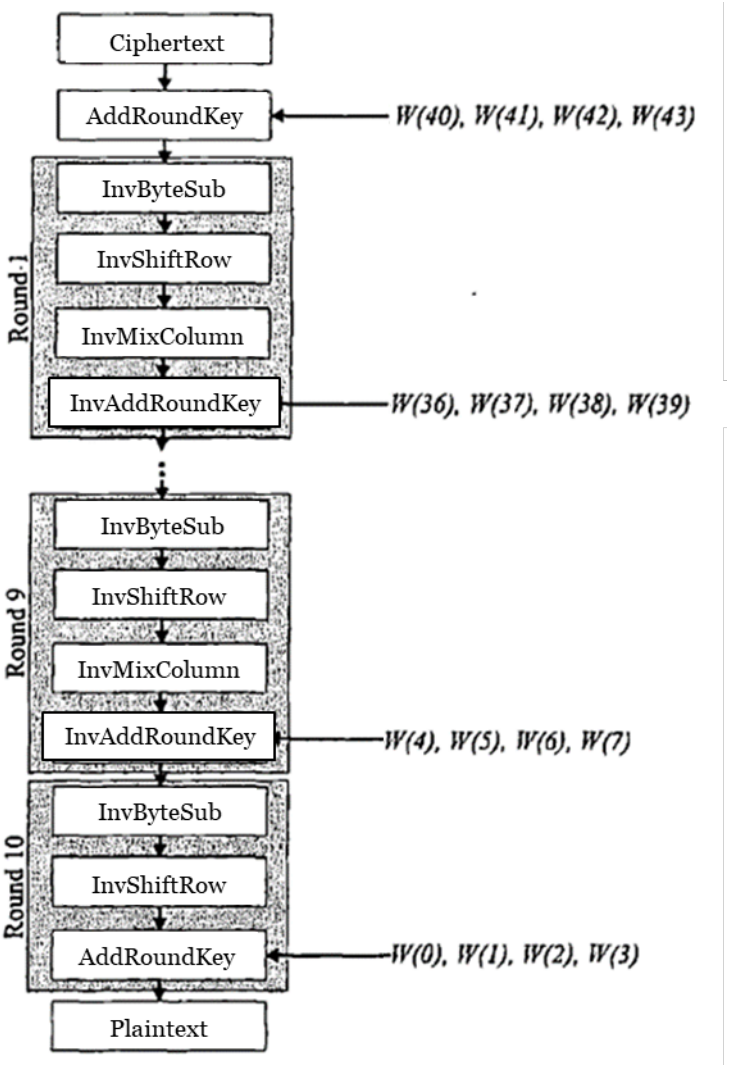

Figure 16. AES-128 Decryption Algorithm. Adapted from [8].

a. InvByteSub Transformation

This transformation utilizes a second look up table, called the *Inverse S-Box*, to undo the ByteSub layer [7]. This operations acts independently on bytes, just as the ByteSub layer does [7]. For example, the hexadecimal byte FF is substituted with the hexadecimal byte 7D, circled in Figure 17, which undoes the substitution example given previously.

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 17. Inverse S-box: Substitution Values for the xy Byte in Hexadecimal Format. Source: [7].

b. InvShiftRow Transformation

This transformation cyclically shifts the bytes in the rows by the corresponding row number just as ShiftRow does, but it reverses the shift direction [7]. Now the cyclical shift of bytes is to the right [7].

c. InvMixColumn Transformation

To undo the MixColumn layer, this operation uses the inverse of the MixColumn matrix [8]. The matrix multiplication is shown in Figure 18.

$$\begin{pmatrix} 00001110 & 00001011 & 00001101 & 00001001 \\ 00001001 & 00001110 & 00001011 & 00001101 \\ 00001101 & 00001001 & 00001110 & 00001011 \\ 00001011 & 00001101 & 00001001 & 00001110 \end{pmatrix} \begin{pmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{pmatrix}$$

Figure 18. Matrix Multiplication for InvMixColumn Layer. Adapted from [7], [8].

d. AddRoundKey Transformation

Because the AddRoundKey layer is the XOR operation on the entries of the matrices, it is its own inverse [7], [8]. We XOR the derived round keys in the reverse order, so the 0^{th} round of decryption uses the 10^{th} round key, the 1^{st} round of decryption uses the 9^{th} round key, and so on until the original key is used for the 10^{th} round of decryption [8].

To summarize, Table 2 shows the steps of AES encryption.

Table 2. AES-128 Encryption Algorithm. Adapted from [8].

Round 0	AddRoundKey
Round 1–9	ByteSub, ShiftRow, MixColumn, AddRoundKey
Round 10	ByteSub, ShiftRow, AddRoundKey

For decryption, we reverse these steps, which yields Table 3.

Table 3. AES-128 Decryption Algorithm. Adapted from [8].

Round 0	AddRoundKey, InvShiftRow, InvByteSub
Round 1–9	AddRoundKey, InvMixColumn, InvShiftRow, InvByteSub
Round 10	AddRoundKey

e. Reordering of Layers

We can rewrite the decryption process to look like the encryption process [8]. Because ByteSub acts on individual bytes one at a time, and ShiftRow simply permutes the bytes in our matrix, the order in which these two operations are performed does not impact the outcome; that is to say, the operations commute [7], [8]. Similarly, the InvByteSub and InvShiftRow layers commute [7], [8]. Reversing the order of the operations in the decryption process yields Table 4.

Table 4. Modified AES-128 Decryption Algorithm. Adapted from [8].

Round 0	AddRoundKey, InvByteSub, InvShiftRow
Round 1–9	AddRoundKey, InvMixColumn, InvByteSub, InvShiftRow
Round 10	AddRoundKey

The AddRoundKey and InvMixColumn layers do not commute, so we cannot simply reverse the order as we did with InvByteSub and InvShiftRow [8]. However, MixColumn and InvMixColumn are linear with respect to the individual bytes in our matrix [7], [8]. Figure 19 shows the result of carrying out the InvMixColumn linearly.

$$InvMixColumns\left(\begin{pmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{pmatrix} \oplus \begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}\right) = InvMixColumns \begin{pmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{pmatrix} \oplus InvMixColumns \begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}$$

Figure 19. Linear InvMixColumns Result. Adapted from [8].

In Figure 19, the matrix with entries s_i is the state of the matrix being decrypted, while the matrix with entries k_i is the round key being applied. So define InvAddRoundKey to be the transformation such that the round key is first transformed by the InvMixColumns layer then added with the XOR operation as shown in the last matrix of Figure 19 [7], [8]. Now InvMixColumns and InvAddRoundKey can commute and we see the algorithm as outlined in Table 5.

Table 5. Further Modified AES-128 Decryption Algorithm. Adapted from [8].

Round 0	AddRoundKey, InvByteSub, InvShiftRow
Round 1–9	InvMixColumn, InvAddRoundKey, InvByteSub, InvShiftRow
Round 10	AddRoundKey

To ensure the decryption process mirrors the encryption process the MixColumn layer is not applied in the final round of encryption [8]. If it had been applied, the decryption process would have begun with an InvMixColumn, which would slow down the algorithm without having added any utility to security [8]. Now that the operations have been reordered, a regrouping yields the final process as seen in Table 6.

Table 6. Final AES-128 Decryption Algorithm. Adapted from [8].

Round 0	AddRoundKey
Round 1–9	InvByteSub, InvShiftRow, InvMixColumn, InvAddRoundKey
Round 10	InvByteSub, InvShiftRow, AddRoundKey

B. GALOIS/COUNTER MODE

AES is designed to encrypt/decrypt 128-bit blocks [7], [8]. However, most of PropCube's messages require more that 128-bits be encrypted. When messages exceed 128-bits, meaning there is more than one block of data to encrypt, the mode of operation describes the algorithm used by AES to encrypt/decrypt the multiple blocks [8].

Galois/Counter Mode (GCM) is a block cipher mode, constructed for a 128-bit block size, which is able to provide authenticated encryption with associated data (AEAD) [9], [10]. AES provides confidentiality, the protection of information from a third party [8]; GCM brings authentication to our cryptosystem [9]. Authentication ensures the data received could have only been sent from the expected second party [8]. Authenticity of the data is established by using a universal hash function [10]. Additional authenticated data (AAD) is data sent unencrypted and is used as an input, along with the original key, the IV, and ciphertexts, by the universal hash function in creating the authentication tag [2], [10]. Generally, a hash function utilizes an arbitrary length input string and produces a fixed length output string [8]. A hash function should also have the following properties: the output can be calculated quickly, the function should be one-way or pre-image resistant, and it is computationally infeasible to find two different messages that result in the same output [8]. The GHASH function is the universal hash function used by

GCM and is described in the Authentication Tag portion of this section [9], [10]. The 16-byte tag seen in Figure 1 is the output of this hash function.

This authentication can prevent malicious actors from harming operations [11]. If a third party were to send a command to remove files from the spacecraft, this authentication process would allow the spacecraft to detect that the command was not sent from a trusted party [11]. The authentication tags would not match because the third party does not know the original key, so the spacecraft would not even attempt to decrypt the message it received and the message would be discarded. The authentication tag is overhead that must be transmitted with each packet at the expense of transmitting data [11].

As described in [9], operations performed in the GCM algorithm are done in the finite field $\mathbb{F}(2^{128})$. The irreducible polynomial $x^{128}+x^7+x^2+x+1$ is used to construct the finite field [9]. Each 128-bit block of data represents an element of our field [9]. The operations performed in this field are similar to those performed in the $\mathbb{F}\left(2^8\right)$ finite field constructed for AES. Each 128-bit element represents a unique polynomial of the form $b_{127}x^{127}+b_{126}x^{126}+...+b_2x^2+b_1x+b_0$, where the coefficient b_n is the bit in the nth place of the byte [9]. As before, addition of two elements is achieved utilizing the XOR function bitwise on the two elements. Multiplication of two elements is polynomial multiplication modulo $x^{128}+x^7+x^2+x+1$, the base polynomial.

1. Authenticated Encryption

GCM utilizes four inputs during the authenticated encryption process: the original key (*K*), the IV, the plaintext (*P*), and AAD (*A*) [9]. These inputs produce two outputs: the ciphertext (denoted *C*), and the authentication tag (denoted *T*) [9]. It is in the implementation of GCM that inputs from PropCube are required. By determining PropCube's unique inputs, the decryption process was able to be reconstructed in the "propcube_decrypt.py" script found in Appendix A of this thesis. Figure 20 depicts where the four inputs and two outputs are in PropCube's message format.

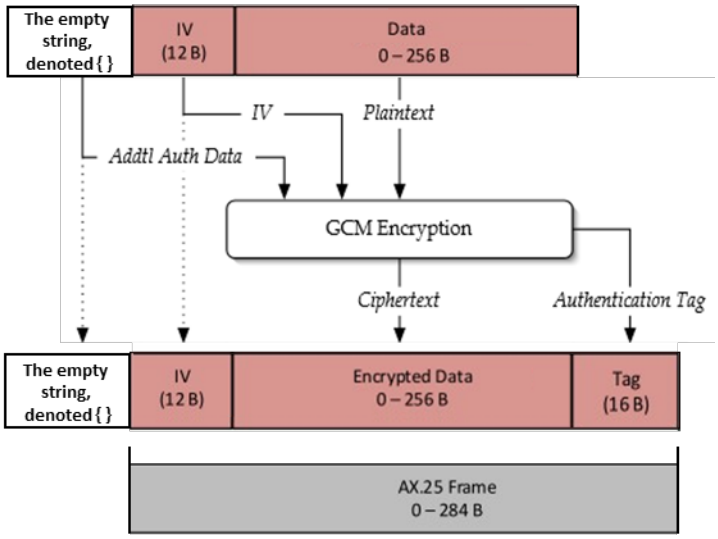

Figure 20. GCM Authenticated Encryption on a PropCube Packet. Adapted from [3], [9].

a. Inputs from PropCube

(1) The Original Key, *K*

The original key is the unique secret cipher key for each PropCube described previously, in section A of this chapter.

(2) The Initialization Vector, *IV*

The IV is utilized to create counter blocks. AES-128 in GCM encrypts the counter blocks. The XOR function is then performed on the encrypted counter block created and the plaintext to produce the ciphertext [9]. Each IV must be distinct and must be a *nonce,* a number used once [9], [10]. This creates a unique set of counter blocks for every message, since the counter blocks are derived from the IV.

PropCube transmits the IV unencrypted, but masked, as the first 12 bytes of the AX.25 frame [3]. The structure of PropCube's IV ensures it is a nonce. The 12 byte IV is composed of two fields: the first eight bytes is a counter, while the last four bytes is a unique identifier for the spacecraft [3]. The counter, transmitted Little Endian, counts the message number the spacecraft is transmitting, so every time a new message is sent out the counter increments by one [3]. This ensures the IV acts as a nonce.

Before transmission, the XOR function is used on the IV and a 12 byte IV mask [3]. This IV mask, which hides our transmitted IV, is constant across all of the spacecraft and is co-located with the cipher key in the AES GCM Keychain File [3]. Before we can utilize the received IV for decryption, we must again perform the XOR function with the IV mask to undo the masking. It is this unmasked IV that is utilized in the authenticated decryption process [3].

Once the IV is unmasked, $Counter_0$, as seen in Figure 21, can be derived. $Counter_0$ is the initial counter block to be used in GCM. As outlined in reference [9], because our IV is 12 bytes, which equals 96 bits, $Counter_0 = IV \,||\, 0^{31}1$. Here the IV is expressed in binary and concatenated with 31 bits of 0 and one bit 1, yielding a total length of 128-bits [9]. In the equation $Counter_0 = IV \,||\, 0^{31}1$, the operation || means to concatenate and 0^{31} is to be interpreted as a string of length 31 with 0 as every bit. To concatenate two strings is to append the second string to the first. For example, 0100||0101 yields the new string 01000101. Strings need not be the same length for concatenation. The length of the output string is the sum of the length of the input strings.

As seen in Figure 21, $Counter_0$ is encrypted using the AES-128 algorithm. Rather than being applied to a plaintext, that output is held to be used in creating the authentication tag, *T*, at the end of the authenticated encryption process [9]. $Counter_0$ is incremented by one to produce $Counter_1$ [9]. In general, $Counter_i = incr(Counter_{i-1})$ where the value of $incr(IV \,||\, I)$ is $IV \,||\, (I + 1 \bmod 2^{32})$ as described in reference [9]. Note, $I + 1 \bmod 2^{32}$ is integer addition and does not utilize the XOR function. In the case of Figure 21, $Counter_1 = IV \,||\, 0^{30}10$ and $Counter_2 = IV \,||\, 0^{30}11$.

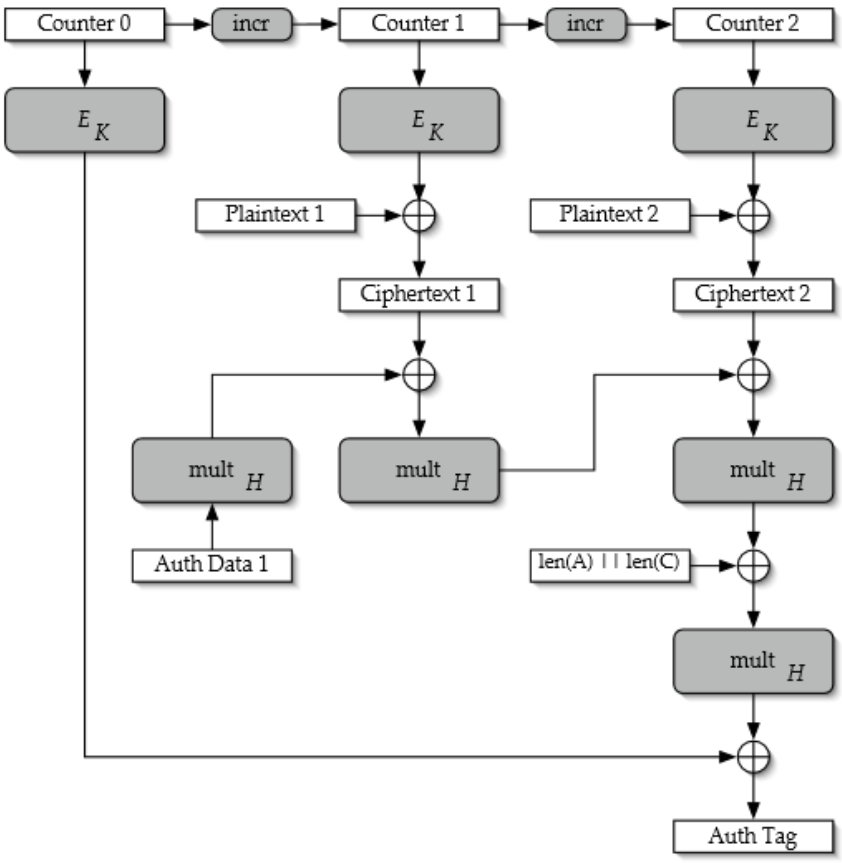

Figure 21. Diagram of Authenticated Encryption. Source: [9].

(3) The plaintext, *P*

The plaintext is the data which needs to be encrypted/decrypted. PropCube transmits the IP, UDP and message data encrypted, as seen in Figure 1.

(4) Additional Authenticated Data, *A*

As depicted in Figure 20, the empty string, denoted {}, is utilized for *A*. This information was not included in the Tyvak documentation. Initially, this research attempted to utilize the AX.25 header as *A*, as the original specification, reference [9], and the NIST publication which outlines GCM block cipher mode of operation both make reference to header information utilized as AAD. However, when the AX.25 header was utilized as *A,* the authenticated decryption failed with all known keys. It was unable to create the

authentication tag. This led to an attempt to decrypt with the {} as *A*. With that, the authenticated decryption was achieved.

Figure 21 graphically depicts how the GCM mode of operation utilizes the described inputs to produce the ciphertext and authentication tag outputs. It creates ciphertexts by first encrypting a 128-bit counter block via the AES-128 algorithm [9]. The encrypted block is then combined with the 128-bit plaintext blocks via the XOR function which produces the ciphertext as the output [9]. The ciphertexts are transmitted as the encrypted data seen in Figure 20.

The authentication tag is generated in this process via the GHASH function [9]. This function uses the ciphertext, *A*, and a polynomial *H*, called the hash subkey, to produce the universal hash value which we call the authentication tag, *T* [9], [10]. As shown in Figure 20, *T* is appended to PropCube's encrypted data for transmission [3], [9].

(5) Hash Subkey, *H*

The hash subkey is not a direct input from PropCube, but it is derived from the cipher key input provided by PropCube. This derived subkey is not to be considered an output of AES-128 in GCM either. The hash subkey is used as an input to create the authentication tag output. To establish *H,* the AES-128 algorithm is used to encrypt a 128-bit block of all zeros with the cipher key, expressed mathematically as $H = E(K, 0^{128})$, in reference [9]. The output of this will yield a polynomial in our field that will be utilized in the GHASH function, which creates the authentication tag.

b. Outputs from AES-128 GCM

(1) Ciphertext

Let n be the number of blocks that will be encrypted. For $i \le n-1$, $C_i = P_i \oplus E(K, Counter_i)$ represents ciphertext production mathematically [9]. For example, the output of encrypting $Counter_1$ is applied to $Plaintext_1$ utilizing the XOR function resulting in $Ciphertext_1$. Similarly, the output of encrypting $Counter_2$ is applied

to *Plaintext*2 utilizing the XOR function resulting in $Ciphertext_2$. These would then be transmitted as encrypted data in a PropCube message.

Let block n be the final block to be encrypted. This block may or may not be a full 128-bit block of plaintext. If the number of bits in our data is a multiple of 128, then all plaintext blocks that need to be encrypted will be full blocks. If the number of bits in our data is not a multiple of 128, then the nth block will not be a full block [9]. Let u be the number of plaintext bits in the nth block, where u is $1 \le u \le 128$. Then to encrypt block n, $C_n = P_n \oplus MSB_u(E(K, Counter_n))$, where MSB_u (S) returns u most significant bits of S as a string [9].

In this process the AES algorithm is applied to the counter, which is always 128-bits. This ensures that the AES algorithm can always be applied, as it requires 128-bit blocks. The $MSB_u()$ function truncates the rightmost $128-u$ bits, resulting in a string which is the same length as the remaining plaintext to be encrypted [9]. This allows the nth block of plaintext to be encrypted regardless of its length. If $u = 128$, then the nth block is a full block and MSB_u (S) would simply return the entire encrypted counter string.

For example, when the MC3 ground station network receives part three of a system status message from Flora, it receives 1,216 bits that are encrypted. This makes nine full 128-bit blocks and one partial 64-bit block. So the 10th block has $u = 64$. $Counter_{10}$ is encrypted with Flora's original key, K, which results in a 128-bit string. The 64 most significant bits, (the leftmost 64) becomes the string S via the MSB_u (S) function. The 64-bit *Plaintext*10 is then combined with S using the XOR function. The result is *Ciphertext*10, which is 64-bits of encrypted data.

(2) Authentication Tag

PropCube uses GCM to generate a 128-bit tag. As [9] explains, the equation $T = MSB_{128}(GHASH(H, A, C) \oplus E(K, Counter_0))$ generates this authentication tag [9]. To calculate the tag, the GHASH operation is applied to the hash subkey derived from encrypting the all zeros block (H), the empty string (AAD), and the ciphertext.

Generally, $GHASH(H, A, C) = X_{m+n+1}$, where m is the number of pieces of AAD inputs, and n is the number of ciphertext inputs [9]. X_i, for $i = 0,\ldots,m+n+1$ are defined in Figure 22.

$$X_i = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} \oplus A_i) \cdot H & \text{for } i = 1, \ldots, m-1 \\ (X_{m-1} \oplus (A_m^* \| 0^{128-v})) \cdot H & \text{for } i = m \\ (X_{i-1} \oplus C_{i-m}) \cdot H & \text{for } i = m+1, \ldots, m+n-1 \\ (X_{m+n-1} \oplus (C_n^* \| 0^{128-u})) \cdot H & \text{for } i = m+n \\ (X_{m+n} \oplus (\text{len}(A) \| \text{len}(C))) \cdot H & \text{for } i = m+n+1. \end{cases}$$

Figure 22. Definition of X_i for the GHASH Operation. Source: [9].

The operation len(), returns the integer value of the input represented as a 64 bit binary string [9]. Since PropCube uses the empty string as A, $m = 0$ and $len(A) = 0^{64}$. Using the previous example of a 1,216-bit message being transmitted, C=1216 and $len(C) = 0^{53} \| 10011000000$. In calculating the GHASH, $len(A) \| len(C) = 0^{64} \| 0^{53} \| 10011000000$.

As described in reference [10], the effect of the GHASH is the calculation of $X_1 \bullet H^{n+m+1} \oplus X_2 \bullet H^{n+m} \oplus \ldots \oplus X_{n+m} \bullet H^2 \oplus X_{n+m+1} \bullet H$, where $\bullet$ represents polynomial multiplication mod $b_{127}x^{127} + b_{126}x^{126} + \ldots + b_2x^2 + b_1x + b_0$. Since PropCube uses the empty string as A, $m = 0$. This results in $A_0 \bullet H = 0$, so $X_1 = C_1 \bullet H$, as $C_1 \oplus 0 = C_1$. The GHASH continues as described in Figure 22, and depicted in Figure 21.

2. Authenticated Decryption

The structure of the authenticated decryption process, as shown in Figure 23, is the same as the authenticated encryption process [9]. However, for the decryption process the tag must be calculated before the encrypted counter and data block are combined with the

XOR function [9]. The data block in this instance is the ciphertext, which is needed to compute an authentication tag. When the ciphertext is combined with the encrypted counter via the XOR function, the plaintext is the resulting output.

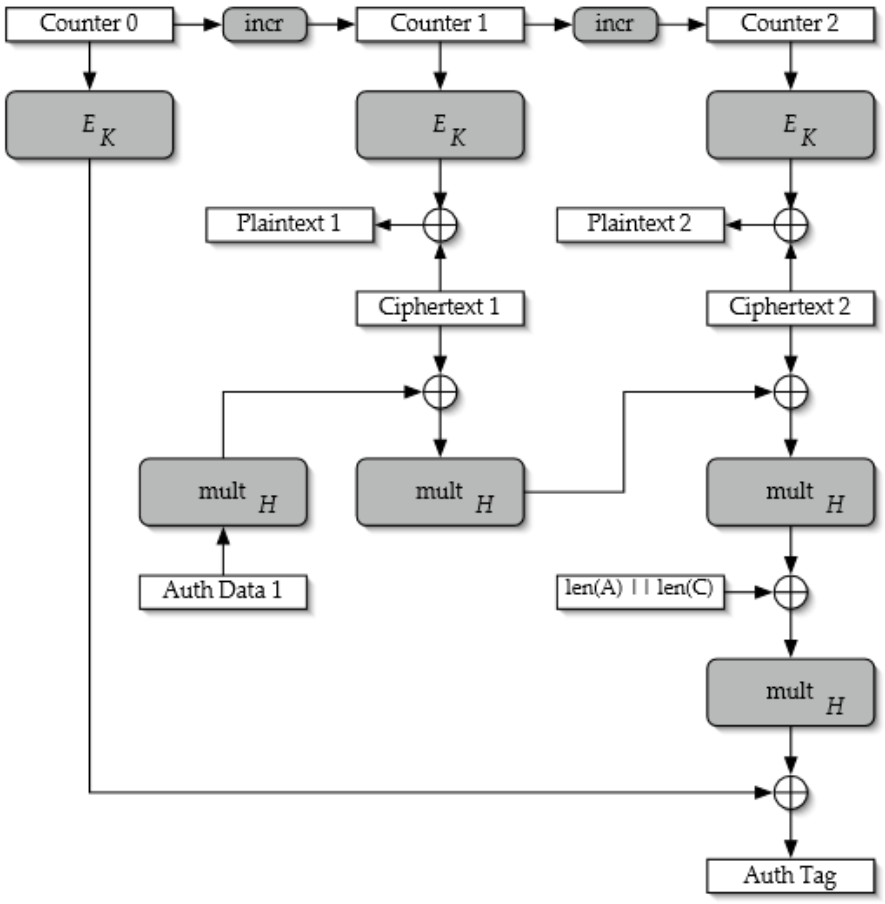

Figure 23. Diagram of Authenticated Decryption. Source: [9]

After the authentication tag is calculated with the decryption process it is compared to the authentication tag received from the originator [9]. If these tags match, then the receiver can be sure it was sent by the trusted party. If these tags do not match, then the message is treated as tampered with and discarded. A summary of the authenticated decryption process is shown in Figure 24.

$$H = E(K, 0^{128})$$
$$Counter_0 = IV \,||\, 0^{31}1$$
$$T = MSB_{128}(GHASH(H, A, C) \oplus E(K, Counter_0)$$
$$Counter_i = incr(Counter_{i-1}), \quad for \quad i = 1, ..., n$$
$$P_i = C_i \oplus E(K, Counter_i), \quad for \quad i = 1, ..., n-1$$
$$P_n = C_n \oplus MSB_u(E(K, Counter_i)), \quad for \quad i = n$$

Figure 24. Summary of Decryption Operations. Adapted from [9].

C. PYTHON CRYPTOGRAPHY LIBRARY

As community moderator, Moshe Zadka, writes on opensource.com, "The first rule of cryptography club is: never *invent* a cryptography system yourself. The second rule of cryptography club is: never *implement* a cryptography system yourself: many real-world holes are found in the *implementation* phase of a cryptosystem as well as in the design" [12]. Because cryptosystems are often difficult to implement in the real-world, the Python library, PyCryptodome, was utilized to build the "propcube_decrypt.py" script. This script is the reconstructed decryption process.

PyCryptodome is a self-contained Python package that implements AEAD, to include AES-128 in GCM [13]. This library allowed for the implementation of the decryption script, found in Appendix A, without the need to create functions that handle all of the difficult operations outlined in Chapter II, sections A and B of this thesis. The creation and layering of complicated functions is often where the implementation phase of a cryptosystem goes awry.

In leveraging open-source libraries, this research was able to marry Tyvak-provided values and recreate AES-128 operating in GCM. The example code in Figure 25 outlined how to generally use the library. The script developed utilizes two arguments, the KISS packet file to be decrypted and the spacecraft name. The spacecraft name identifies which original key is to be used for decryption.

```
import json
from base64 import b64decode
from Crypto.Cipher import AES

# We assume that the key was securely shared beforehand
try:
    b64 = json.loads(json_input)
    json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
    jv = {k:b64decode(b64[k]) for k in json_k}

    cipher = AES.new(key, AES.MODE_CCM, nonce=jv['nonce'])
    cipher.update(jv['header'])
    plaintext = cipher.decrypt_and_verify(jv['ciphertext'], jv['tag'])
    print("The message was: " + plaintext)
except ValueError, KeyError:
    print("Incorrect decryption")
```

Figure 25. Example Decryption Code. Source: [13].

Additional code was utilized to ingest a KISS packet file. Treating the data in the file as a string allows for the KISS packet to be parsed appropriately. First it is unframed so that bytes inserted by the KISS protocol are removed. The remaining data is parsed into the IV (which it then unmasks), the ciphertext, and the tag. In Appendix A of this thesis, there is no line of code to delineate *A*. Since *A* is {}, it was not necessary to directly assign any AAD for the Python library to decrypt the received message.

Once all inputs are identified by the script, the library enacts decryption as described in this chapter, producing a plaintext hexadecimal output string, as seen in Figure 26.

```
cipher = DF 8C F7 10 85 E7 CC BD 36 7C 52 EA 18 00 6F 0B 42 91 90
69 35 01 0D 90 0B 14 65 07 68 F2 BB 0A 29 62 2A 75 FE 19 5F 0F A0
80 60 85 F8 11 0B 54 CC EC EF 24 A5 9A 4A D6 41 E1 24 4A 14 E5 DB
2F 2D 21 96 BB 4D 84 33 97 62 C0 A4 9E F1 36 76 D1 B8 BE 61 5B 0C
7E 92 01 EF D8 C9 62 B8 CB 0B D0 FC A7 42 C1 95 02 70 DF B8 2C 46
1E 4A 8B 1D 1F A3 83 09 32 11 05 5D 15 1E F4 EE 36 D6 99 6D FA D7
EE 93 09 9F 77 3C 6B 88 3D 2F 6B AF 43 AE 1A 71 17 D8 FB D1 87 F4
EE 15 7B 87 0E 42 58 CD 5C 31 F0 D4 B6 97 4E 59 CC 78 73 D4 8D C3
99 97 6C 95 4F 3E 2B EF 89 2C 1A 6C F4 87 13 76 E4 F1 6B 45 50 2C
53 5C D5 52 A3 66 16 BF 63 4A 2F 44 D7 31 1E 5D 00 57 0A 71 C5 61
3B 2C 2B 63 54 1E 89 13 FF 99 AD 95 FC F6 7E 94 5A 04 C1 89 2F 6F
10 41 39 85 01 8B 24 1B 45 95 B2 3E 5D

plain  = 45 00 00 FC 50 6C 20 00 40 11 AF 3D AC 19 01 0A AC 19 01
0B C3 51 B7 A5 02 54 E8 15 F1 04 46 E0 46 D0 47 00 49 80 44 E0 44
D0 42 40 42 00 41 30 41 50 43 B0 44 00 41 80 41 80 43 40 43 30 44
40 44 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 04 3D 70 00 00 09 FC 00 03 4E D9 00 00 04 D7 00
03 8C 49 00 00 01 DD 00 00 01 06 FF FF FF 8A 00 00 01 06 00 00 00
00 00 00 E1 47 00 00 00 3C 00 04 45 1E FF FF D7 4C 00 1B 45 1E 00
00 00 00 00 00 00 00 00 00 00 00 00 03 4E D9 00 00 02 91 00 05 09
37 00 00 01 2A 00 05 06 24 00 00 00 2A 00 00 00 00 00 00 00 00 00
04 41 89 FF FF FF E5 00 05 09 37 00 00 00 7D 00 04 41 89 FF FF FF
C9 00 05 08 31 00 00 00 70 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 04 42 8F FF FF FF E5 00 05
```

Figure 26. Example of a Decrypted Flora Ciphertext Produced by "propcube_decrypt.py" Script

The decrypted data was validated by comparing it to known decrypted data contained in stored .pcap files. The validation of the decryption process allowed for this thesis to begin reconstructing PropCube's data handling process.

III. RECONSTRUCTION OF PROPCUBE'S DATA HANDLING PROCESS FOR INTERPRETATION OF DECRYPTED DATA

After data is decrypted, that data still needs to be interpreted and translated into a human-readable format. After the reconstruction of the decryption process was complete, this thesis was then able to reconstruct PropCube's data handling process, which translates binary bit sequences into tangible, actionable information for PropCube operations. This thesis reconstructs the data handling process in a second python script, seen in Appendix B, called "data_parse.py." To create the "data_parse.py" script this research first examined the Tyvak-provided Data Expression ICD, then created a way to identify message data received by the ground station, and finally reconstructed a process to interpret the received data.

A. DATA EXPRESSION INTERFACE CONTROL DOCUMENT

PropCube utilizes a customized data handling process for interpretation of transmitted information. Each type of message PropCube transmits has a specific format, which is also know by the MC3 network that receives the information. This specific format allows the systems to ingest and interpret commands and data. This format is outlined in the Data Expression ICD [4]. The Data Expression ICD provided by Tyvak is a spreadsheet document that describes the format of message data. As shown in Figure 2 and Figure 3, messages have a message ID byte that follows the IP and UDP information. This byte is called the Frame ID in the spreadsheet document. The primary message types received by the MC3 network are ARSFTP_METADATA, which provides a hash value MD5 sum for a file; ARSFTP_DATA1, which returns requested information such as directory listings and GPS information; and SM_STATUS_RESPONSE, which provides health and telemetry information for the spacecraft. These are indicated by the hexadecimal bytes 04, 05, and F1, respectively [4].

Frame Name	Frame ID	Port Number	Frame Description	Telemetry Point / Command Argument Name	Telemetry Point / Command Argument Description	Size (in Bits)	Units	Data Type	C0	C1	Byte Number
ARSFTP_METADATA				IP information							0-19
				UDP Information							20-27
	4 (04 in Hex)			Frame ID							28
		3223		REQUEST_ID	Transfer Request ID	32		unsigned	0	1	29-32
				SIZE	File Size	32	Bytes	unsigned	0	1	33-36
				UTC_MOD_TIME_S	UTC Mod Time Seconds	32	Seconds	unsigned	0	1	37-40
				UTC_MOD_TIME_NS	UTC Mod Time NanoSeconds	32	NanoSeconds	unsigned	0	1	41-44
				MD5	MD5 Sum	128		array			45-60

Figure 27. ARSFTP_METADATA Message Format. Adapted from [3], [4].

Figure 27 is an example of how the spreadsheet delineates message data for the ARSFTP_METADATA message. ARSFTP_METADATA contains 61 bytes of information that is decrypted once received by a ground station. Bytes 0 through 19 are the IP header, and bytes 20 through 27 is the UDP information [3], [4]. The IP and UDP information was originally absent from the Data Expression ICD but is described in the UHF Space-to-Ground ICD. This thesis merged the relevant information to create a more accurate documentation of received data. Byte 28, 0x 04, is the message ID [3], [4]. The message data that follows the frame ID is delineated by bit length. In ARSFTP_METADATA, the 32 bits that follow the frame ID represent the transfer request ID [4]. As described by the Data Expression ICD, that binary string is then converted to a decimal number. That decimal number is then multiplied by the constant given in column C1, outlined in purple, which is then summed with the constant in C0, outlined in green [4]. The output produced by this affine transformation is the value of the request ID. ARSFTP_METADATA has the same constants for all values, but that does not hold true for other message types; SM_STATUS_RESPONSE utilizes several different affine transformations to produce interpretable data [4].

As described in Chapter II and seen in Appendix A, the reconstructed decryption process produces a hexadecimal string as the output. Thus the "data_parse.py" script acts not on binary strings but on hexadecimal strings, which it converts to decimal values before applying the appropriate affine transformation.

B. DETERMINING MESSAGE DATA

To interpret message data, the "data_parse.py" script must first be able to identify which message type has been received. Initially, this appeared to be an easy task, as the ICD provided frame IDs for all message types. To determine the message type, the data parser would only have to look to byte 28 to determine which message type was received.

However, due to ambiguities in the ICD, relying only on the frame ID for message type determination does not work. The SM_STATUS_RESPONSE message exceeds the 256-byte maximum PropCube encrypts and transmits [3], [4]. The ICD indicates that SM_STATUS_RESPONSE is received as three separate messages, SM_STATUS_PART1, SM_STATUS_PART2, and SM_STATUS_PART3, each with a unique frame ID—0x F2, 0x F3, and 0x F4 respectively [4]. However, in examining the decrypted data, this research determined that these three frame IDs are not being utilized. Only SM_STATUS_RESPONSE messages, with frame ID 0x F1, are transmitted by PropCube, which, utilizing IP and UDP formatting, fragments the larger F1 message into three packets that are transmitted separately. The result is that only SM_STATUS_PART1 contains a frame ID—0x F1—and UDP information. The remaining two packets only contain IP header data and message data. Thus, relying on only the frame ID to determine which message type has been received is not possible.

Nevertheless, SM_STATUS_PART1, SM_STATUS_PART2, and SM_STATUS_PART3 messages were able to be identified and reconstructed by partitioning SM_STATUS_RESPONSE messages in accordance with IP and UDP standards. All received messages contain the IP header. Utilizing the fragment offset information, outlined in green in Figure 28, it was possible to determine which message was received. If the fragment offset indicates that there is 0 offset, then it is the first fragment of the message [14]. This means byte 28 will be the message ID, which can be

used to determine if the message is ARSFTP_METADATA, ARSFTP_DATA1, or SM_STATUS_RESPONSE.

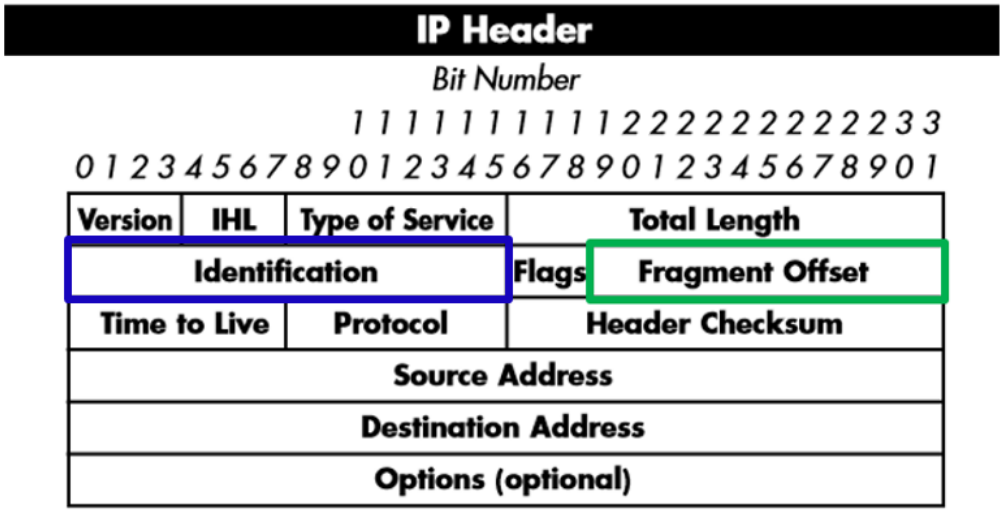

Figure 28. IP Header Format. Source: [14].

Only the SM_STATUS_RESPONSE message is too large to be sent in one packet, so, if the received packet has an offset that does not equal 0, then it is the second or third packet of an SM_STATUS_RESPONSE. A second packet, SM_STATUS_PART2, returns a fragment offset of 29, while a third packet, SM_STATUS_PART3, returns a fragment offset of 58. By utilizing a combination of IP information and frame ID information, the message type can be determined.

Although the fragment offset allows for the determination that SM_STATUS_PART*i*, where *i*= 1, 2, or 3, has been received it does not directly indicate what message data from the larger SM_STATUS_RESPONSE message is contained in each fragment. By utilizing IP and UDP protocols to partition data for fragmentation, SM_STATUS_RESPONSE message data as delineated in the Data Expression ICD, can be partitioned into the three packets, which allows for reconstruction and interpretation of that data.

When fragmentation occurs, the message data of the IP is what is being fragmented [15]. For this reason, only SM_STATUS_PART1 contains UDP information. The fragment offset indicates where the data of the fragment is in relation to the beginning of the message data [15]. This follows IP and UDP standards but was not indicated directly in the Tyvak documentation.

Fragments are grouped by octets, made up of 8 bytes [15]. The fragment offset 29 for SM_STATUS_PART2 indicates that it begins after the 29th octet, which means its data is placed after the 232 bytes of data that were transmitted in packet 1 of SM_STATUS_RESPONSE. The first packet, SM_STATUS_PART1, transmits 252 total bytes of encrypted data, consisting of 20 bytes of IP header data, eight bytes of UDP data, and 224 bytes of message data. The 20 bytes of IP header does not count as data for the offset calculation because it is not message data.

PropCube can transmit a maximum of 256 bytes of data encrypted, but SM_STATUS_RESPONSE packets cannot take advantage of that because of IP fragmentation constraints. Data must be fragmented on octet boundaries [15]. Each fragment requires the IP header so that messages can be reassembled using the identification block, outlined in blue in Figure 28 [14]. All fragments have the same unique identification value so that they can be traced to each other [15]. After the 20 bytes for the IP header are utilized, 236 bytes are left for data. Since 236 is not divisible by eight, there is an inefficiency created; leaving four bytes unusable.

When SM_STATUS_RESPONSE was partitioned in accordance with IP and UDP standards, the reconstructed packets matched the partitioning as described in the Data Expression ICD for messages with hexadecimal frame IDs F2, F3, and F4 with slight differences: namely that the F2 message is indicated with the frame ID F1, while the messages for F3 and F4 do not utilize any frame ID. Figure 29 shows the logic used in the "data_parse.py" script to determine what message type has been received.

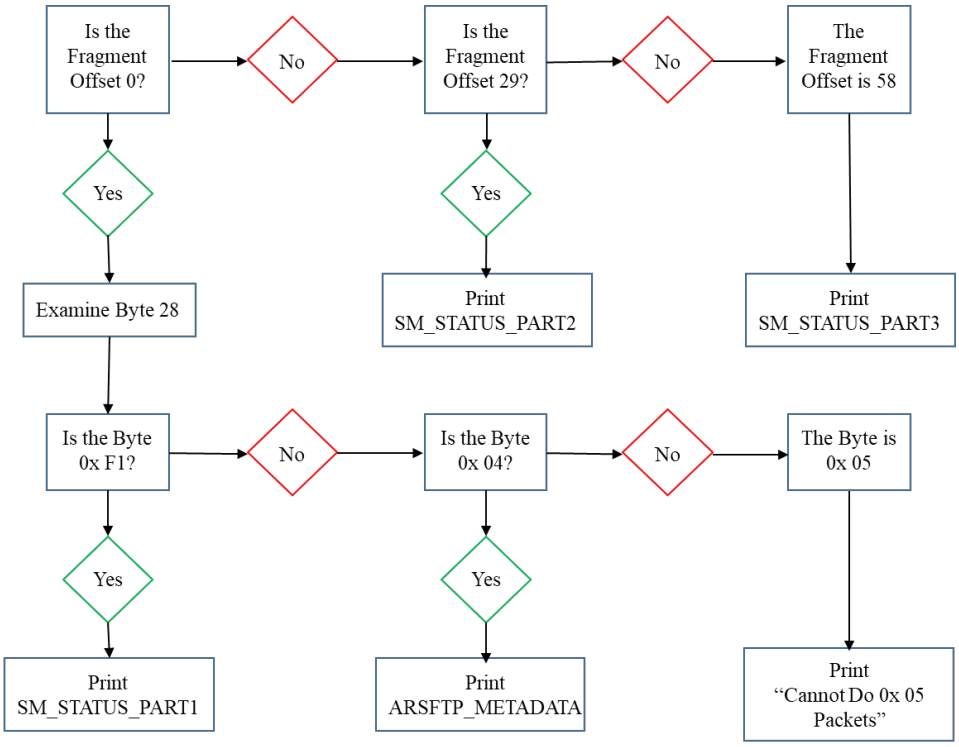

Figure 29. Logic Flow Diagram for "data_parse.py" Script

Utilizing the fragment offset information provided by the IP header and the frame ID, messages can be classified after their decryption. If the fragment offset is zero, the script examines byte 28 to classify the message. If the fragment offset is not zero, it classifies the message as either SM_STATUS_PART2 or SM_STATUS_PART3 as dictated by an offset of 29 or 58, respectively.

C. INTERPRETING THE MESSAGE

Once the message type is identified, the next step is to interpret the data. The message is treated as a hexadecimal string. That string is appropriately dissected based on the message type so that affine transformations from the spreadsheet can be applied to correctly interpret the data. This research was only able to reconstruct SM_STATUS_PART1, SM_STATUS_PART2, SM_STATUS_PART3, and ARSFTP_METADATA messages, as these message types have consistent length and a clearly outlined interpretation description in the Data Expression ICD. This consistency allowed for a reconstruction of their interpretation process. ARSFTP_DATA1, by contrast,

has a variable return structure. An examination of the decrypted data showed varying message lengths, and the description in the Data Expression ICD was not specific enough to enable a reconstruction of the interpretation process in the time available for this research. As seen in Figure 30, the entire block of message data is called "File Data," outlined in red, with no corresponding affine transformation described to interpret the data [4]. The "data_parse.py" script outputs "Cannot do 0x 05 packets" when this message type is received.

Frame Name	Frame ID	Port Number	Frame Description	Telemetry Point / Command Argument Name	Telemetry Point / Command Argument Description	Size (in Bits)	Units	Data Type	C0	C1	Byte Number
ARSFTP_DATA1				IP Information							0-19
				UDP Information							20-27
	5 (05 in hex)			Frame ID							28
		3223	File Data	BLOCK_NUM_0	Data 1 Block Number 0	8		unsigned	0	1	29
				DATA	File Data			array			29-251

Figure 30. ARSFTP_DATA1 Message Format. Adapted from [3], [4].

Although the last 16 bytes of an ARSFTP_METADATA message are also an array with no corresponding affine transformation, this message is still interpretable because the information is defined in a less ambiguous manner. As Figure 27 shows, the Data Expression ICD indicates that "MD5 sum" is the same data type as "File Data. However, unlike the nondescript term "File Data," an MD5 Sum is a standard data block known to be a hexadecimal output, which makes it reproducible.

To enable data interpretation of all messages, five affine transformations are defined in the "data_parse.py" script. First, the hexadecimal string is converted to a decimal value. Then the affine transformations convert the decimal value to a value with units, such as volts or amps. The five transformations, as seen in Figure 31, convert raw values to their corresponding temperature, voltage, current, angle, attitude control, or magnetometer measurement.

```
60  #These functions take our raw data and convert them into their real values
61  def convert_temp(tin):
62      return (float(tin)/64.0)-273.0
63
64
65  def convert_volt_or_amp(temp):
66      return float(temp)/65536.0
67
68
69  def convert_angle(temp):
70      return float(temp)/24.0
71
72
73  def convert_adc(temp):
74      return float(temp)*0.0000000596
75
76
77  def convert_mag(temp):
78      return float(temp)/128.0
```

Figure 31. Python Code for Conversions From "data_parse.py"

As seen in Figure 32, the appropriate affine transformation is delineated with defined coefficients and clearly defined units. The unambiguous conversion provided by the Data Expression ICD allows for data in SM_STAUS_RESPONSE messages to be correctly segmented and converted into human-readable values.

Frame Name	Frame ID	Port Number	Frame Description	Telemetry Point / Command Argument Name	Telemetry Point / Command Argument Description	Size (in Bits)	Units	Data Type	C0	C1	Byte Number
SM_STATUS_PART1				IP information							0-19
				UDP information							20-27
	241 (F1 in Hex)			Frame ID							28
			System Status Part 1	VERSION	Packet Version Number	8		unsigned	0	1	29
				DAUGHTER_A_TEMP	Daughter A Temperature	16	DegC	unsigned	-273	0.015625	30-31
				DAUGHTER_B_TEMP	Daughter B Temperature	16	DegC	unsigned	-273	0.015625	32-33
				3V_TEMP	3V3 PL Temperature	16	DegC	unsigned	-273	0.015625	34-35
				RF_AMP_TEMP	RF Amp Temperature	16	DegC	unsigned	-273	0.015625	36-37
				MINUSZ_INTERNAL_TEMP	Minus Z Internal Temp Sensor	16	DegC	unsigned	-273	0.015625	38-39
				MINUSZ_EXTERNAL_TEMP	Minus Z External Temp Sensor	16	DegC	unsigned	-273	0.015625	40-41
				MINUSX_INTERNAL_TEMP	Minus X Internal Temp Sensor	16	DegC	unsigned	-273	0.015625	42-43
				MINUSX_EXTERNAL_TEMP	Minus X External Temp Sensor	16	DegC	unsigned	-273	0.015625	44-45
				MINUSY_INTERNAL_TEMP	Minus Y Internal Temp Sensor	16	DegC	unsigned	-273	0.015625	46-47
				MINUSY_EXTERNAL_TEMP	Minus Y External Temp Sensor	16	DegC	unsigned	-273	0.015625	48-49
				PLUSZ_INTERNAL_TEMP	Plus Z Internal Temp Sensor	16	DegC	unsigned	-273	0.015625	50-51
				PLUSZ_EXTERNAL_TEMP	Plus Z External Temp Sensor	16	DegC	unsigned	-273	0.015625	52-53
				PLUSX_INTERNAL_TEMP	Plus X Internal Temp Sensor	16	DegC	unsigned	-273	0.015625	54-55
				PLUSX_EXTERNAL_TEMP	Plus X External Temp Sensor	16	DegC	unsigned	-273	0.015625	56-57
				PLUSY_INTERNAL_TEMP	Plus Y Internal Temp Sensor	16	DegC	unsigned	-273	0.015625	58-59
				PLUSY_EXTERNAL_TEMP	Plus Y External Temp Sensor	16	DegC	unsigned	-273	0.015625	60-61
				PAYLOAD_LFREQ_TEMP	Payload Low Frequency Temp	16	DegC	unsigned	-273	0.015625	62-63
				PAYLOAD_HFREQ_TEMP	Payload High Frequency Temp	16	DegC	unsigned	-273	0.015625	64-65
				PAYLOAD_3V0_V	Payload 3V0 Voltage	32	V	unsigned	0	0.000015259	66-69
				PAYLOAD_3V0_A	Payload 3V0 Current	32	A	twosCompliment	0	0.000015259	70-73
				PAYLOAD_5V0_V	Payload 5V0 Voltage	32	V	unsigned	0	0.000015259	74-77
				PAYLOAD_5V0_A	Payload 5V0 Current	32	A	twosCompliment	0	0.000015259	78-81
				PAYLOAD_3V3_V	Payload 3V3 Voltage	32	V	unsigned	0	0.000015259	82-85
				PAYLOAD_3V3_A	Payload 3V3 Current	32	A	twosCompliment	0	0.000015259	86-89
				ATMEL_BUS_V	Atmel Bus Voltage	32	V	unsigned	0	0.000015259	90-93
				ATMEL_BUS_A	Atmel Bus Current	32	A	twosCompliment	0	0.000015259	94-97
				3V_BUS_V	3V Bus Voltage	32	V	unsigned	0	0.000015259	98-101
				3V_BUS_A	3V Bus Current	32	A	twosCompliment	0	0.000015259	102-105
				3VPL_BUS_V	3V Payload Voltage	32	V	unsigned	0	0.000015259	106-109
				3VPL_BUS_A	3V Payload Current	32	A	twosCompliment	0	0.000015259	110-113
				5V_BUS_V	5V Bus Voltage	32	V	unsigned	0	0.000015259	114-117
				5V_BUS_A	5V Bus Current	32	A	twosCompliment	0	0.000015259	118-121
				DAUGHTERA_V	Daughter A Voltage	32	V	unsigned	0	0.000015259	122-125
				DAUGHTERA_A	Daughter A Current	32	A	twosCompliment	0	0.000015259	126-129
				DAUGHTERB_V	Daughter B Voltage	32	V	unsigned	0	0.000015259	130-133
				DAUGHTERB_A	Daughter B Current	32	A	twosCompliment	0	0.000015259	134-137
				FUEL1_V	Fuel 1 Voltage	32	V	unsigned	0	0.000015259	138-141
				FUEL1_A	Fuel 1 Current	32	A	twosCompliment	0	0.000015259	142-145
				FUEL1_Acuum	Fuel 1 Accumulated Current	32	A	twosCompliment	0	0.000015259	146-149
				FUEL2_V	Fuel 2 Voltage	32	V	unsigned	0	0.000015259	150-153
				FUEL2_A	Fuel 2 Current	32	A	twosCompliment	0	0.000015259	154-157
				FUEL2_Acuum	Fuel 2 Accumulated Current	32	A	twosCompliment	0	0.000015259	158-161
				MINUSZ_3V_V	Minus Z 3V Voltage	32	V	unsigned	0	0.000015259	162-165
				MINUSZ_3V_A	Minus Z 3V Current	32	A	twosCompliment	0	0.000015259	166-169
				MINUSZ_5V_V	Minus Z 5V Voltage	32	V	unsigned	0	0.000015259	170-173
				MINUSZ_5V_A	Minus Z 5V Current	32	A	twosCompliment	0	0.000015259	174-177
				MINUSZ_PWRA_V	Minus Z Power A Voltage	32	V	unsigned	0	0.000015259	178-181
				MINUSZ_PWRA_A	Minus Z Power A Current	32	A	twosCompliment	0	0.000015259	182-185
				MINUSZ_PWRB_V	Minus Z Power B Voltage	32	V	unsigned	0	0.000015259	186-189
				MINUSZ_PWRB_A	Minus Z Power B Current	32	A	twosCompliment	0	0.000015259	190-193
				MINUSY_PWRA_V	Minus Y Power A Voltage	32	V	unsigned	0	0.000015259	194-197
				MINUSY_PWRA_A	Minus Y Power A Current	32	A	twosCompliment	0	0.000015259	198-201
				MINUSY_PWRB_V	Minus Y Power B Voltage	32	V	unsigned	0	0.000015259	202-205
				MINUSY_PWRB_A	Minus Y Power B Current	32	A	twosCompliment	0	0.000015259	206-209
				MINUSX_PWRA_V	Minus X Power A Voltage	32	V	unsigned	0	0.000015259	210-213
				MINUSX_PWRA_A	Minus X Power A Current	32	A	twosCompliment	0	0.000015259	214-217
				MINUSX_PWRB_V	Minus X Power B Voltage	32	V	unsigned	0	0.000015259	218-221
				MINUSX_PWRB_A	Minus X Power B Current	32	A	twosCompliment	0	0.000015259	222-225
				PLUSZ_PWRA_V	Plus Z Power A Voltage	32	V	unsigned	0	0.000015259	226-229
				PLUSZ_PWRA_A	Plus Z Power A Current	32	A	twosCompliment	0	0.000015259	230-233
				PLUSZ_PWRB_V	Plus Z Power B Voltage	32	V	unsigned	0	0.000015259	234-237
				PLUSZ_PWRB_A	Plus Z Power B Current	32	A	twosCompliment	0	0.000015259	238-241
				PLUSY_PWRA_V	Plus Y Power A Voltage	32	V	unsigned	0	0.000015259	242-245
				PLUSY_PWRA_A	Plus Y Power A Current	32	A	twosCompliment	0	0.000015259	245-248
				PLUSY_PWRB_Part0_V	Plus Y Power B Part 0 Voltage	16	V	unsigned	0	1	249-251

Figure 32. SM_STATUS_PART1 Message Format. Adapted from [3], [4].

The script, "data_parse.py," is able to take a decrypted string, as seen in Figure 26, and convert it appropriately, as seen in Figure 33, successfully reconstructing the data handling process. Figure 34 demonstrates how to utilize the reconstructed processes.

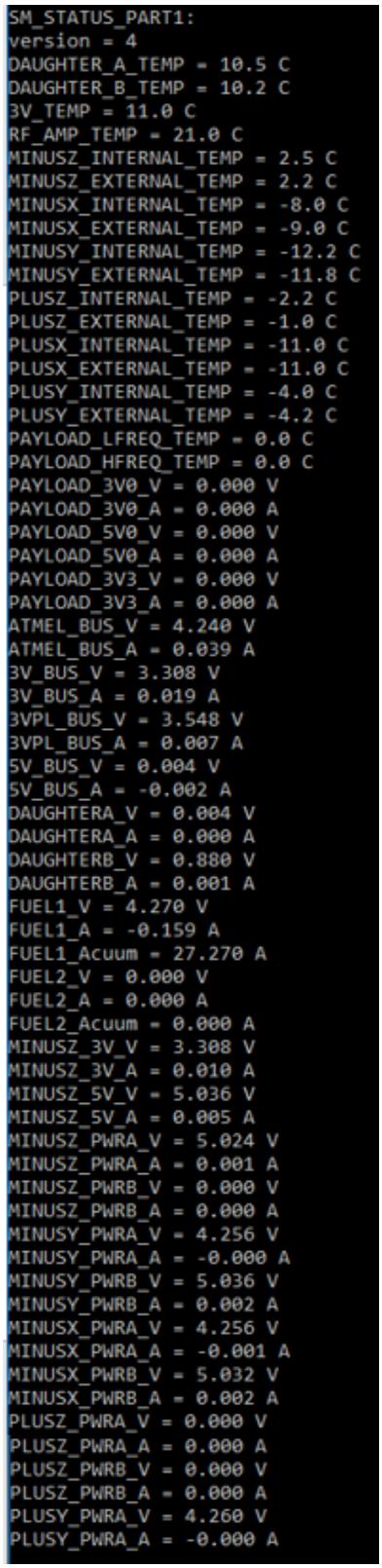

Figure 33. Plaintext From Flora Interpreted by “data_parse.py” Script

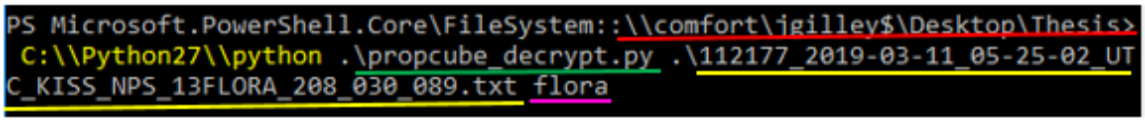

Figure 34. Command to Utilize Reconstructed Processes

By importing the "data_parse.py" script into "propcube_decrypt.py" one command can be utilized to decrypt and interpret a message. From the command terminal access the directory, underlined in red in Figure 34, in which the Python scripts and KISS file logs are stored. The "propcube_decrypt.py" script is called, underlined in green, utilizing two arguments as inputs. The arguments are the KISS log file to be decrypted and interpreted, underlined in yellow, and the spacecraft the data was received from, underlined in fuchsia. The command terminal will then output the data as seen in Figure 33.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. FUTURE WORK AND CONCLUSION

This thesis has reconstructed the decryption process and partially reconstructed the data handling process of PropCube. In doing so, this research found ambiguities in the Tyvak documentation. Resolving the ambiguities took considerable time, which hindered the complete reconstruction of the interpretation process and prevented the reconstructed processes from being implemented in real-time by this project. However, this thesis has laid the foundation for this objective to be accomplished in future work. This chapter will outline areas for future research and conclude with recommendations for improved operational performance.

A. FUTURE WORK

From the work of this thesis, there are three primary projects for future work: implement real-time operations, reconstruct the ARSFTP_DATA1 message, and reconstruct the encryption and uplink data handling processes.

1. Real-Time Concept of Operations

The processes that have been reconstructed in this thesis can work in parallel with the Tyvak created processes. Currently both systems analyze data received from PropCube after the completion of a spacecraft's overhead pass. That means the information received cannot be acted upon until the next time that spacecraft is overhead a ground station. However, the reproduced processes can be altered to enable decryption and interpretation of data while the pass is occurring. With more timely information, operators can make better decisions. For example, as a pass occurs, the ground station is transmitting new commands to be performed by the spacecraft, while simultaneously receiving data about the spacecraft's health status. Currently, operators first try to receive health information for part of one pass to verify a particular spacecraft is in good working order; then whether this data has been downloaded or not, during the rest of the pass, the operator can task a spacecraft to perform a mission task. The spacecraft's battery health is transmitted to the ground in a SM_STATUS_PART1 message. For example, a low battery voltage prevents operators from activating the PropCube's experiments. Over time its solar panels recharge

the battery, allowing operators to resume experimentation. The ability to process data in real-time would allow for the spacecraft's health to be verified immediately, enabling potential same-pass mission tasking.

Further changes could be made to the processes allowing for more autonomous operations. Alert logic can be built into scripts that would notify operators of status conditions outside of acceptable levels. For example, if the battery level were to fall below a certain threshold, operators could be notified with a special warning indicating abnormal operating conditions. The system could then automatically command the spacecraft to pause transmissions allowing it to recharge its battery. These alerts could be made for other known values such as operational temperature ranges as well.

2. Reconstruct ARSFTP_DATA1 Messages

Since ARSFTP_DATA1 messages were too broadly defined in the Data Expression ICD and return variable lengths of data, this research was unable to reproduce the interpretation process. However, the SSAG received some flight software source code from Tyvak as this research was being conducted. This source code could be examined to see if it offers any insight into ARSFTP_DATA1 messages. Future research could also compare decrypted ARSFTP_DATA1 message data to Tyvak log files to see if some way of reproducing the data handling process reveals itself.

3. Uplink Data Handling and Encryption

This thesis focused on downlinked data; future research could reconstruct the uplink processes. By utilizing selected unencrypted uplink messages and examining the Data Expression ICD it may be possible to reconstruct the uplink data handling process in a similar manner the downlink data handling process was reconstructed. Utilizing several KISS packets from each spacecraft that encompass all of the different possible uplink messages should enable the uplinked data handling process to be reconstructed.

To reconstruct the encryption process the PyCryptodome library that enabled the "propcube_decrypt.py" script to be made can also be leveraged. Figure 35 provides an

example code for encryption utilizing AES-128 in GCM. The required encryption components are the same as outlined in Chapter II of this thesis.

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> header = b"header"
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_GCM)
>>> cipher.update(header)
>>> ciphertext, tag = cipher.encrypt_and_digest(data)
>>>
>>> json_k = [ 'nonce', 'header', 'ciphertext', 'tag' ]
>>> json_v = [ b64encode(x).decode('utf-8') for x in cipher.nonce, header, ciphertext, tag ]
>>> result = json.dumps(dict(zip(json_k, json_v)))
>>> print(result)
{"nonce": "DpOK8NIOuSOQlTq+BphKWw==", "header": "aGVhZGVy", "ciphertext": "CZVqyacc", "tag": "B2tBgICbyu
```

Figure 35. Example AES-128 in GCM Encryption Code. Source: [13].

This research found it necessary to reconstruct the decryption process before reconstruction of the data handling process was possible. Similarly to reconstruct the uplink process, typically the data handling process would be reconstructed before the encryption process is reconstructed. However, since the "propcube_decrypt.py" script has already been demonstrated to work, it could be used to test and validate an encryption script. After the encryption process is reconstructed, any string of data could be encrypted. That encrypted string can then be passed to "propcube_decrypt.py." If the string decrypts correctly, then the reconstructed encryption process would be valid. In testing this way, the reconstruction of the encryption process can be done independently of reconstructing the uplink data handling process.

B. CONCLUSION AND RECOMMENDATIONS

It was possible to reconstruct the decryption processes because a well-known, open source standard was utilized for security. Had Tyvak created their own cryptosystem rather than utilizing a NIST standard, it would not have been feasible to reconstruct the decryption process. Likewise, it was possible to recreate data interpretation process largely because a

standard data handling processes were utilized. A clear data expression ICD is also needed for reconstructing the data interpretation process. As this research shows, where data formats are not clearly defined, it is difficult to reconstruct the interpretation process, as was the case with ARSFTP_DATA1 messages. Thus, when possible, it is critical to require detailed documentation from a manufacturer when acquiring technologies.

In addition, based on its examination of the use of AES-128 in GCM for operational use by small satellites with limited power, this research recommends utilizing a different mode of operation for security. Although authentication is a valuable best practice, its application in this environment imposes an overhead cost that compounds data reception problems caused by a noisy environment. Authentication in a symmetric key cryptosystem exists without the addition of a tag. Since the key is secret, if the message decrypts to something interpretable, it can be assumed to have been sent by a party that knows the key [8]. To illustrate this concept, let Eve be the malicious actor. If she wanted to remove a file from the spacecraft, she would have to know what the file was called and how to command its removal. She would then have to send an encrypted command to do so. Since Eve does not know the key, she can send some command encrypted with some random key. When the spacecraft receives that command it would decrypt the message. Since the key was random and likely wrong, the message would be decrypted to a message calling for a file name that does not exist to be removed. Although using the authentication tag would have prevented the process from even happening, the cost imposed is higher than the value of the security gained.

Although it is possible to send a receiving ground station a message which could be interpreted as incorrect data, the cost of doing so would not be worth it for Eve. She would have to send down data at the expected time and from the expected place the ground station was expecting a spacecraft to pass. This received data would decrypt and be interpreted as data due to the format of the received messages. However, the data would likely be outside of believable values due to the incorrect key. It would be easier for Eve to jam the ground station to prevent data collection.

Figure 36 is example code utilizing the PyCryptodome library for AES-128 in Counter mode (CTR). CTR encrypts AES similarly to GCM but without the universal

hash [8], [9]. It encrypts counters, which are then combined with plaintext to provide confidentiality, but without the additional data overhead required for transmitting an authentication tag [8].

```
>>> import json
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Random import get_random_bytes
>>>
>>> data = b"secret"
>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_CTR)
>>> ct_bytes = cipher.encrypt(data)
>>> nonce = b64encode(cipher.nonce).decode('utf-8')
>>> ct = b64encode(ct_bytes).decode('utf-8')
>>> result = json.dumps({'nonce':nonce, 'ciphertext':ct})
>>> print(result)
{"nonce": "XqP8WbylRt0=", "ciphertext": "Mie5lqje"}
```

Figure 36. Example AES-128 in CTR Encryption Code. Source: [13].

Reconstructing the decryption and data handling processes gives better insight not only into how PropCube operates, but also into operations in general. Going forward, the SSAG may use cryptosystems on platforms developed at NPS. Cryptosystems inherently sacrifice efficiency for privacy. To keep data confidential, encryption imposes costs on processing power, memory, and at times transmission overhead. If NPS develops platforms and associated payloads that face similar noisy environments, utilizing AES-128 in a mode of operation like CTR could provide a better cost-benefit balance for operations.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. PROPCUBE DECRYPTOR PYTHON SCRIPT

```
#########################################################################
#   Naval Postgraduate School
#   propcube_decrypt.py
#
#   Revision History:
#   =================
#   Date             Who             What
#   ---------------+--------------+-----------------------------------
#   2019-02-11       Jim Horning     Creation
#   2019-04-23       Jim Horning     Import data_parse
#   2019-04-27       Joseph Gilley   Comments
#
#########################################################################

from __future__ import print_function
from Crypto.Cipher import AES
import binascii
import struct
import sys
import time

import hexdump
import data_parse



class PROPCUBE_Decrypt(object):
    #IV mask and keys are found in ".aes_gcm_keychain" files provided
by Tyvak
    #IV mask is a 12 byte mask represented in hex. The IV Mask
presented here is for reference only, not actual implementation.
    IV_mask = binascii.unhexlify('012345678901234567890ABCD')

    #To use this library, only an 16 byte original key represented in
hex is required. The round keys are derived from the original by the
PyCrytodome library.
    #The keys presented here are for reference only, not actual
implementation. The true IV mask and keys are omitted for security
purposes.
    keys = {'merryw':
binascii.unhexlify('0123456789012345678901234567890AB'),
            'fauna':
binascii.unhexlify('ABCDEFABCDEFABCDEFABCDEFABCDEF12'),
            'flora':
binascii.unhexlify('A1B2C3D4E5F6A7B8C9D0E1F2A3B4C5D6')}

    #The first 18 bytes are AX.25 header and KISS protocol
    XOR_START = 18
    #This is the length in bytes of the IV mask that must. The packet
IV is the 12 bytes that follow the AX.25 and KISS protocol bytes
    XOR_LENGTH = 12
```

```
    START_OF_CIPHER_DATA = 30
    TAG_LENGTH = 16

    def __init__(self, debug=False):
        self.debug = debug


    def make_addr(self, addr):
        return '%d.%d.%d.%d' % ((addr&0xFF000000)>>24,
(addr&0x00FF0000)>>16, (addr&0x0000FF00)>>8, addr&0xFF)


    #The first 20 bytes of our encrypted data is the IP information of
the message
    def decode_ipv4_header(self, header):
        protocol_lookup = {1:'ICMP', 2:'IGMP', 6:'TCP', 9:'IGRP',
17:'UDP', 47:'GRE', 50:'ESP', 51:'AH', 57:'SKIP', 88:'IEGRP',
89:'OSPF', 115:'L2TP'}
        (b0, tos, tot_len, id, w3, ttl, protocol, check, saddr, daddr)
= struct.unpack('!BBHHHBBHII', header)
        version = (b0&0xF0)>>4
        ihl = (b0&0x0F)
        flags = (w3&0xE000)>>12
        offset = w3&0x1FFF
        if self.debug:
            print('version             = %u' %version)
            print('IHL                 = %u' %ihl)
            print('TOS                 = %u' %tos)
            print('total len           = %u' %tot_len)
            print('ID                  = %u' %id)
            print('flags               = 0x%01X' %flags, end='')
            if flags&0x4:
                print('  Do Not Fragment  ', end='')
            if flags&0x2:
                print(' More Fragments follow  ', end='')
            print()
            print('fragment offset     = %u' %offset)
            print('TTL                 = %u' %ttl)
            print('protocol            = %u: ' %protocol, end='')
            if protocol in protocol_lookup:
                print(protocol_lookup[protocol])
            else:
                print('UNKNOWN')
            print('source address      = %s' %(self.make_addr(saddr)))
            print('destination address = %s' %(self.make_addr(daddr)))

    #If the packet has a fragment offset of 0, found in the IP header,
the following 8 bytes contain UDP information
    def decode_udp_header(self, header):
        (sp, dp) = struct.unpack('!HH', header)
        if self.debug:
            print('source port      = %u'%sp)
            print('destination port = %u'%dp)

    def unframe(self, msg):
```

```
        # Unframe a KISS packet
        #Before decryption can occur, any bytes that were inserted by
the KISS protocol must be removed
        FEND = "\xC0"
        FESC = "\xDB"
        TFEND = "\xDC"
        TFESC = "\xDD"
        unframed = ''
        i = 0
        try:
            while i < (len(msg)):
                if msg[i] == FESC:
                    if msg[i+1] == TFEND:
                        unframed += FEND
                        i += 2
                    elif msg[i+1] == TFESC:
                        unframed += FESC
                        i += 2
                    else:
                        return ''
                else:
                   unframed += msg[i]
                   i += 1
        except:
            print('Error: cannot unframe the packet (index=%d)'%i)
        return unframed


    def xor(self, data_a, data_b):
        # XOR two arbitrary lengthed (but equal) byte-strings
        # (performed on successive bytes)
        x = ''
        for a, b in zip(data_a, data_b):
            x += struct.pack('B', ord(a) ^ ord(b))
        return x


    def hprint(self, data, hexlify=True):
        if hexlify:
            temp = binascii.hexlify(data).upper()
        else:
            temp = data
        s = ''
        for i in range(0, len(temp), 2):
            s += temp[i]+temp[i+1] + ' '
        return s


    def process_packet(self, kissed_data, spacecraft):
        if self.debug:
            print('---------------------------------')
            print('packet = %s' %self.hprint(kissed_data))

        raw_data = self.unframe(binascii.unhexlify(kissed_data))
```

```
        raw_data = raw_data[:-1]        # remove end C0 from the
unframed data (the starting C0 is still there)

        #This unmasks the IV, which is used as the nonce
        nonce =
self.xor(raw_data[PROPCUBE_Decrypt.XOR_START:PROPCUBE_Decrypt.XOR_START
+PROPCUBE_Decrypt.XOR_LENGTH], PROPCUBE_Decrypt.IV_mask)

        #This extracts the encrypted information from the KISS packet
        ciphertext = raw_data[PROPCUBE_Decrypt.START_OF_CIPHER_DATA:-
PROPCUBE_Decrypt.TAG_LENGTH]
        #The Tag is the last 16 bytes of the unframed KISS packet
        tag = raw_data[-PROPCUBE_Decrypt.TAG_LENGTH:]

        #This selects the appropriate key for our spacecraft. merryw is
used to indicate Merryweather, flora to indicate Flora, and fauna to
indicate Fauna
        key = PROPCUBE_Decrypt.keys[spacecraft]
        cipher = AES.new(key, AES.MODE_GCM, nonce)
        plaintext = cipher.decrypt_and_verify(ciphertext, tag)

        if self.debug:
            print('IV     = %s' %self.hprint(PROPCUBE_Decrypt.IV_mask))
            print('key    = %s' %self.hprint(key))
            print('nonce  = %s' %self.hprint(nonce))
            print('cipher = %s' %self.hprint(ciphertext))
            print('plain  = %s' %self.hprint(plaintext))
            print()
        self.decode_ipv4_header(plaintext[0:20])
        self.decode_udp_header(plaintext[20:24])

        #This enables one command to be used in the terminal window to
decyrpt and parse a KISS log file
        data_parse.data_parse_packet(plaintext, spacecraft)

        return(plaintext)


    def process_file(self, fname, spacecraft):
        # open a ASCII-encoded file and process all lines that begin
with a 'C0'
        fp = open(fname, 'r')
        data = fp.read().split('\n')
        fp.close()
        counter = 0
        for line in data:
            if line[0:2] == 'C0':
                data = self.process_packet(line, spacecraft)
                fp = open('%s_%d.bin'%(spacecraft, counter), 'wb')
                fp.write(data)
                fp.close()
                counter += 1


if __name__ == "__main__":
```

```
#   fname = '41695_2017-05-18_15-46-
53_UTC_KISS_HSFL_10MERRYW_188_036_051.txt'
#   spacecraft = 'merryw'
    fname = sys.argv[1]
    spacecraft = sys.argv[2]
    decryptor = PROPCUBE_Decrypt()
    decryptor.process_file(fname, spacecraft)
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. DATA PARSE PYTHON CODE

```
#########################################################################
#   Naval Postgraduate School
#   data_parse.py
#
#   Revision History:
#   =================
#   Date             Who             What
#   ---------------+--------------+-----------------------------------
#   2019-04-23       Jim Horning     Creation
#   2019-04-27       Joseph Gilley   Comments
#
#########################################################################

from __future__ import print_function
from Crypto.Cipher import AES
import binascii
import struct
import sys
import time

import hexdump


def make_addr(addr):
    return '%d.%d.%d.%d' % ((addr&0xFF000000)>>24,
(addr&0x00FF0000)>>16, (addr&0x0000FF00)>>8, addr&0xFF)


def decode_ipv4_header(header, do_print=False):
    protocol_lookup = {1:'ICMP', 2:'IGMP', 6:'TCP', 9:'IGRP', 17:'UDP',
47:'GRE', 50:'ESP', 51:'AH', 57:'SKIP', 88:'IEGRP', 89:'OSPF',
115:'L2TP'}
    (b0, tos, tot_len, id, w3, ttl, protocol, check, saddr, daddr) =
struct.unpack('!BBHHHBBHII', header)
    version = (b0&0xF0)>>4
    ihl = (b0&0x0F)
    flags = (w3&0xE000)>>12
    offset = w3&0x1FFF
    if do_print:
        print('version             = %u' %version)
        print('IHL                 = %u' %ihl)
        print('TOS                 = %u' %tos)
        print('total len           = %u' %tot_len)
        print('ID                  = %u' %id)
        print('flags               = 0x%01X' %flags, end='')
        if flags&0x4:
            print('  Do Not Fragment  ', end='')
        if flags&0x2:
            print(' More Fragments follow  ', end='')
        print()
        print('fragment offset     = %u' %offset)
```

```
        print('TTL                 = %u' %ttl)
        print('protocol            = %u: ' %protocol, end='')
        if protocol in protocol_lookup:
            print(protocol_lookup[protocol])
        else:
            print('UNKNOWN')
        print('source address      = %s' %(make_addr(saddr)))
        print('destination address = %s' %(make_addr(daddr)))

    return offset


#These functions take our raw data and convert them into their real
values
def convert_temp(tin):
    return (float(tin)/64.0)-273.0


def convert_volt_or_amp(temp):
    return float(temp)/65536.0


def convert_angle(temp):
    return float(temp)/24.0


def convert_adc(temp):
    return float(temp)*0.0000000596


def convert_mag(temp):
    return float(temp)/128.0

#This enables KISS log files from Flora or Merryweather to be converted
into human readable data
#Flora and Merryweather have the same message format, which differs
slightly from Fauna
def data_parse_packet(data, satname):
    offset = decode_ipv4_header(data[0:20])

    if offset == 0:
        # Part 1
        if satname == 'flora' or satname == 'merryw':
            temp_names = ['DAUGHTER_A_TEMP', 'DAUGHTER_B_TEMP',
'3V_TEMP', 'RF_AMP_TEMP', 'MINUSZ_INTERNAL_TEMP',
'MINUSZ_EXTERNAL_TEMP', 'MINUSX_INTERNAL_TEMP', 'MINUSX_EXTERNAL_TEMP',
'MINUSY_INTERNAL_TEMP', 'MINUSY_EXTERNAL_TEMP', 'PLUSZ_INTERNAL_TEMP',
'PLUSZ_EXTERNAL_TEMP', 'PLUSX_INTERNAL_TEMP', 'PLUSX_EXTERNAL_TEMP',
'PLUSY_INTERNAL_TEMP', 'PLUSY_EXTERNAL_TEMP', 'PAYLOAD_LFREQ_TEMP',
'PAYLOAD_HFREQ_TEMP']
            va_names = ['PAYLOAD_3V0_V', 'PAYLOAD_3V0_A',
'PAYLOAD_5V0_V', 'PAYLOAD_5V0_A', 'PAYLOAD_3V3_V', 'PAYLOAD_3V3_A',
'ATMEL_BUS_V', 'ATMEL_BUS_A', '3V_BUS_V', '3V_BUS_A', '3VPL_BUS_V',
'3VPL_BUS_A', '5V_BUS_V', '5V_BUS_A', 'DAUGHTERA_V', 'DAUGHTERA_A',
'DAUGHTERB_V', 'DAUGHTERB_A', 'FUEL1_V', 'FUEL1_A', 'FUEL1_Acuum',
```

```
'FUEL2_V', 'FUEL2_A', 'FUEL2_Acuum', 'MINUSZ_3V_V', 'MINUSZ_3V_A',
'MINUSZ_5V_V', 'MINUSZ_5V_A', 'MINUSZ_PWRA_V', 'MINUSZ_PWRA_A',
'MINUSZ_PWRB_V', 'MINUSZ_PWRB_A', 'MINUSY_PWRA_V', 'MINUSY_PWRA_A',
'MINUSY_PWRB_V', 'MINUSY_PWRB_A', 'MINUSX_PWRA_V', 'MINUSX_PWRA_A',
'MINUSX_PWRB_V', 'MINUSX_PWRB_A', 'PLUSZ_PWRA_V', 'PLUSZ_PWRA_A',
'PLUSZ_PWRB_V', 'PLUSZ_PWRB_A', 'PLUSY_PWRA_V', 'PLUSY_PWRA_A']
            # need to inspect the 0x28 offset byte
            b28 = struct.unpack('B', data[28:28+1])[0]
#           print('b28 = %u'%b28)
            if b28 == 4:
                print('\nARSFTP_METADATA:')
                # ARSFTP_METADATA packet
                (request_id, fsize, utc_sec, utc_nsec) =
struct.unpack('>IIII', data[29:29+4*4])
                md5 = struct.unpack('16s', data[29+4*4:])[0]
                md5 = ''.join('%02X ' %ord(x) for x in md5)
                print('request_id=%u, fsize=%u, utc_sec=%u,
utc_nsec=%u' %(request_id, fsize, utc_sec, utc_nsec))
                print('md5 = %s'%md5)
            elif b28 == 5:
                # can't deal with these
                print('\nCannot do b28==5 packets')
            elif b28 == 0xF1:
                # SM_STATUS_PART1
                print('\nSM_STATUS_PART1:')
                version = struct.unpack('>B', data[29:29+1])[0]
                temps = struct.unpack('>18H', data[30:30+18*2])
                s = 23*'Ii'
                volts_amps = struct.unpack('>%s'%s, data[66:250])
                print('version = %u'%version)
                for tname, t in zip(temp_names, temps):
                    temp = convert_temp(t)
                    print('%s = %2.1f C' %(tname, temp))

                for vaname, va in zip(va_names, volts_amps):
                    va = convert_volt_or_amp(va)
                    s = vaname.split('_')[-1]
                    if 'V' in s:
                        print('%s = %2.3f V' %(vaname, va))
                    else:
                        print('%s = %2.3f A' %(vaname, va))


    elif offset == 29:
        # Part 2
        # If the IP header indicates there is a message offset of 29,
then we know that it is the second part of a system status
        # This offset indication is the same for all spacecraft
        print('\nPart2:')

        va_names = ['PLUSY_PWRB_A', 'PLUSX_PWRA_V', 'PLUSX_PWRA_A',
'PLUSX_PWRB_V', 'PLUSX_PWRB_A']
        angle_names = ['MINUSZ_SAS_X', 'MINUSZ_SAS_Y', 'MINUSY_SAS_X',
'MINUSY_SAS_Y', 'MINUSX_SAS_X', 'MINUSX_SAS_Y', 'PLUSZ_SAS_X',
```

```
'PLUSZ_SAS_Y', 'PLUSY_SAS_X', 'PLUSY_SAS_Y', 'PLUSX_SAS_X',
'PLUSX_SAS_Y']
        adc_names = ['MINUSZ_ADC', 'MINUSY_ADC', 'MINUSX_ADC',
'PLUSZ_ADC', 'PLUSY_ADC', 'PLUSX_ADC']
        mag_names = ['MINUSZ_MAG_X', 'MINUSZ_MAG_Y', 'MINUSZ_MAG_Z',
'MINUSY_MAG_X', 'MINUSY_MAG_Y', 'MINUSY_MAG_Z', 'MINUSX_MAG_X',
'MINUSX_MAG_Y', 'MINUSX_MAG_Z', 'PLUSZ_MAG_X', 'PLUSZ_MAG_Y',
'PLUSZ_MAG_Z', 'PLUSY_MAG_X', 'PLUSY_MAG_Y', 'PLUSY_MAG_Z',
'PLUSX_MAG_X', 'PLUSX_MAG_Y', 'PLUSX_MAG_Z']
        pib_names = ['PIB_HP_3V3_V', 'PIB_HP_3V3_A', 'PIB_VSUM1_V',
'PIB_VSUM1_A', 'PIB_VSUM2_V', 'PIB_VSUM2_A']

        volts_amps = struct.unpack('>iIiIi', data[22:42])
        angles = struct.unpack('>12I', data[42:90])
        adcs = struct.unpack('>6i', data[90:114])
        mags = struct.unpack('>18i', data[114:186])
        pibs = struct.unpack('>IiIiIi', data[228:252])
        for name, va in zip(va_names, volts_amps):
            va = convert_volt_or_amp(va)
            s = name.split('_')[-1]
            if 'V' in s:
                print('%s = %2.3f V' %(name, va))
            else:
                print('%s = %2.3f A' %(name, va))
        for name, a in zip(angle_names, angles):
            a = convert_angle(a)
            print('%s = %2.3f degrees' %(name, a))
        for name, a in zip(adc_names, adcs):
            a = convert_adc(a)
            print('%s = %2.3f' %(name, a))
        for name, a in zip(mag_names, mags):
            a = convert_mag(a)
            print('%s = %d mGauss' %(name, a))
        for name, va in zip(pib_names, pibs):
            va = convert_volt_or_amp(va)
            s = name.split('_')[-1]
            if 'V' in s:
                print('%s = %2.3f V' %(name, va))
            else:
                print('%s = %2.3f A' %(name, va))


    elif offset == 58:
        # Part 3
        # If the IP header indicates there is a message offset of 58,
then we know that it is the second part of a system status
        # This offset indication is the same for all spacecraft
        print('\nPart3:\n')

        va_names = ['PIB_VSUM3_V', 'PIB_VSUM3_A', 'PIB_HP_BOOST_V',
'PIB_HP_BOOST_A', 'PIB_GPS_3V3_V', 'PIB_GPS_3V3_A']
        misc_names = ['DSTRING', 'DUINT', 'USERTIME', 'LPUSERTIME',
'SYSTIME', 'IDLETIME', 'PAGEIN', 'PAGEOUT', 'SWAPIN', 'SWAPOUT',
'INTR', 'CTXT', 'BTIME', 'PROCESSES', 'PROCS_RUNNING', 'PROCS_BLOCKED',
```

```
'MEMFREE', 'BUFFERS', 'CACHED', 'ACTIVE', 'INACTIVE', 'VMALLOCTOTAL',
'VMALLOCUSED', 'FREEDATAFLASH', 'FREESD', 'UNIXTIME']

        volts_amps = struct.unpack('>IiIiIi', data[20:44])
        PIB_HP_BOOST_TEMP = struct.unpack('>H', data[44:46])[0]
        misc = struct.unpack('>26I', data[46:150])
        LDC = struct.unpack('>H', data[150:152])[0]

        for name, va in zip(va_names, volts_amps):
            va = convert_volt_or_amp(va)
            s = name.split('_')[-1]
            if 'V' in s:
                print('%s = %2.3f V' %(name, va))
            else:
                print('%s = %2.3f A' %(name, va))
        temp = convert_temp(PIB_HP_BOOST_TEMP)
        print('PIB_HP_BOOST_TEMP = %2.1f C' %temp)
        for name, va in zip(misc_names, misc):
            print('%s = %u' %(name, va))
        print('LDC = %u'%LDC)


def data_parse_file(fname, satname):
    fp = open(fname, 'rb')
    data = fp.read()
    fp.close()

#   hexdump.hexdump(data)

    offset = decode_ipv4_header(data[0:20])

    if offset == 0:
        # Part 1
        if satname == 'flora' or satname == 'merryw':
            #This is the temperature information provided in the first
packet of a system status
            temp_names = ['DAUGHTER_A_TEMP', 'DAUGHTER_B_TEMP',
'3V_TEMP', 'RF_AMP_TEMP', 'MINUSZ_INTERNAL_TEMP',
'MINUSZ_EXTERNAL_TEMP', 'MINUSX_INTERNAL_TEMP', 'MINUSX_EXTERNAL_TEMP',
'MINUSY_INTERNAL_TEMP', 'MINUSY_EXTERNAL_TEMP', 'PLUSZ_INTERNAL_TEMP',
'PLUSZ_EXTERNAL_TEMP', 'PLUSX_INTERNAL_TEMP', 'PLUSX_EXTERNAL_TEMP',
'PLUSY_INTERNAL_TEMP', 'PLUSY_EXTERNAL_TEMP', 'PAYLOAD_LFREQ_TEMP',
'PAYLOAD_HFREQ_TEMP']
            #This is the voltage and current information provided in
the first packet of a system status
            va_names = ['PAYLOAD_3V0_V', 'PAYLOAD_3V0_A',
'PAYLOAD_5V0_V', 'PAYLOAD_5V0_A', 'PAYLOAD_3V3_V', 'PAYLOAD_3V3_A',
'ATMEL_BUS_V', 'ATMEL_BUS_A', '3V_BUS_V', '3V_BUS_A', '3VPL_BUS_V',
'3VPL_BUS_A', '5V_BUS_V', '5V_BUS_A', 'DAUGHTERA_V', 'DAUGHTERA_A',
'DAUGHTERB_V', 'DAUGHTERB_A', 'FUEL1_V', 'FUEL1_A', 'FUEL1_Acuum',
'FUEL2_V', 'FUEL2_A', 'FUEL2_Acuum', 'MINUSZ_3V_V', 'MINUSZ_3V_A',
'MINUSZ_5V_V', 'MINUSZ_5V_A', 'MINUSZ_PWRA_V', 'MINUSZ_PWRA_A',
'MINUSZ_PWRB_V', 'MINUSZ_PWRB_A', 'MINUSY_PWRA_V', 'MINUSY_PWRA_A',
'MINUSY_PWRB_V', 'MINUSY_PWRB_A', 'MINUSX_PWRA_V', 'MINUSX_PWRA_A',
```

```
'MINUSX_PWRB_V', 'MINUSX_PWRB_A', 'PLUSZ_PWRA_V', 'PLUSZ_PWRA_A',
'PLUSZ_PWRB_V', 'PLUSZ_PWRB_A', 'PLUSY_PWRA_V', 'PLUSY_PWRA_A']
            # need to inspect the 0x28 offset byte
            b28 = struct.unpack('B', data[28:28+1])[0]
            print('b28 = %u'%b28)
            if b28 == 4:
                # ARSFTP_METADATA packet
                (request_id, fsize, utc_sec, utc_nsec) =
struct.unpack('>IIII', data[29:29+4*4])
                md5 = struct.unpack('16s', data[29+4*4:])[0]
                md5 = ''.join('%02X ' %ord(x) for x in md5)
                print('request_id=%u, fsize=%u, utc_sec=%u,
utc_nsec=%u' %(request_id, fsize, utc_sec, utc_nsec))
                print('md5 = %s'%md5)
            elif b28 == 5:
                # can't deal with these
                print('Cannot do b28==5 packets')
            elif b28 == 0xF1:
                # SM_STATUS_PART1
                version = struct.unpack('>B', data[29:29+1])[0]
                temps = struct.unpack('>18H', data[30:30+18*2])
                s = 23*'Ii'
                volts_amps = struct.unpack('>%s'%s, data[66:250])
                print('version = %u'%version)
                for tname, t in zip(temp_names, temps):
                    temp = convert_temp(t)
                    print('%s = %2.1f C' %(tname, temp))

                for vaname, va in zip(va_names, volts_amps):
                    va = convert_volt_or_amp(va)
                    s = vaname.split('_')[-1]
                    if 'V' in s:
                        print('%s = %2.3f V' %(vaname, va))
                    else:
                        print('%s = %2.3f A' %(vaname, va))


    elif offset == 29:
        # Part 2
        # all spacecraft have the same
        #This is the voltage and current information provided in the
second packet of a system status
        va_names = ['PLUSY_PWRB_A', 'PLUSX_PWRA_V', 'PLUSX_PWRA_A',
'PLUSX_PWRB_V', 'PLUSX_PWRB_A']
        #This is the solar angle sensor information provided in the
second packet of a system status
        angle_names = ['MINUSZ_SAS_X', 'MINUSZ_SAS_Y', 'MINUSY_SAS_X',
'MINUSY_SAS_Y', 'MINUSX_SAS_X', 'MINUSX_SAS_Y', 'PLUSZ_SAS_X',
'PLUSZ_SAS_Y', 'PLUSY_SAS_X', 'PLUSY_SAS_Y', 'PLUSX_SAS_X',
'PLUSX_SAS_Y']
        #This is the attitude control information provided in the
second packet of a system status
        adc_names = ['MINUSZ_ADC', 'MINUSY_ADC', 'MINUSX_ADC',
'PLUSZ_ADC', 'PLUSY_ADC', 'PLUSX_ADC']
```

```
        #This is the magnetometer information provided in the second
packet of a system status
        mag_names = ['MINUSZ_MAG_X', 'MINUSZ_MAG_Y', 'MINUSZ_MAG_Z',
'MINUSY_MAG_X', 'MINUSY_MAG_Y', 'MINUSY_MAG_Z', 'MINUSX_MAG_X',
'MINUSX_MAG_Y', 'MINUSX_MAG_Z', 'PLUSZ_MAG_X', 'PLUSZ_MAG_Y',
'PLUSZ_MAG_Z', 'PLUSY_MAG_X', 'PLUSY_MAG_Y', 'PLUSY_MAG_Z',
'PLUSX_MAG_X', 'PLUSX_MAG_Y', 'PLUSX_MAG_Z']
        #This is the adiitional voltage and current information
provided in the second packet of a system status
        pib_names = ['PIB_HP_3V3_V', 'PIB_HP_3V3_A', 'PIB_VSUM1_V',
'PIB_VSUM1_A', 'PIB_VSUM2_V', 'PIB_VSUM2_A']

        volts_amps = struct.unpack('>iIiIi', data[22:42])
        angles = struct.unpack('>12I', data[42:90])
        adcs = struct.unpack('>6i', data[90:114])
        mags = struct.unpack('>18i', data[114:186])
        pibs = struct.unpack('>IiIiIi', data[228:252])
        for name, va in zip(va_names, volts_amps):
            va = convert_volt_or_amp(va)
            s = name.split('_')[-1]
            if 'V' in s:
                print('%s = %2.3f V' %(name, va))
            else:
                print('%s = %2.3f A' %(name, va))
        for name, a in zip(angle_names, angles):
            a = convert_angle(a)
            print('%s = %2.3f degrees' %(name, a))
        for name, a in zip(adc_names, adcs):
            a = convert_adc(a)
            print('%s = %2.3f' %(name, a))
        for name, a in zip(mag_names, mags):
            a = convert_mag(a)
            print('%s = %d mGauss' %(name, a))
        for name, va in zip(pib_names, pibs):
            va = convert_volt_or_amp(va)
            s = name.split('_')[-1]
            if 'V' in s:
                print('%s = %2.3f V' %(name, va))
            else:
                print('%s = %2.3f A' %(name, va))


    elif offset == 58:
        # Part 3
        # all spacecraft have the same
        #This is the voltage and current information provided in the
third packet of a system status
        va_names = ['PIB_VSUM3_V', 'PIB_VSUM3_A', 'PIB_HP_BOOST_V',
'PIB_HP_BOOST_A', 'PIB_GPS_3V3_V', 'PIB_GPS_3V3_A']
        #This is the additional information provided in the third
packet of a system status
        misc_names = ['DSTRING', 'DUINT', 'USERTIME', 'LPUSERTIME',
'SYSTIME', 'IDLETIME', 'PAGEIN', 'PAGEOUT', 'SWAPIN', 'SWAPOUT',
'INTR', 'CTXT', 'BTIME', 'PROCESSES', 'PROCS_RUNNING', 'PROCS_BLOCKED',
```

```
'MEMFREE', 'BUFFERS', 'CACHED', 'ACTIVE', 'INACTIVE', 'VMALLOCTOTAL',
'VMALLOCUSED', 'FREEDATAFLASH', 'FREESD', 'UNIXTIME']

        volts_amps = struct.unpack('>IiIiIi', data[20:44])
        PIB_HP_BOOST_TEMP = struct.unpack('>H', data[44:46])[0]
        misc = struct.unpack('>26I', data[46:150])
        LDC = struct.unpack('>H', data[150:152])[0]

        for name, va in zip(va_names, volts_amps):
            va = convert_volt_or_amp(va)
            s = name.split('_')[-1]
            if 'V' in s:
                print('%s = %2.3f V' %(name, va))
            else:
                print('%s = %2.3f A' %(name, va))
        temp = convert_temp(PIB_HP_BOOST_TEMP)
        print('PIB_HP_BOOST_TEMP = %2.1f C' %temp)
        for name, va in zip(misc_names, misc):
            print('%s = %u' %(name, va))
        print('LDC = %u'%LDC)



if __name__ == "__main__":
    try:
        fname = sys.argv[1]
        satname = sys.argv[2]
    except:
        print('%s <raw data filename> <satname>'%sys.argv[0])
    else:
        data_parse(fname, satname)
```

LIST OF REFERENCES

[1] G. Minelli, M. Karpenko, I. M. Ross, and J. Newman, "Autonomous operations of large-scale satellite constellations and ground station networks," presented at AAS/AIAA Astrodynamics Specialist Conf., Stevenson, WA, USA, 2017.

[2] J. Leone III, "CubeSat pass quality analysis and predictive model," M.S. thesis, Space System Academic Group, NPS, Monterey, CA, USA, 2018.

[3] A. Ortega, "UHF space-to-ground interface control document." Release number 01, 01 Nov 2017. Tyvak, Irvine CA.

[4] Tyvak, "Data expression space-to-ground interface control document." 01 Nov 2017. Tyvak, Irvine, CA.

[5] J. M. Roehrig, "Development of a versatile groundstation utilizing software defined radio," M.S. thesis, Space System Academic Group, NPS, Monterey, CA, USA, 2016.

[6] M. Chepponis-K3MC and P. Karn-KA9Q, "The kiss tnc: A simple host-to-tnc communications protocol," in *ARRL 6th Computer Networking Conference*, 1987.

[7] Announcing the Advanced Encryption Standard (AES). Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, 2001.

[8] L. C. Washington and W. Trappe, *Introduction to Cryptography with Coding Theory*, 2nd ed. Pearson Education, 2006.

[9] D. A. McGrew and J. Viega, "The Galois/counter mode of operation GCM." 31 May 2005. Available: https://pdfs.semanticscholar.org/b4c4 /66e7158c158fb513b729d6302521017d72fa.pdf. [Accessed: 01-May-2019].

[10] M. Dworkin, Recommendation for block cipher modes of operation: Galois/Counter mode (GCM) and GMAC. Gaithersburg, MD: U.S. Dept. of Commerce, National Institute of Standards and Technology, 2007.

[11] A. Morrison, "Amateur radio satellite file transfer protocol (ARSFTP)," Jun-2012. [Online]. Available: https://digitalcommons.calpoly.edu/cgi /viewcontent.cgi?referer=&httpsredir=1&article=1179&context=eesp. [Accessed: 01 May 2019].

[12] M. Zadka, "Getting started with Python's cryptography library," *Opensource.com*, 08-Apr-2019. [Online]. Available: https://opensource.com/article/19/4 /cryptography-python. [Accessed: 01 May 2019].

[13] “Modern modes of operation for symmetric block ciphers,” *Modern modes of operation for symmetric block ciphers – PyCryptodome 3.8.1 documentation*. Available: https://pycryptodome.readthedocs.io/en/latest/src/cipher/modern.html#gcm-mode. [Accessed: 01 May 2019].

[14] “TCP/IP and tcpdump pocket reference guide.” [Online]. Available: http://www.cs.mun.ca/~yzchen/bib/tcpip.pdf. [Accessed: 01 May 2019].

[15] “RFC 760 - DoD standard internet protocol,” faqs.org. [Online]. Available: http://www.faqs.org/rfcs/rfc760.html. [Accessed: 01 May 2019].

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California