Flight Software Architectures for Safe and Sustainable Missions

Ethan Chee
Electrical and Computer Engineering
University of Hawaii at Mānoa
Honolulu, HI

***Abstract*—Flight software implementation into satellites is critical for a mission's success. This paper discusses the requirements for a reliable flight software framework and how current frameworks implement these standards. Ideally, a safe and sustainable framework must have modularity, portability, a small footprint, unit testing capabilities, a standardized packet protocol, and documentation. Of the existing frameworks analyzed, COSMOS and F Prime were implemented into the HyTI and Artemis CubeSat kit. Additionally, AES encryption was implemented and analyzed with the COSMOS framework and tested with flight hardware for the HyTI mission. Results show that cryptography does not affect the likelihood of packet loss. Since COSMOS was determined to not meet all of the flight software requirements defined by the ideal architecture, the framework will be improved after the delivery of HyTI.**

***Index Terms*—Flight Software (FSW), Embedded Systems, Real-Time Operating Systems (RTOS), Cybersecurity, Advanced Encryption Standard (AES)**

I. Introduction

A variety of existing flight software frameworks were analyzed to inspire a standardized architecture that all frameworks must adapt for safe and sustainable missions. Two spacecraft were focused on throughout the research that implemented COSMOS, F Prime, and a custom-designed baremetal FSW model. To ensure a secure communication between the spacecraft and trusted ground station(s), encryption algorithms were analyzed, implemented, and tested with the COSMOS framework.

A. Flight Software Requirements

Flight software is a critical component of all spacecraft operations. Different satellites can have a wide range of hardware, and thus the flight software must be engineered to support those systems. From [1], some of the essential components of a flight software framework must have sufficient technical documentation, including training for ease of use, a small footprint, modularity, portability, and unit testing capabilities. The same criteria is applied to ground software as it must be able to interact with various ground hardware as well as the flight software.

B. CCSDS Recommendations

The Consultative Committee for Space Data Systems (CCSDS) recommends standards for interfaces and protocols for space missions for communication effectiveness and compliance. The complete list of CCSDS recommended standards can be found in [2], but one of the important standards to consider is the packet protocols for space telecommunications. Having a standardized data structure to create, store, and transfer variable-length data ensures packet identification and source/destination determination. Packet identifiers include, but not limited to, version numbers, type IDs, sequencing, and data length. Figure 1 shows an example of a space packet header that follows the CCSDS recommendations.

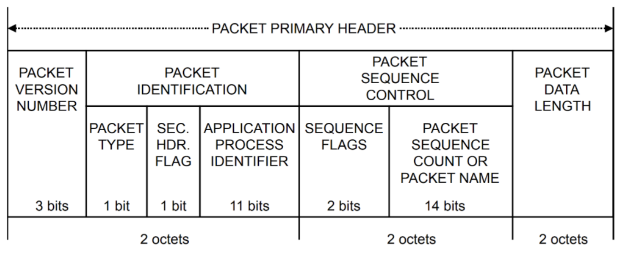


Fig. 1. Sample CCSDS Header

II. Existing Frameworks

Various flight software frameworks exists that attempt to fulfill these requirements for stable space research and exploration: GERICOS, NASA cFS, NASA F Prime, COSMOS, etc.

A. GERICOS

The GERICOS flight software framework is component-based and architectured into three layers: Core, Blocks, and Drivers [3]. The core layer interfaces with the Operating System (OS) to provide features for task execution, threading, synchronization, buffer management, etc. The block layer defines components for standard flight software functionalities, including command handling, telemetry, operation mode management, CCDS protocol management, etc. Finally, the driver layer provides capabilities to interface with the hardware. [3] describes their driver layer specific to a specific processor family, the COTS IP cores, such as the UT699 LEON3-FT CPU, but the concept of having a driver layer can be expanded to support various peripherals available to the hardware.

B. NASA cFS

NASA's core Flight System (cFS) is a component-based architecture that was designed to be mission-independent. It

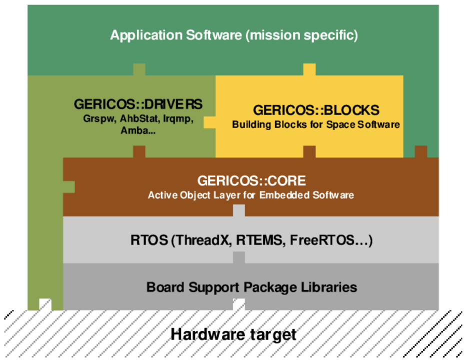

Application Software (mission specific)
GERICOS::DRIVERS
Grspw, AhbStat, Irqmp, Amba...
GERICOS::BLOCKS
Building Blocks for Space Software
GERICOS::CORE
Active Object Layer for Embedded Software
RTOS (ThreadX, RTEMS, FreeRTOS...)
Board Support Package Libraries
Hardware target

Fig. 2. GERICOS Architecture Layers

is also independent from any operating system or hardware, and architectured into three layers: an OS abstraction layer, core layer, and application layer [4]. The application layer contains flight-specific and mission-specific functionalities, such as event scheduling, timing, file management, logging, etc. The core layer manages the applications in the cFS layer and interfaces with the OS abstraction layer to communicate with the kernel and hardware being used. This ensures that any functionalities within the core and application layers are independent from the OS and hardware, as the OS abstraction layer can be replaced with different abstraction layers to meet the requirements of a different OS, or even a RTOS.

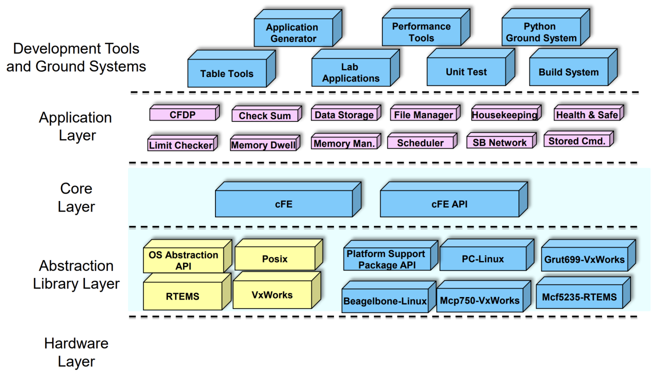


Fig. 3. NASA cFS Architecture Layers

C. F Prime

Key features of the F Prime architecture described in [5] include components, ports, automated code generation, and unit tests. F Prime has three different types of components, but they all contain data fields defined by the user and how they are operated on. The first is an active component which is multi-threaded and contains a queue to process incoming requests. The second is a passive component that behaves like a general C++ class, providing user-defined functions, similar to a library. The third is a queued component that was designed to perform periodic operations, such as beacon data. Data in a component can be viewed as private variables of a class. Multiple instances of a component can exist in the flight software. Ports are tied with defining data within these components, as they create a communication link with other components. For example, one component can have an input port to a GPIO pin or an output port for telemetry data. These components and ports provide a sense of modularity and portability, which are critical requirements of an effective flight software architecture.

F Prime also has an OS abstraction layer for common OS features, such as threading, synchronization, queues, file management, timers, and clocks [5]. This allows the framework to be supported for different embedded system hardware as well as operating systems. An F Prime component would call upon the functions defined in the OS abstraction layer rather than functions defined in standard packages. That way, these functions can be easily replaced to meet the requirements of the hardware and OS without interfering with F Prime's core components nor user-defined components.

Another key feature of the F Prime framework is automated code generation. Upon defining a component with its ports and data fields, building the project generates a C++ header and class for that component that contains a template for component initialization and execution. This makes usability much simpler as developers do not have to design the back-end of F Prime's architecture for each component.

F Prime also supports unit testing, which is another key component to a flight software framework. More specifically, these unit tests were designed to test the ports of a component to simulate a communication link between two instances where the flight software instance contains an input port while the test instance contains an output port, and/or vice versa. End users can modify these unit tests to suit their requirements, such as processing such data being sent to the unit test or sending test data to the flight software instance.

D. COSMOS

The COSMOS flight software framework was designed for space flight and ground mission operations. It provides a set of custom libraries that allow different components or subsystems, such as a satellite, ground network, or payload, to communicate with each other. Key features of the COSMOS architecture described in [6] include nodes such as satellites, ground stations, servers, rovers, etc., and agents which are executed on said nodes. They are processes that are capable of sending and receiving packets to and from other agents.

To command an agent, agents can register Agent Requests and point it to a function which would be executed when that request is received. All requests of an agent are stored in a statically allocated array.

COSMOS also uses the concept of channels, which are packet queues that are available to multi-threaded processes. Channels are most beneficial in systems that require asynchronous operations, such as radio communications. These features satisfy the modularity and portability of flight software architectures since several agents can be created for different applications. These agents essentially create a local network within the node to communicate data between them using the COSMOS PacketComm protocol.

One key feature of COSMOS is the design of its PacketComm protocol, a packet structure used to relay data on board the spacecraft, across multiple spacecraft, and between a spacecraft and ground station network. The PacketComm protocol does not directly replicate the CCSDS recommended packet header shown in Figure 1, but still successfully creates a structured communication protocol for easy implementation into spacecraft. A PacketComm packet consists of a type ID field that indicates the purpose of the packet (i.e. commanding, telemetry, etc.), a data size packet that specifies the variable-length (in bytes) of the packet payload, node origin/destination fields to determine the sending/receiving agent of all COSMOS agents in the network, and channel in/out fields to specify which channel the packet originated from and is destined to be handled (i.e. radio channel, EPS channel, etc.). Following the header is the packet payload that contains custom data, such as telemetry values or raw bytes for file transfers. Finally, the PacketComm packet ends with a 2 byte cyclic redundancy check (CRC) checksum to ensure that the packet was not corrupted during transmission. The complete PacketComm packet is shown in Figure 4.

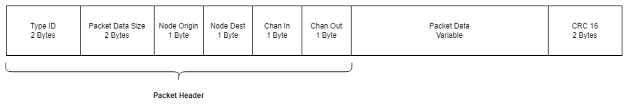


Fig. 4. PacketComm Protocol

III. The Ideal FSW Architecture

An ideal flight software framework must provide modularity in a way where a component can be replaced without disrupting the flow and operations of other components. This naturally creates ”layers” within the architecture where the highest layer is the application layer that contains user-defined operations, such as radios, actuators, sensors, etc. This layer can also contain core FSW framework applications, such as packet handling, commanding, telemetry, debug logging, etc. Below this application layer would be an OS layer, which contains components for OS specific operations, including file systems, task schedulers, network managers, memory management, device drivers (GPIO, I2C, SPI, UART), timers, system health, etc. In the case for baremetal applications, or in other words, embedded systems that do not run any operating system, the OS abstraction layer will either simulate certain functionalities that are unsupported, or use stub functions (functions that are defined but have no behavior) in its place. An example of a simulated function is a task scheduler for single-threaded processors. To simulate multi-threading, a one dimensional vector can be used to store tasks where each task takes turns to execute for a specified clock period, and then yield to the next task in the vector.

The final layer is the hardware layer that the OS layer interacts with. The goal for a FSW framework is to be deployable on any type of hardware. The proposed FSW architecture layer to be standardized for all FSW frameworks is shown in Fig. 5.

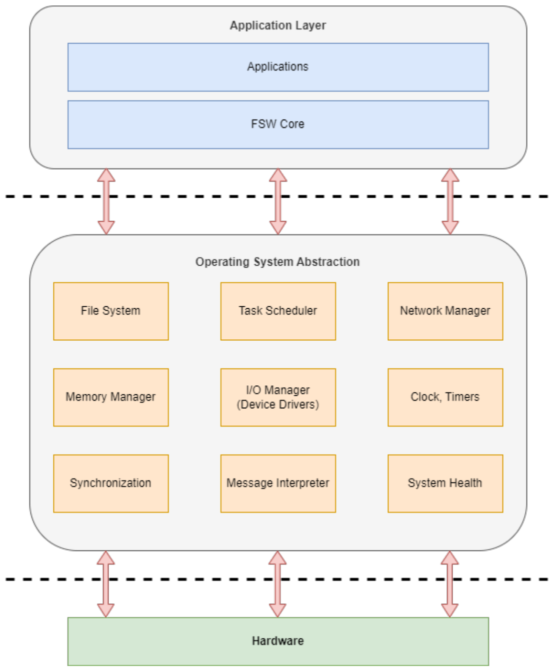


Fig. 5. Ideal FSW Architecture

For the FSW framework to be successfully deployed on various architectures and/or operating systems, this OS layer can be swapped with one compatible for the desired hardware. Example OS layer implementations include Posix calls for UNIX based machines, Arduino functions for select embedded microcontrollers, or system calls for real-time operating systems (RTOS) such as FreeRTOS or ZephyrRTOS. All libraries used within the application layer do not call to any third-party libraries, such as stdlib or stdio, since not all architectures support these packages.

Many existing frameworks follow this architecture, such as GERICOS, CFS, and F Prime, which were previously discussed.

Other essential design considerations for the FSW to be “safe” involve memory management. The software must ensure that it does not have memory leaks nor is it capable of overflowing the available memory of the spacecraft as that would lead to a mission-ending scenario. Therefore, dynamic memory allocations must be avoided. The memory manager of the FSW must statically allocate memory buffers at compile time, and any “allocations” performed during run-time would fill those statically allocated buffers. If the requested allocation size is greater than the available buffer size, an error would assert rather than causing a segmentation fault. Many standard C libraries use dynamic memory allocation which should be considered, so those functions should be avoided when possible. Instead, select functions can be replaced with custom functions within the core framework that does not dynamically allocate memory.

The core components of a FSW framework must also be mission-independent. In other words, the framework creates a skeleton that end users can build off of and expand to meet their mission-specific requirements. If an end user would add a feature to their spacecraft, they should not have to edit the core framework. Example mission-independent core components include, but not limited to, command handling (does not include the registration of commands), telemetry, file uplink and downlink, or packet reconstruction and checksum validations.

IV. Security Implications

With the increase in popularity and production of satellites, particularly SmallSats, there is an increase in security risks relating to satellite telecommunications. Satellite telecommunications are the use of satellites to provide a means of communication across one or several ground stations located on Earth's surface. This link is achieved through transceiver technology. Radios can operate at a range of frequencies, but the receiving radio must be operating at the same frequency as the transmitting radio to successfully read the incoming packets. The use of software defined radios (SDRs) makes it easier to detect packets at a range of frequencies as they can be easily programmed and modified accordingly. The security risk is that satellite telecommunications are connectionless; a prior handshake is not established before packets are downlinked, and therefore, all packets being transmitted from a satellite are essentially being broadcasted to any available SDRs in range listening within the operating frequency range. If the satellite collects sensitive data and telemetry needed to be downlinked to a ground station, these packets should be encrypted such that any unintended recipients will not be able to decrypt the data. The same security risk can be applied to packets being uplinked.

The goals for satellite telecommunication security are to prevent malicious attackers from sniffing sensitive data, as well as rebroadcasting commands that will result in a denial-of-service (DoS) attack or even end the mission entirely.

Radio Frequency (RF) attacks are to be expected, so a flight software framework should be designed to handle such attacks. One of the primary mitigation techniques within F-Prime involves a command dispatcher/handler, which ensures that all commands received are genuine. All commands are registered with an opcode and placed in a registration table upon initialization of the FSW [5]. If any commands are decoded but are not mapped to the registration table, that command packet will become invalid and dropped. Likewise, if the FSW is unable to deserialize the packet to retrieve the opcode, this will be flagged as a malformed packet and dropped.

For any general packet frame, including commanding, telemetry, or file transfers, they are commonly verified with a cyclic redundancy check (CRC) checksum footer which is calculated using its preceding bytes. In the event that a malicious attack was conducted to modify a packet, or insert random bytes into a captured packet frame, the checksum will fail and the packet will be dropped. The current F-Prime FSW and GDS implementations utilize checksums to ensure any packet communications are legitimate.

Any packet analysis protocols, such as decoding, decrypting, or CRC checking, should minimize computation time to allow processing time to be allocated to other incoming packets. This ensures that if RF noise is present, the receiving station is capable of keeping up with the rate at which all packets are being transmitted to it.

A. Radio Encryption

Encryption algorithms currently exist that are capable of encrypting strings of text, particularly using the Advanced Encryption Standard (AES). All AES algorithms share a common encryption technique: the use of private keys. However, some algorithms still have security risks where these private keys can be easily cracked by malicious attackers.

Various encryption algorithms were explored, including Electronic Codebook (ECB), Cipher Block Chaining (CBC), Counter (CTR), and Galois/Counter Mode (GCM).

1) AES-ECB: AES-ECB [7] is the simplest implementation of AES-256 encryption as it only uses a private key to encrypt and decrypt blocks of text. This method is fast as it can perform these operations in parallel. However, this method does not protect against accidental modification or malicious tampering of bytes. The ciphered data is easy to decipher because only a single key is used for every block size in a packet. With experienced cryptography hackers, they are capable of identifying a pattern with encrypted data, and thus determine the private key used for encryption and decryption. Due to the numerous security risks AES-ECB proposes, this method was not considered.

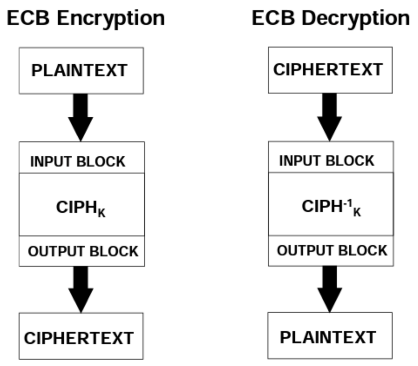


Fig. 6. AES-ECB Encryption and Decryption Diagram

2) AES-CBC: The second algorithm explored was AES-CBC [7], which is a better solution than ECB as the algorithm uses both a private key and initialization vector (IV). Each block size is encrypted using a hash generated from the previous block size and the private key, thus making the encrypted data more difficult to crack. For the first block size, it uses the IV instead since there is no previous block. This method is slower as the algorithm performs XOR operations sequentially. Additionally, this algorithm also requires byte padding which is not ideal because radio communications oftentimes have limited buffer sizes.

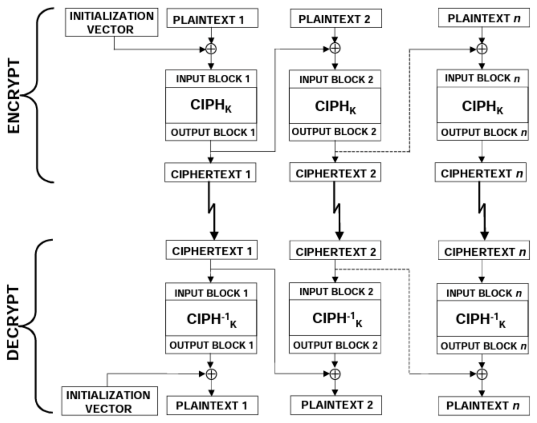


Fig. 7. AES-CBC Encryption and Decryption Diagram

3) AES-CTR: The third algorithm explored was AES-CTR [7]. Much like CBC, CTR uses both a private key and IV. The key feature of CTR is the implementation of a counter that increments for each block size. This increases the difficulty of reverse engineering the private key as each block size uses different hashes to encrypt or decrypt. Different from CBC, this algorithm can encrypt and decrypt blocks in parallel. CTR also does not do byte padding which is a benefit for radio packets with limited buffer sizes. The only disadvantage with this algorithm is that there is still a systematic way of encryption since the counter increases by a constant amount, usually by 1, for each block. Nonetheless, this is still an acceptable method of encryption.

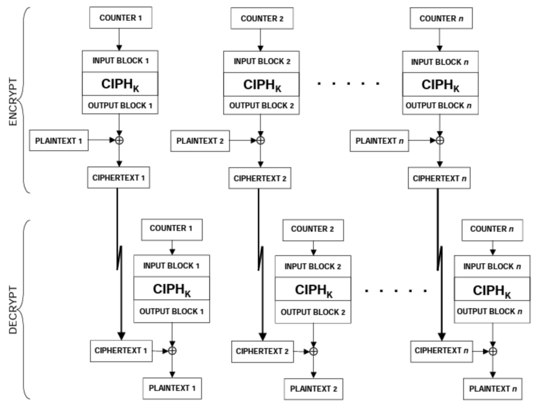


Fig. 8. AES-CTR Encryption and Decryption Diagram

4) AES-GCM: The final algorithm explored was AES-GCM [8] which is an adaptation of the AES-CTR algorithm. GCM uses the same encryption and decryption as CTR with the addition of an authentication token, which is essentially a 16 byte checksum using the GHASH protocol. However, this checksum is not practical for satellite use because the 16 byte checksum is too large for radio packets, especially for low-cost radios with significantly smaller buffer sizes. Modern flight software frameworks have integrated checksums into their protocol, such as COSMOS's PacketComm and FPrime's Framer/Deframer protocols. Because of this, the GHASH authentication token for GCM was not a priority. Due to the effectiveness of AES-GCM and low risk of malicious tampering, this algorithm was used for encryption and decryption of commanding and data frames.

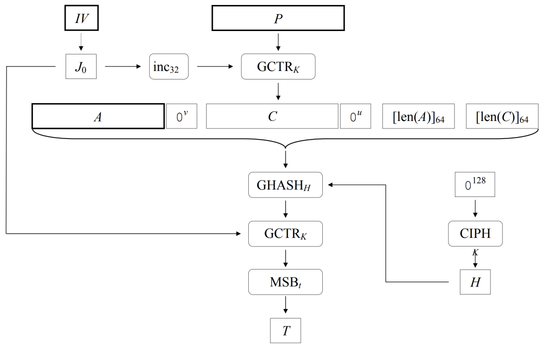


Fig. 9. AES-GCM Encryption Logic

The encryption algorithm is shown in Algorithm 1. The decryption algorithm is shown in Algorithm 2.

Algorithm 1 AES-GCM Encryption Implementation

```
Input: int $ivSize$; String $plaintext$, $key$
Output: String $encrypted$
1: $iv \leftarrow$ RandBytes($ivSize$)
2: $seq \leftarrow$ GetSequenceNum()
3: $plaintext \leftarrow$ concat($seq, plaintext$)
4: $ciphertext \leftarrow$ Encrypt($plaintext, key, iv$)
5: $encrypted \leftarrow$ concat($ciphertext, iv$)
```

Algorithm 2 AES-GCM Decryption Implementation

```
Input: int $ivSize$; String $encrypted$, $key$
Output: String $plaintext$
1: $iv \leftarrow encrypted[encrypted.len - ivSize, encrypted.len]$
2: $ciphertext \leftarrow encrypted[0, encrypted.len - ivSize]$
3: $plaintext \leftarrow$ Decrypt($ciphertext, key, iv$)
```

The AES-GCM algorithm can be implemented into satellites by integrating a library capable of encrypting and decrypting a vector of bytes. However, the security risk of rebroadcasting radio packets still exists; any attacker can intercept a command packet, such as a reboot command, and continuously rebroadcast it at the same frequency ultimately stopping the functionality of the satellite. To prevent this, a sequence number can be added to the protocol, much like how the TCP protocol operates. Packets can only be accepted if the incoming sequence number is greater than the previous packet's sequence number. To also prevent attackers from choosing an arbitrarily large number, a windowing can also be implemented where the new sequence number can only be a certain value greater than the previous. This sequence number should also be encrypted using the AES-GCM algorithm to prevent any modifications to this value. For example, a malicious attacker cannot simply append a really large sequence number to bypass the protocol because they do not have the private key to properly encrypt that sequence number to be decrypted by the satellite.

There are various ways to choose the sequencing value: (1) start at zero on boot and increase for every transmitted packet; (2) select a number at random; and (3) set the sequence value to the current time, in seconds, since January 1, 1970. All three of these options has its advantages and disadvantages. For instance, starting at zero will result in difficult tracking if the spacecraft has been online for a very long time and will allow for 4,294,967,296 packets before an integer overflow with an unsigned 32 bit sequence value. However, it is very easy to guess the sequence number at boot time. Selecting a number at random solves this issue as it makes it practically impossible to guess the previous sequence number, but it will result in less packet transmissions before an integer overflow.

An example that utilizes sequencing using the AES-GCM encryption and decryption algorithms is shown in Algorithm 3.

Algorithm 3 Encryption and Decryption Using Sequence Numbers

```
Input: int $seq$, $ivSize$, $window$; String $frame$, $key$
Output: String $data$
1: $plaintext \leftarrow$ concat($seq, frame$)
2: $encrypted \leftarrow$ GCM_Encrypt($plaintext, key, ivSize$)
3: $decrypted \leftarrow$ GCM_Decrypt($encrypted, key, ivSize$)
4: $newSeq \leftarrow decrypted[0, 4]$
5: if $newSeq > prevSeq$ and $newSeq \leq prevSeq + window$ then
6:     $data \leftarrow decrypted[4, decrypted.len]$
7:     $prevSeq \leftarrow newSeq$
8: end if
```

B. Encryption Restrictions

While packet encryption is essential for security and privacy, it is important to be aware of packet frame requirements for radio hardware. Specific bytes of a frame cannot be encrypted as the radio needs those bytes as markers (i.e. indicating a start and end of a packet). This limitation encourages the necessity for flight software modularity, which will allow developers to add an encryption component that will encrypt or decrypt select bytes of a frame without modifying the software of other components. For instance, suppose the current pass of a packet consists of collecting telemetry, then packetizing into a frame, and sending it out the radio. If encryption were to be implemented, none of those components would be modified, but instead, an encryption component would be placed in between the packetizing and radio components. The new pass would be: collecting telemetry, packetizing into a frame, encrypting the frame, and finally sending it out the radio.

V. Implementation

A. Hyperspectral Thermal Imager (HyTI)

1) FSW Architecture: HyTI is a 6U CubeSat that uses the COSMOS flight software framework for its mission. The satellite contains two on-board computers, resulting in two COSMOS agents: iOBC and Unibap. To communicate with a ground station, a third agent is added to this link: a ground agent. The iOBC agent incorporates several channels to handle core functionalities, including the S-Band radio, UHF radio, electrical power system (EPS), Attitude Determination And Control System (ADCS), Unibap communications, and commanding/telemetry. The Unibap agent also contains several channels to handle core functionalities, including S-Band and X-Band radios, camera operations, iOBC communications, and commanding/telemetry.

2) Encryption: Algorithms 1-3 guided the development of a COSMOS module that incorporates OpenSSL and COSMOS to allow easy encryption and decryption of any vector of bytes, but more specifically, PacketComm packets.

The code snippet below shows the header file of the Crypto class.

```
class Crypto
{
public:
 static std::atomic<uint32_t> prev_sequence;

Crypto();
Crypto( string key);
~Crypto();
 int32_t setKey( string key);
 int32_t setIV( vector<uint8_t> iv);
 int32_t randomizeIV(uint8_t size);
 int32_t setSequenceWindow(uint8_t windowLo, uint8_t windowHi);
 int32_t encrypt( vector<uint8_t> plaintext_str, vector<uint8_t> &ciphertext_str, size_t iv_size);
 int32_t decrypt( vector<uint8_t> ciphertext_str, vector<uint8_t> &plaintext_str, size_t iv_size);

private:
bool hasKey();

EVP_CIPHER_CTX *ctx;
 string key = "";
 vector<uint8_t> iv;
 uint8_t windowLo;
 uint8_t windowHi;

bool initialized = false;
};
```

An important feature in this class includes a static atomic instance of the previous sequence number. The idea behind making this a static instance is to maintain a common sequence value across multiple Crypto objects, usually when more than one radio is used on the satellite. Therefore, an atomic instance is required to ensure synchronization across multiple radio channels attempting to decrypt and modify the sequence number.

The encryption implementation using OpenSSL and Algorithm 1 is shown below.

```
int32_t Crypto::encrypt( vector<uint8_t> plaintext_str, vector<uint8_t> &ciphertext_str, size_t iv_size)
{
if (! this->hasKey())
{
return -1;
}

int ciphertext_len ;
vector<uint8_t> buf(plaintext_str.size() + 4);
vector<uint8_t> plaintext;

 plaintext.resize(4);
uint32to(decisec(), plaintext.data());
 plaintext.insert( plaintext.end(), plaintext_str.begin(), plaintext_str.end());

randomizeIV(iv_size);

EVP_CIPHER_CTX_ctrl(this->ctx, EVP_CTRL_GCM_SET_IVLEN, this->iv.size(), NULL);
EVP_EncryptInit_ex(this->ctx, NULL, NULL, (unsigned char *)this->key.c_str(), this->iv.data());
EVP_EncryptUpdate(this->ctx, buf.data(), &ciphertext_len , plaintext . data () , plaintext . size () ) ;

 ciphertext_str . resize ( ciphertext_len ) ;
memcpy(ciphertext_str . data () , &buf[0], ciphertext_len ) ;

 ciphertext_str . insert ( ciphertext_str .end() , this->iv.begin(), this->iv.end());

return 0;
}
```

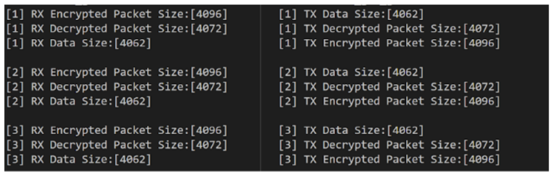
This function also incorporates the COSMOS time library to generate a systemized sequence number that is simple to monitor. The decisec() function returns a 4 byte unsigned integer of the 1/10th seconds (deciseconds) since the beginning of the decade.

The decryption implementation using OpenSSL and Algorithm 2 is shown below.

```
int32_t Crypto :: decrypt( vector <uint8_t> ciphertext_str, vector <uint8_t> &plaintext_str, size_t iv_size )
{
if (! this->hasKey())
{
return -1;
}

int plaintext_len ;
vector <uint8_t> buf(ciphertext_str . size () + 4);
vector <uint8_t> plaintext;

this->iv.clear() ;
for ( size_t i = ciphertext_str . size () - iv_size ; i < ciphertext_str . size () ; i++)
{
this->iv.push_back( ciphertext_str [ i ]) ;
}

 plaintext . insert ( plaintext .begin () , ciphertext_str .begin () , ciphertext_str .end() - this->iv.size()) ;

EVP_CIPHER_CTX_ctrl(this->ctx, EVP_CTRL_GCM_SET_IVLEN, this->iv.size(), NULL);
EVP_DecryptInit_ex(this->ctx, NULL, NULL, (unsigned char *)this->key.c_str(), this->iv.data());
EVP_DecryptUpdate(this->ctx, buf.data(), &plaintext_len , plaintext . data () , plaintext . size () ) ;

vector <uint8_t> sequence(4);
memcpy(sequence.data(), &buf[0], 4) ;

 plaintext_str . resize ( plaintext_len - 4);
memcpy( plaintext_str . data () , &buf[4], plaintext_len - 4);

uint32_t curr_sequence = uint32from(sequence. data () ) ;
uint32_t prev = ( this->prev_sequence.load() < this->windowLo) ?
  this->prev_sequence.load() : this->prev_sequence.load() - this->windowLo;


// windowHi not a good idea when generating sequence number based on time.
if (curr_sequence < prev /*|| curr_sequence > prev_sequence.load() + windowHi*/)
{
printf ("Prev: %d, Curr: %d\n", prev_sequence.load(), curr_sequence) ;
return COSMOS_GENERAL_ERROR_BAD_ACK;
}

this->prev_sequence = curr_sequence;
return 0;
}
```

To test the cryptography library, a simple client-server was developed using sockets on the localhost network. The client simulates a satellite downlinking packets to a ground station, simulated as the server. The test included encrypting a 4072 byte PacketComm packet with a 4062 byte data frame appended with a 16 byte IV vector, 4 byte sequence number, and 4 byte ASM protocol header to send a packet with a total length of 4096 bytes. The server would decrypt this packet to retrieve the original 4072 PacketComm packet and 4062 byte data frame. The results can be seen in Figure 10 where the left terminal is the server and the right terminal is the client. The client was successfully able to transmit an encrypted ASM packet over the socket port in which the server is listening to and successfully decrypted it to retrieve the original raw data.

3) MC3 Network Encryption/Decryption Testing: To ensure the encryption and decryption is operational with hardware in the loop, the cryptography library was tested using S-Band, X-Band, and UHF radios communicating with the MC3 ground station network located at the Naval Information Warfare Center in Pearl City.



```
[1] RX Encrypted Packet Size:[4096]      [1] TX Data Size:[4062]
[1] RX Decrypted Packet Size:[4072]      [1] TX Decrypted Packet Size:[4072]
[1] RX Data Size:[4062]                  [1] TX Encrypted Packet Size:[4096]

[2] RX Encrypted Packet Size:[4096]      [2] TX Data Size:[4062]
[2] RX Decrypted Packet Size:[4072]      [2] TX Decrypted Packet Size:[4072]
[2] RX Data Size:[4062]                  [2] TX Encrypted Packet Size:[4096]

[3] RX Encrypted Packet Size:[4096]      [3] TX Data Size:[4062]
[3] RX Decrypted Packet Size:[4072]      [3] TX Decrypted Packet Size:[4072]
[3] RX Data Size:[4062]                  [3] TX Encrypted Packet Size:[4096]
```

Fig. 10. Encryption Testing using a Client-Server Setup

The S-Band radio incorporates the ASM header as well as the CCSDS header. These bytes cannot be modified as they are used as markers for packet identification, and thus cannot be encrypted. The PacketComm packet gets encrypted first, and the CCSDS/ASM protocols are applied to the encrypted PacketComm packet. The X-Band radio also uses the ASM header which cannot be modified by encryption. Additionally, all packets to be transmitted from the X-Band radio must fill its buffer of 4096 bytes to successfully transmit. As a result, even after encryption, the packet frame is appended with null terminating characters (hex 0x00). However, the radio ignores packets if there are consecutive sequences of binary 0s or 1s. Since encryption could result in the packet having consecutive 0s or 1s, Galois LFSR scrambling must be applied after encryption and padding to ensure the full 4096-byte packet frame can successfully transmit. Much like the S-Band radio, the PacketComm packets gets encrypted first, the ASM header is inserted to the start of the encrypted packet, then padding is applied to the end of the packet to reach a length of 4096 bytes, and Galois LFSR scrambling is applied to the entire packet excluding the ASM header. Finally, the UHF Lithium-3 radio incorporates the AX.25 protocol which uses the High-Level Data Link Control (HDLC) protocol. It inserts a 16 byte header to the beginning and a 2 byte CRC checksum to the end of the packet. The difference between the Lithium-3 radio compared to the S-Band and X-Band is that the AX.25 protocol is handled at the hardware level instead of software. Therefore, the flight software can simply transmit an encrypted PacketComm packet to the radio without the AX.25 header as the radio handles that automatically. However, the ground station would need to be able to decode an AX.25 packet manually. The AX.25 packet would have to be verified by using the provided checksum. If successful, the encrypted packet can be obtained by stripping away the first 16 and last 2 bytes of the packet, and then decrypting to receive the PacketComm packet.

The following are the steps for downlinking encrypted packets for all three frequency bands for HyTI:

S-Band Downlink Encryption:

1) Construct PacketComm packet
2) PacketComm Wrap
3) Encrypt wrapped PacketComm packet
4) Insert CCSDS header to beginning of packet
5) Insert SATSM header 0x35, 0x2E, 0xF8, 0x53 after CCSDS header
6) Insert encrypted wrapped PacketComm packet after

SATSM header

X-Band Downlink Encryption:

1) Construct PacketComm packet
2) PacketComm Wrap
3) Encrypt wrapped PacketComm packet
4) Insert ATSM header 0x1A, 0xCF, 0xFC, 0x1D to beginning of packet
5) Insert encrypted wrapped PacketComm packet after ATSM header.
6) Add padding to ensure a packet length of 4096 bytes
7) Apply Galois LFSR scrambling to packet excluding ATSM header.

UHF Downlink Encryption:

1) Construct PacketComm packet
2) PacketComm Wrap
3) Encrypt wrapped PacketComm packet

Uplink encryption behavior varies depending on the SDR used. In the case for the MC3 network, all uplink packets were constructed using the AX.25 protocol, with an additional KSat header.

S-Band and UHF Uplink Encryption:

1) Construct PacketComm packet
2) PacketComm Wrap
3) Encrypt wrapped PacketComm packet
4) Insert 16 byte AX.25 header to beginning of packet
5) Insert encrypted packet after AX.25 header
6) Calculate 2 byte checksum and insert after encrypted packet
7) Apply HDLC protocol
8) Insert KSat header to beginning of packet

To validate whether a packet was successfully encrypted and decrypted, the same packet type of PacketComm type DataRadioTest (0x701) was used with arbitrary data. For packets using the AX.25 protocol, the checksum must pass. Likewise, all packets must pass the PacketComm CRC checksum check. All received packets must successfully decode the packet and recognize it as a DataRadioTest (0x701) packet.

Snippets of the received packets for S-Band, X-Band, and UHF are shown below:

S-Band Downlink at 625k data rate:

```
[6.439]  Receive [Name=SELF, Enabled=0, Size=227 Age=0.000065 Length=0 PacketCnt=3617 ByteCnt=698081] [Type
         =0x701, Size=183 Node=[254 255] Chan=[0 0]] ab 24 19 00 a9 00 00 00 55 0e 00 00
Test: Good: MET: 60244.941807 Test_Id: 1647787 Packet_Id: 3669 Good: 3617 Skip: 53 Size: 0 Crc: 0 Bytes: 618507
         Count: 3670 Seconds: 1.241647 Speed: 498134.105793
```

S-Band Downlink at 2.5M data rate:

```
[100.961]  Receive [Name=SELF, Enabled=0, Size=227 Age=0.000063 Length=0 PacketCnt=3671 ByteCnt=708503] [
         Type=0x701, Size=183 Node=[254 255] Chan=[0 0]] 1a 43 70 00 a9 00 00 00 ff ff ff ff
Test: Complete: MET: 60237.852308 Test_Id: 7357210 Packet_Id: 3670 Good: 3671 Skip: 0 Size: 0 Crc: 0 Bytes:
         627741 Count: 3671 Seconds: 62.863406 Speed: 9985.793445
```

S-Band Uplink:

```
Good: MET: 60237.839046 Test_Id: 7230519 Packet_Id: 1017 Good: 1017 Skip: 1 Size: 0 Crc: 0 Bytes: 142380 Count:
         1018 Seconds: 188.887501 Speed: 73
raw packetized len: 182
decrypted packetized len: 162
Good: MET: 60237.839047 Test_Id: 7230519 Packet_Id: 1018 Good: 1018 Skip: 1 Size: 0 Crc: 0 Bytes: 142520 Count:
         1019 Seconds: 188.998431 Speed: 70
raw packetized len: 182
decrypted packetized len: 162
```

X-Band Downlink at 3M data rate:

```
[6.806]  Receive [Name=SELF, Enabled=0, Size=4096 Age=0.000067 Length=0 PacketCnt=4932 ByteCnt=20083104] [
         Type=0x701, Size=4062 Node=[254 255] Chan=[0 0]] 8d 23 13 00 d0 0f 00 00 43 13 00 00
packetized size: 4096
encrypted size: 4092
wrapped size: 4072
Test: Good: MET: 60244.941147 Test_Id: 1254285 Packet_Id: 4932 Good: 4933 Skip: 0 Size: 0 Crc: 0 Bytes:
         19978650 Count: 4933 Seconds: 1.519528 Speed: 13147916.615011
```

UHF Uplink:

```
Update: Good: MET: 60244.112155 Test_Id: 954649 Packet_Id: 123 Good: 83 Skip: 40 Size: 0 Crc: 0 Bytes: 14774
         Count: 123 Seconds: 68.856620 Speed: 214.561790
Got slip packet
2023−10−27T02:41:31Z 584.008 1.213 packet_id: 124 Packets: 117 Bytes_total: 23400
```

Based on these results, encryption and decryption was successfully implemented on the flight software and ground software as the DataRadioTest (0x701) packets successfully decoded. However, there were some packet losses during the tests. Table I shows a summary of the packet losses for all encryption/decryption tests. Packets that were successfully decoded were marked as "Good" packets, while malformed packets were skipped.

TABLE I
PACKET LOSSES FOR ENCRYPTED TRANSMISSIONS

Radio	Direction	TX Rate	Good	Skipped	% Loss
UHF	Up	—	85	40	32%
UHF	Down	9.6K	17	31	64.6%
SBand	Up	—	1019	1	0.098%
SBand	Down	625K	3618	53	1.44%
SBand	Down	2.5M	3671	0	0%
XBand	Down	3M	4941	0	0%

To determine whether the packet losses were caused by the cryptography library, this data was compared with previous tests for raw data and did not include encryption. The packet losses for non-encrypted tests are shown in Table II.

TABLE II
PACKET LOSSES FOR NON-ENCRYPTED TRANSMISSIONS

Radio	Direction	TX Rate	Good	Skipped	% Loss
UHF	Up	—	95	27	22.13%
UHF	Down	9.6K	33	53	61.63%
SBand	Up	—	779	0	0%
SBand	Down	625K	2916	84	2.8%
SBand	Down	2.5M	2956	3	0.1%
XBand	Down	3M	4916	0	0%

Comparing Tables I and II, introducing encryption does not affect the likelihood of packet loss; the differences in percent loss are negligible.

B. Artemis CubeSat Kit and Ke Ao

This section will discuss the FSW architectures used in the Artemis CubeSat kit and Ke Ao, which also uses majority of Artemis' hardware and has an identical FSW model. Both the COSMOS and F Prime frameworks were implemented into these satellites.

1) FSW Architecture:

COSMOS (and a custom model):

The flight software for both the Teensy 4.1 and Raspberry Pi Zero, as well as the ground software for a Teensy 4.1 ground station board were developed.

Since the primary on-board computer for the CubeSat is a Teensy 4.1 microcontroller and does not run any form of operating system, the flight software is a baremetal implementation to meet the requirements of the ideal flight software architecture. The developed architecture is a custom model specifically designed for Arduino platforms.

Every major functionality of the satellite is split into "channels" which are essentially threads that are executed in parallel. This ensures that every component is isolated from the other components to enable modularity, single-component unit testing, and debugging. These "threads" are merely simulated using the TeensyThreads library as the Teensy ARM Cortex-M7 is a single-core processor. Another key feature of this custom model is that every channel contains a packet queue for processing. Any incoming packets to be handled by that specific component will be sent to the queue where the channel will handle them in first-in-first-out (FIFO) order. There is also a main channel that handles initial deployment functionalities, such as antenna deployments and initializing all of the other components. In addition, this main channel acts as a router for all the other channels: every packet transfer between channels gets sent to the main channel first and then routed to the desired channel. The reasoning behind this method is to localize how packets are routed all in one channel rather than being scattered across multiple channels. For example, if a command packet is received from the radio channel, this packet could be meant for the PDU channel, or a sensor channel, or the raspberry pi channels, etc. Rather than decoding the packet destination within the radio channel itself, it can be simply forwarded to the main channel where any packets can be decoded and forwarded to the proper destinations. A high level architecture of this model is shown in Figure 11.

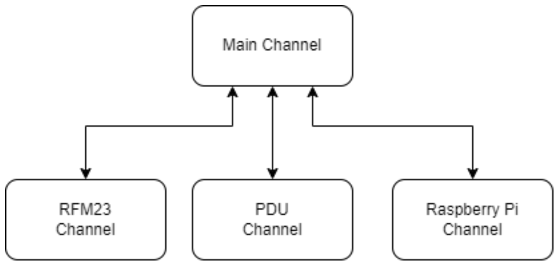


Fig. 11. FSW Model Implementing COSMOS for Artemis / Ke Ao

The structure and organization of the PlatformIO code base for the Teensy 4.1 was redesigned to be more user friendly for end users. Specifically, the libraries for Artemis specific components, such as the PDU and RFM23 radio, were generalized in a way that allowed for flexibility in device configurations. For example, the PDU serial UART bus is no longer hard-coded and can be modified to meet the design requirements of third-party OBCs using the Artemis PDU. The only constraint for this PDU library is that the OBC must still be Arduino compatible. The RFM23 library now has a struct to specify every configuration of the radio, including a custom SPI bus, transmission power, frequency, interrupt pin, and state configuration pins (TX or RX).

The code base also integrated the micro-cosmos library as a library submodule. This ensures that all PacketComm protocols used by the flight software are always up to date and supported with the protocols used by other satellites and ground stations running COSMOS.

In addition, Artemis channels were removed from the list of libraries as they are dependent on the requirements of the end user. Libraries were intended to have minimal changes over time and be shared across all users of the Artemis CubeSat kit, and the Artemis channels did not fall under that category.

A design modification to the Artemis OBC introduced a UART bus to the Raspberry Pi in addition to the I2C bus. Since the Raspberry Pi Zero only supports I2C master and not I2C slave, it was not a practical form of communication between the Teensy 4.1 and Raspberry Pi, which prompted the change to use UART. The Raspberry Pi channel in the flight software was modified to use UART instead of I2C. More specifically, the COSMOS SLIP protocol was used to satisfy this change.

The flight software of the Raspberry Pi Zero was developed to support taking a picture using the Raspberry Pi camera as well as basic PacketComm commands such as pinging the device.

The architecture of the Raspberry Pi flight software is similar to that of the Teensy flight software with the use of COSMOS channels. They are multi-threaded processes to support parallel packet handling across primary features on the Raspberry Pi, such as the Teensy serial communication and payload camera. The Teensy channel uses the COSMOS SLIP protocol to receive and process incoming packets from the Teensy 4.1. Currently the flight software handles the CommandCameraCapture and CommandObcHalt packet types to take a picture and safely shut down the Raspberry Pi, respectively. The CameraCapture command executes a python script that takes a picture using the picamera2 python package. Likewise, the ObcHalt command executes "sudo shutdown now" which shuts down the Raspberry Pi before the 5V power line is turned off.

The ground station setup included an RFM23 radio connected to a Teensy 4.1. The purpose of this setup was to provide a simple method of simulating a ground station without the use of SDRs nor GNU radio. The ground station software is capable of uplinking PacketComm packets to flight hardware. Key features include pinging the satellite, enabling and disabling switches, requesting for beacons, and taking a picture. The beacons include temperature, voltage and current, magnetometer, IMU, GPS, and switch statuses. Upon receiving these beacons on the ground, the Teensy 4.1 converts them into JSON strings and forwards them to COSMOS web using sockets.

F Prime:

In addition to the COSMOS FSW implementation into the Artemis CubeSat kit, a collaboration with NASA Jet Propulsion Laboratory was established to integrate the F Prime FSW framework into the kit as well. The goal was to replicate, or even improve, the functionalities of the COSMOS implementation, including but not limited to radio telecommunications, PDU switch controlling, periodic IMU, magnetometer, temperature, and voltage/current telemetry, and taking a picture using the Raspberry Pi camera module. The benefit of the F Prime implementation is that it is robust and can be deployed on various hardware, unlike COSMOS. Therefore, F Prime can be deployed on both the Teensy 4.1 and Raspberry Pi Zero W on the Artemis OBC.

Beginning with the Teensy deployment, several components needed to be rewritten to follow the communications flow of the F Prime architecture, which included the RFM23 radio, PDU, INA219 current sensor, LIS3MDL magnetometer, LSM6DS IMU, and the TMP36 temperature sensor. Each of these components behaves similar to how they were implemented as "channels" in the COSMOS model. Every component can be viewed as separate threads which are driven by a rate group and executed though a scheduler. Again, these threads are merely simulated and executed in a round robin fashion rather than in parallel simultaneously since the Teensy is only a single-core processor and can handle a single thread.

A separate F Prime deployment on the Raspberry Pi Zero was also developed to allow commanding of taking a picture using the libcamera package. Since the primary computer is the Teensy 4.1, direct commanding of the Raspberry Pi is not possible. Instead, the core F Prime GenericHub component was added to both the Teensy and Raspberry Pi deployments which allows ports to be connected between multiple deployments. The GenericHub is essentially a packet forwarder that simply sends data frames to the other deployment. The Framer and Deframer components allow the software to pinpoint the beginning and end of a packet being transmitted between deployments. GenericHub also incorporates an array of ports (both input and output) for components that require return values, where each port is designated to a specific use case: (1) commanding; (2) file downlink; and (3) file uplink. A CommandSplitter component is also present in the Teensy 4.1 deployment and is designed to forward a command packet either to the Teensy's CommandDispatcher component or the Raspberry Pi's CommandDispatcher component based on the command ID value. If the command is intended for the Raspberry Pi, the packet is simply sent to the GenericHub component.

Lastly, a ground deployment was also developed to simulate wireless communications between the satellite and a ground station using the on-board RFM23 transceiver module. For downlinking packets, this deployment forwards packets received from the RFM23 and sends them to a UART driver which would be connected to a host machine running the F Prime ground data system (GDS). For uplinking packets (i.e. commanding), the GDS sends packets to the UART driver of the ground deployment via USB, which forwards it to the RFM23 transceiver to be sent to the Teensy FSW deployment.

The primary components of the three deployments: Ground, Teensy, and Raspberry Pi, are depicted in Figure 12.

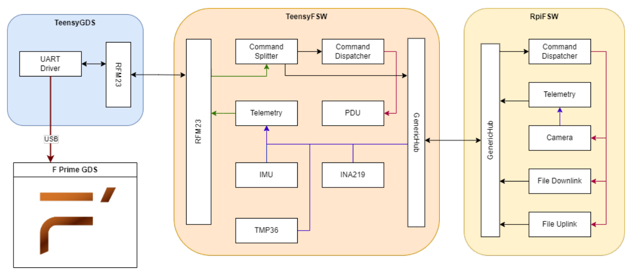


Fig. 12. High Level Architecture of the Artemis CubeSat Kit Deployments

2) Encryption: With the addition of encryption for HyTI, the possibility of implementing encryption for Artemis and other CubeSats utilizing the Artemis CubeSat kit, such as Ke Ao, was explored. The same encryption and decryption algorithms shown in Algorithm 1 and Algorithm 2 were used. The only difference is that OpenSSL was not available for Arduino microcontrollers so a separate encryption library was developed. The Arduino version of a cryptography library called "Crypto" was used.

The encryption implementation on an Arduino based microcontroller using Algorithm 1 is shown below.

```
int32_t Crypto::encrypt(vector<uint8_t> plaintext_str, vector<uint8_t> &ciphertext_str, size_t iv_size)
{
vector<uint8_t> plaintext;

 plaintext.resize(4);
uint32to(now(), plaintext.data());
 plaintext.insert(plaintext.end(), plaintext_str.begin(), plaintext_str.end());

randomizeIV(iv_size);

 ciphertext_str.resize(plaintext.size());
gcm.encrypt( ciphertext_str.data(), plaintext_str.data(), plaintext_str.size());
 ciphertext_str.insert( ciphertext_str.end(), iv.begin(), iv.end());

return 0;
}
```

Rather than using the COSMOS time library to generate a sequence number, the Arduino TimeLib was used instead. The now() function returns a 4 byte integer of the time elapsed in seconds since January 1, 1970.

The decryption implementation on an Arduino based microcontroller using Algorithm 2 is shown below.

```
int32_t Crypto::decrypt(vector<uint8_t> ciphertext_str, vector<uint8_t> &plaintext_str, size_t iv_size)
{
iv.clear();
for (size_t i = ciphertext_str.size() - iv_size; i < ciphertext_str.size(); i++)
{
iv.push_back(ciphertext_str[i]);
}

 ciphertext_str.resize( ciphertext_str.size() - iv.size());
 plaintext_str.resize( ciphertext_str.size());
gcm.decrypt( plaintext_str.data(), ciphertext_str.data(), ciphertext_str.size());

return 0;
}
```

This library has yet to be tested on the Artemis CubeSat kit as encryption is not a priority for kit distribution.

VI. Fallback

Many space missions utilize SD cards to store critical flight information, or even full operating systems that operate the primary flight software for critical mission functionalities. The hardware of SD cards are prone to radiation exposure that can cause critical damage to the internal components, rendering the media device inoperable. As a result, failsafe features need to be implemented to ensure basic communications can still be established with the satellite during an SD card failure. Without an operating system to run a full FSW deployment, a baremetal FSW deployment should be available to take over. This further emphasizes the necessity for a FSW framework to be portable, as the same deployment can be used but using a different OS abstraction layer. This also often results in smaller footprints as there is less on-board storage available due to the damaged SD card, and thus, the deployment must disable features that require significant CPU and memory usage. The baremetal deployment should have the capability to receive/handle commands and downlink telemetry from sensor readings.

A. HyTI

The HyTI 6U CubeSat utilizes the ISIS on-board computer (iOBC) running KubOS to perform most of the flight software operations. However, the mission is dependent solely on the successful boot into KubOS from the SD cards; if the SD cards were to fail after deployment, KubOS will fail to boot and the flight software will not operate, thus ending the HyTI mission. The option of running a standalone flight software program on the iOBC was explored in the case of SD card failure. The goal was to run FreeRTOS and micro-cosmos, enabling basic PacketComm and I/O functionality for the radios, much like the flight software architecture for the Artemis CubeSat kit.

The full flowchart for HyTI's flight software operations with the addition of the new standalone program is shown in Figure 13. After deployment of HyTI, the iOBC will initiate the standalone program and increment a global counter stored in local memory, such as the FRAM. According to [9], FRAM is non-volatile and can retain its information when the system is powered off. This makes it useful to store critical flight software parameters in the case of hardware or software failure, as well as after periodic power cycles. If this global counter were to exceed a value of 3, that means that the standalone program has executed three times, indicating a failure to boot into KubOS on the SD card, leading to the need to run the FreeRTOS and micro-cosmos standalone program. If this counter has not yet reached 3, then u-boot will be executed, boot into KubOS, the COSMOS flight software agents will initiate, and the counter in local memory will reset. If u-boot fails to boot into KubOS, the on-board watchdog timer would not get a signal to reset its counter, thus rebooting the iOBC, and restarting the standalone program.

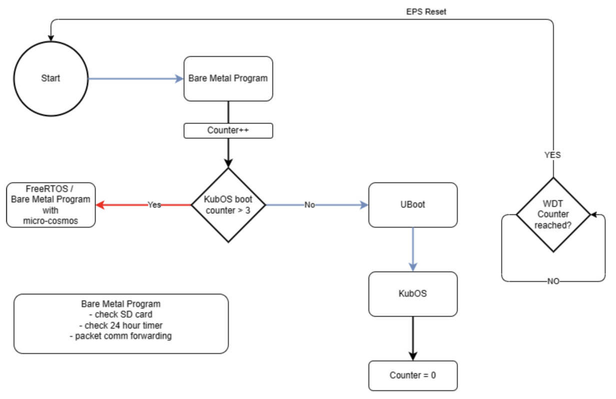


Fig. 13. iOBC Flight Software Boot Process

VII. Summary of FSW Architectures

Table III shows a summary of the mentioned flight software frameworks from Section II. The general trend of the popular FSW architectures (GERICOS, cFS, F Prime) heavily emphasize a layered architecture as that is the best design to enable modularity, portability, and having a small footprint. In other words, having a layered architecture allows end users to replace these layers with custom implementations to meet the requirements of the mission's hardware and/or operating system. While the COSMOS framework and baremetal model implementation for the Artemis CubeSat kit partially meets the modularity requirement because of their use of multi-threaded channels, each channel does not have a well-defined structure; they are simply threads that handle tasks in parallel. These channels can often get substantially large, increasing the difficulty of debugging, while also decreasing the capability of unit testing since independent and isolated features are less defined. Likewise, COSMOS does not incorporate a layered architecture, preventing the deployment into other architectures and operating systems. Linux system calls are directly integrated into the core libraries and channels of COSMOS, and the Arduino libraries are directly integrated into the channels of the Artemis / Ke Ao model, forcing integration into a single platform that supports those system calls.

A. A Deeper Look into COSMOS vs. F Prime

As the COSMOS and F Prime flight software frameworks were heavily emphasized and implemented into missions such as NASA's HyTI mission and the Artemis CubeSat kit, respectively, these two frameworks will be analyzed more in depth.

Both flight software frameworks have unique implementations with strengths and weaknesses. Unlike COSMOS, F Prime has more modularity and portability features because components of the flight software are broken down into smaller parts. Modularity should allow for implementations of new components without the need to modify other components. For example, let's have a telemetry and radio component. The output of the telemetry component would be connected to the input of the radio component, allowing for all telemtry bytes to be transmitted out the radio. If an encryption component were to be implemented, we can simply reconnect the telemetry output to the input of the encryption component, and the encryption output into the input of the radio component without needing to modify the source code of any of those three components. However, during the implementation of

TABLE III
FSW Requirements Met By Various Frameworks

Requirements	**GERICOS**	**NASA cFS**	**F Prime**	**COSMOS**	**Artemis / Ke Ao "Model"**
Modularity (component-based design)	Yes	Yes	Yes	Partially	Partially
Portability (Support for other hardware/OS)	Yes	Yes	Yes	No (Limited to Linux)	No (Limited to Arduino)
Small Footprint	Yes	Yes	Yes	No	Yes
Unit Testing with Significant Code Coverage	Unknown	Yes	Yes	No	Partially
Documentation	Unknown	Yes	Yes	Partially	Partially
Standardized Packet Protocol	Yes	Yes	Yes	Yes	Yes

encryption for the HyTI mission using COSMOS, this was not possible. Multiple files across numerous channels and modules needed to be modified to incorporate encryption. Additionally, some of the core framework needed to be modified to support encryption, which was not ideal.

The F Prime framework also incorporates automated code generation, making it more user friendly for development. The back-end features of F Prime, such as command and telemetry registrations, port connections, fault handling, and many more do not need to be implemented by end users. Instead, the end users are responsible for the design of misson-specific capabilities, which facilitates the modularity of F Prime. One of the major advantages of F Prime is that it has a smaller footprint than COSMOS since it is capable of running on low-powered embedded systems, such as Arduino boards, whereas COSMOS requires significant CPU and memory usage, typically one that supports a Linux distribution. F Prime was tested to work with Arduino based micro-controllers and utilize the libraries available to the Arduino development environment. Additionally, F Prime has been successfully deployed using the Zephyr Real-time Operating System (ZephyrRTOS).

The powerful feature of COSMOS includes the concept of COSMOS aware programs, which are multi-threaded agents connected in a network. Multiple agents are capable of being executed on a single machine, or on multiple machines, while still being able to be linked together in this network. Any agent within that network can seamlessly send requests to other agents within the same network. This enables capabilities for organizing and coordinating a swarm of satellites. Currently, F Prime deployments have limited capabilities of connecting to other deployments, although the GenericHub component implemented in the Artemis CubeSat kit has been progressing to fulfill this capability.

VIII. Future Work

Based on Table III, the COSMOS framework needs improvement to meet the requirements of modularity, portability, unit testing capabilities, and decreasing the footprint. The current core framework of COSMOS had significant changes to create functions specific to the HyTI mission. After the delivery of HyTI, COSMOS should be redesigned such that it incorporates a layered architecture, similar to the GERICOS, cFS, and F Prime architectures. Not only would this encourage modularity, but it will enable portability as a layered architecture will allow for interchangeable OS abstraction layers for various hardware and operating systems. The ability to deploy COSMOS on other embedded systems, especially those of lower resources, will encourage a small footprint design which can be achieved by adapting the COSMOS channels into a component-based design.

IX. Conclusion

A robust, safe, and sustainable flight software framework is essential for a successful mission. Many of the existing frameworks that has been proven to succeed in numerous large scale missions including GERICOS, NASA's cFS, and F Prime, follow a strict architecture. These frameworks use a component-based design and layered architecture to enable modularity and portability. Adapting from these existing frameworks, a standardized architecture for all flight software must implement an OS abstraction layer that handles essential OS functionalities, such as a file system, task scheduler, network manager, memory manager, I/O manager, clocks and timers, synchronization, messager interpreter, and system health. These can be interchanged with other OS abstractions (i.e. Posix, RTOS, RISC-V, Arduino, etc.) to meet the hardware requirements without interfering with other framework components. Above this OS abstraction layer is an application layer consisting of essential FSW core components that all spacecraft adapts, such as commanding, telemetry, file uplink and downlink, and a packet protocol validation. Any mission-dependent components, such as radios, sensors, or actuators are implemented outside the scope of the core framework but still within the application layer of the architecture.

Another framework design consideration involves implementing cryptography. Encryption prevents unwanted commanding of the spacecraft from malicious attackers; only those with access to the same private key as the spacecraft may command it. Likewise, telemetry may include sensitive sensor data which could be decoded if unencrypted. The ideal encryption protocol that was tested was the AES-GCM algorithm which implements AES-256 encryption.

References

[1] D. J. F. Miranda, M. Ferreira, F. Kucinskis, and D. McComas, "Comparative survey on flight software frameworks for 'new space' nanosatellite missions," *Journal of Aerospace Technology*

and Management, vol. 11, p. e4619, 2019. [Online]. Available: https://doi.org/10.5028/jatm.v11.1081

[2] "Space packet protocol," Consultative Committee for Space Data Systems, 2020. [Online]. Available: https://public.ccsds.org/Pubs/133x0b2e1.pdf

[3] P. Plasson, C. Cuomo, G. Gabriel, N. Gauthier, L. Gueguen, and L. Malac-Allain, "GERICOS: A Generic Framework for the Development of On-Board Software," in *DASIA 2016 - Data Systems In Aerospace*, ser. ESA Special Publication, L. Ouwehand, Ed., vol. 736, Aug. 2016, p. 39.

[4] "core flight system (cfs) background and overview," NASA, 2014. [Online]. Available: https://cfs.gsfc.nasa.gov/cFS-OviewBGSlideDeck-ExportControl-Final.pdf

[5] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, "F prime: An open-source framework for small-scale flight software systems." AIAA/USU Conference on Small Satellites, July 2018.

[6] M. Nunes, T. Sorensen, E. Pilger, M. Wood, H. Garbeil, J. Lewis, and D. Azimov, "Expanding the comprehensive open-architecture space mission operations system (cosmos) for integrated guidance, navigation and control of multiple small satellites." American Institute of Aeronautics and Astronautics, May 2014.

[7] M. Dworkin, "Recommendation for block cipher modes of operation," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. 800-38A, 2001.

[8] ——, "Recommendation for block cipher modes of operation: Galois/-counter mode (gcm) and gmac," National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. 800-38D, 2007.

[9] H. Péter-Contesse, A. Piplani, and E. Timmer, "Isis-obc datasheet," Innovative Solutions in Space (ISIS), 2018.