2025 MITRE eCTF
Final Paper for GT1 (Huzz)

Nicolas Amato
Georgia Institute of Technology
Atlanta, Georgia
namato6@gatech.edu

Daniel Winsness
University of North Georgia
Dahlonega, Georgia
dswins3550@ung.edu

Tony Tanory
Georgia Institute of Technology
Atlanta, Georgia
ttanory3@gatech.edu

Tracy Guo
Georgia Institute of Technology
Atlanta, Georgia
tguo72@gatech.edu

Mahir Riki
Georgia Institute of Technology
Atlanta, Georgia
mriki@gatech.edu

Denny Hall
Georgia Institute of Technology
Atlanta, Georgia
dhall317@gatech.edu

Shayan Aqeel
Georgia Institute of Technology
Atlanta, Georgia
saqeel3@gatech.edu

***Abstract*—The 2025 MITRE Embedded Capture the Flag (eCTF) competition challenges teams to design and implement a secure Satellite TV System, ensuring the confidentiality, integrity, and authenticity of TV data streams. This document outlines the functionality and security requirements of our system, detailing our approach to implementing robust security mechanisms within the Decoder and Encoder firmware.**

1. Introduction

We are the first team (GT1) of Georgia Tech's Embedded Systems and Cyber Security VIP program. Our team consists of 7 undergraduates, most of whom have never participated in the competition before. The team is advised by GTRI researcher Kevin Thompson.

1.1. Initial Considerations

Our design is built using C. We decided that switching to another language would pose the issue of having to maintain a new build toolchain and having to rely on third-party ports of the MSDK HAL, if there existed any. In addition, the C language is simple to understand, allowing all team members regardless of experience to contribute to the design. Languages such as Rust offer a robust type system and enhanced security features, including protection against buffer overflows. However, this would be an unnecessary risk given the steep learning curve of the language and additional infrastructure needed.

2. Build Environment and Setup

The build environment will host all of the required tools, packages, dependencies, and other developer tools necessary for the device to run and operate. Docker will be the main way these dependencies will be added and used in the design. Docker is a containment management system that will create a reproducible build environment (container) that compiles code exactly the same way, regardless of the host platform. The design environment is created using a "*docker build*" based on the requisite packages provided in the "*DockerFile*" by the competition organizers.

The dependencies included in the "*DockerFile*" are *gdb, gdb-multiarch, gcc-arm-none-eabi, binutils-arm-none-eabi, make, python3.10, git, wget, libtool, and texinfo.* The Docker deployment will need to be created after the secrets folder is generated. Once the Docker container has been successfully created, a subscription update will need to be generated. The Decoder firmware will have to be flashed onto the MAX78000FTHR boards, which is needed to build the design.

3. Functional Requirements

This section outlines the functional requirements of the design. The system consists of an Encoder that is responsible for encoding and broadcasting TV frames via Uplink, and a Decoder responsible for decoding and displaying TV frames for authorized users. The final

design must adhere to all functional requirements before progressing to the attack phase.

3.1. Architecture

The architecture of the system is built around the Analog Devices MAX78000FTHR development platform, which serves as the hardware foundation for the Decoder. The MAX78000FTHR provides the necessary processing power and connectivity to support the system requirements.

3.2. Building Process

Global Secrets are generated by the host tool. There is a Docker based environment to install dependencies, compilers, packages, and build tools. The Decoder firmware is built from the design and flashed onto the MAX78000FTHR. Subscription updates are generated including channel numbers, Decoder IDs, timestamps, and information from Global Secrets.

3.3. Encoder Requirements

Once the system is built, the Encoder can be started. The Encoder is implemented as a function with raw frames, channel number, and secrets as inputs. The Encoder prepares the frame for Uplink by encoding it with the necessary metadata.

3.4. Decoder Requirements

With a functioning Decoder, the system is ready to accept signals from the satellite TV system. The Decoder must be able to respond to the list channels, update subscriptions, and decode frame commands. Using the Decoder from the reference design, and adding security measures on top of that would satisfy the functional requirements.

4. Cryptographic Primitives

In this section, we outline the cryptographic primitives used to fulfill the security requirements of our design. We selected well-established, standardized algorithms that have undergone rigorous research and cryptanalysis. This ensures our strong confidence in their algorithmic security.

4.1. HMAC

Hash-based message authentication codes [1] are used to verify message integrity. It achieves this by using a secret key and a hash function to generate a tag unique to the message. The HMAC is included with the outgoing message to allow the receiver to verify message integrity. The receiver does this by recomputing the HMAC and comparing tags.

Algorithm 1 HMAC Verification

Require: Message M, Key K, Received MAC T_{received}
Ensure: Valid or Invalid
1: computed_mac $\leftarrow$ **HMAC**(M, K)
2: **if** computed_mac $= T_{\text{received}}$ **then**
3: **return** Valid
4: **else**
5: **return** Invalid
6: **end if**

4.2. AES-CBC

The Advanced Encryption Standard is a block cipher utilizing a symmetric key to both encrypt and decrypt data. Since the cipher purely operates on a block of 16 bytes, it requires a mode of operation in order to operate on larger data. AES-ECB is the simplest yet most insecure mode of operation, which uses the same key to operate on all blocks leading to cases such as patterns in the plaintext mapping to patterns in the ciphertext, revealing information that should otherwise be hidden. A famous example is the "ECB Penguin [2]" illustrating this effect.

We chose to use AES-CBC [3] (Cipher Block Chaining) using an IV (Initialization Vector) to introduce randomness and prevent the aforementioned ECB problem. We also use a PKCS#7 [4] padding scheme to ensure that any sized data can be decrypted. Upon decryption failure, a generic error message is returned to reduce the risk of padding oracle attacks. The CBC mode operation for encryption and decryption is defined as follows:

Encryption with PKCS#7 Padding

Given plaintext blocks $P_1, P_2, \ldots, P_n$, initialization vector IV, and block cipher E_k, the plaintext is first padded using PKCS#7 to ensure its length is a multiple of the block size. Let $P = P_1 \| P_2 \| \cdots \| P_n$ be the original message, and $\widetilde{P} = \textbf{PKCS7}(P)$ the padded plaintext, split into blocks $\widetilde{P}_1, \ldots, \widetilde{P}_m$.

$$C_0 = IV$$
$$C_i = E_k(\widetilde{P}_i \oplus C_{i-1}) \quad \text{for } i = 1, 2, \ldots, m$$

Decryption with PKCS#7 Padding

Given ciphertext blocks $C_0, C_1, \ldots, C_m$ and block cipher D_k:

$$\widetilde{P}_i = D_k(C_i) \oplus C_{i-1} \quad \text{for } i = 1, 2, \ldots, m$$

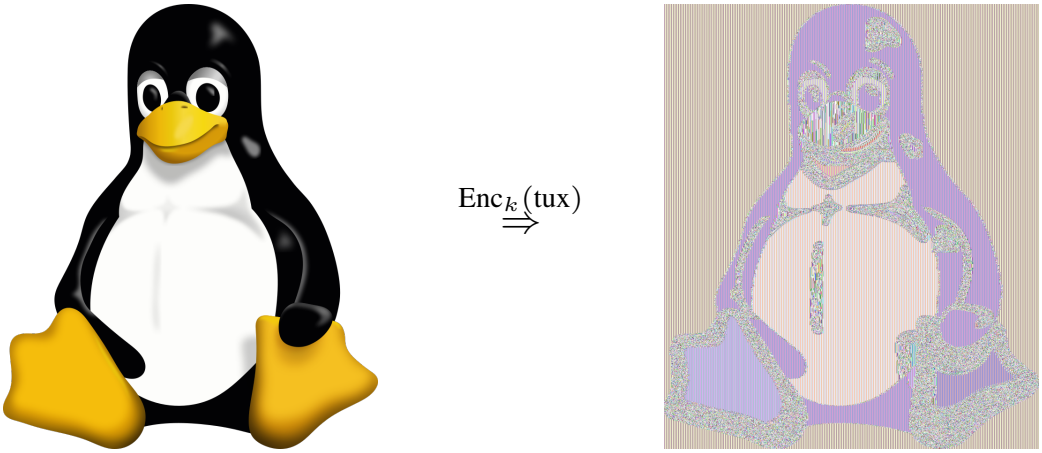

Figure 1: Image encryption using AES-ECB resulting in visually similar images [2].

After decryption, the original plaintext P is recovered by removing the PKCS#7 padding:

$$P = \textbf{RemovePKCS7}(\widetilde{P}_1 \| \cdots \| \widetilde{P}_m)$$

In addition, CBC by itself does not have an authentication mechanism like GCM, so we employ HMAC in an encrypt-then-HMAC fashion. This approach requires separate keys for encryption and authentication unlike GCM which uses a single key for both. Even if the HMAC key is compromised attackers cannot decrypt messages due to this key separation.

4.3. PBKDF2

Password-based key derivation function [5] generates a key from a password and a salt by appending the salt to the password and repeatedly hashing the value. The number of hashes performed is called the iteration count and derives security from its key stretching process making brute-force attacks computationally expensive.

5. Security Requirements

This section details the necessary security requirements the design must follow and how we implement each of them. The following requirements are put in place to ensure the protection of data and functionality critical to the secure system operating as intended. To prevent unauthorized access an attacker should not be able to decode TV frames without a Decoder that has a valid, active subscription to that channel. Additionally, to maintain system integrity the Decoder should only decode valid TV frames generated by the Satellite System the Decoder was provisioned for. This requirement ensures that our system cannot be used to decode frames in any context other than what our team has designed it for. Lastly, the Decoder should only decode frames with strictly monotonically increasing timestamps. This will ensure that frames are only processed and displayed in the order intended by the TV station.

5.1. Security Requirement 1: Channel Subscription Verification

For this requirement, the Decoder should only be able to decode TV frames from a channel it has an active, valid subscription for. This requirement is crucial for protecting our system from attackers using pirated and out of date subscriptions to access frames from channels they are not permitted to use.

The question arises of how to ensure that only the intended decoder receives the subscription data. To begin, outgoing subscription update packets need to be encrypted with a symmetric key only the Decoder and Encoder know. This key is called the subscription update key K_s and is derived from a salt S_s from Global Secrets and the Decoder ID. The salt S_s is generated using the Host's CSPRNG (**urandom**).

$$K_s := \textbf{Hash}(S_s \parallel \text{DecoderID})$$
$$P_s := \textbf{Dec}_{K_s}(C_s)$$

This guarantees that only valid and intended Decoders can decrypt subscription update messages. The subscription update payload can now be decrypted and written to the Decoder's flash. Upon every frame decode, the provisioned Decoder checks to see if the current frame is within the valid time window specified by the subscription, if not it rejects the frame.

Subscription Update Decrypted Payload

Name	Offset	Size (bytes)
Device ID	0x00	4
Start Timestamp	0x04	8
End Timestamp	0x0C	8
Channel ID	0x14	4
Channel Key	0x18	16

5.2. Security Requirement 2: TV Frame Validation

For this requirement, the Decoder should validate TV frames before decoding them, making sure that the TV frames were generated by the satellite system for which the Decoder was provisioned for. With this, TV frames injected by a malicious party who do not have control of the satellite system will not be decoded, preventing malicious operations from being run on the Decoder.

We utilize HMAC tags embedded in every message to verify integrity. The Global Secrets contain an HMAC key generated using **urandom**. This guarantees that only valid decoders can verify message authenticity using the shared key. We utilize HMAC with the SHA256 hash algorithm. HMAC computation is as follows:

Let M be the message payload, τ the HMAC tag, and K_h the HMAC key.

$$\tau = \textbf{HMAC}(M, K_h)$$

Generic Message Structure

Name	Offset	Size (bytes)	
Channel ID (?)	0x00	4	(Frames only)
HMAC Tag	0x00	32	
IV	0x20	16	
Payload	0x30	??	

Incoming messages to the Decoder are then checked for authenticity with **HMACVerify**(M_p, τ, K_h) with M_p being the message payload.

5.3. Security Requirement 3: Frame Order Integrity

This requirement ensures that the Decoder will only decode frames with strictly monotonically increasing timestamps, preserving the correct order of frames as intended by the TV station.

To ensure this, the Decoder validates that each incoming frame's timestamp is strictly greater than the timestamp of the last successfully processed frame. Frames with older or equal timestamps are discarded. Additionally, to prevent replay attacks, the integrity of the timestamp is secured by HMACs, ensuring that the timestamp cannot be altered or replayed. As a result, frames with invalid timestamps are automatically discarded, ensuring that the frames are decoded and displayed in the order intended.

However, there is still the problem of the "first timestamp". Since we maintain a previous timestamp value, if we assume the value is 0 upon initialization, then an incoming timestamp of 0 will be rejected. We solve this by saving whether we have received a frame for a specific channel, if not we accept any incoming frame. This sets the reference timestamp for future frames.

Let $B \in \{0,1\}^8$ denote a byte representing the frame reception status for 8 channels. Each bit B_i (with $i = 0$ being the least significant bit) corresponds to channel i:

$$B_i = \begin{cases} 1 & \text{if a frame has been received for channel } i \\ 0 & \text{otherwise} \end{cases}$$

$$\text{for i = 0, 1, \ldots, 7}$$

Thus, a bitwise representation like $B = \texttt{0b01001001}$ indicates that frames have been received for channels 0, 3, and 6.

Let $E \in \{0,1\}^8$ be a second byte representing the emergency frame reception status. We define:

$$E = 0xFF \iff \text{An emergency frame has been received}$$

Otherwise, if $E \neq 0xFF$, no valid emergency frame has been received.

6. Additional Mitigations

Since sensitive data is stored on the stack when processing messages, we take the precaution and zero out those values to ensure that in the case of a stack-based or RCE attack, attackers will not be able to extract leftover keys.

In addition, we reject all messages longer than 256 bytes to prevent attackers from overflowing the UART buffer. Since the maximum longest message for our protocol is less than 256 bytes, we can afford this.

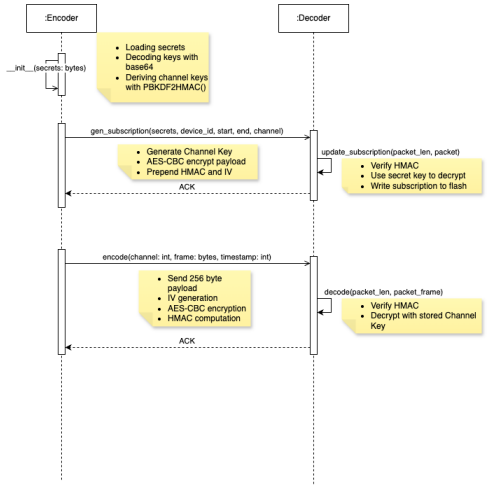

Figure 2: System Design diagram

7. Attack Phase

This section outlines the steps we used to attack other teams.

7.1. Oklahoma Christian University (OC)

OC uses AES-GCM which leverages AES-CTR for encryption while using the Galois Mode for authentication. The encryption can be broken through OC's design as they made the fatal mistake of reusing the same AES-GCM key and IV for every channel and every message. This means the AES-CTR keystream generated for every encryption is the same. AES-CTR encrypts plaintext in the following manner:

$$\text{keystream} \oplus \text{plaintext} = \text{ciphertext}$$

Since xoring is reversible and the keystream remains the same for every encryption in OC's design, the keystream can be obtained by xoring a known plaintext and ciphertext pair like so:

$$\text{ciphertext} \oplus \text{plaintext} = \text{keystream}$$

A known plaintext and ciphertext pair is known for channel 1 as the attacker decoder has a valid subscription for this channel. Therefore, this pair can be used to get the reused keystream from which every channel's encrypted content can be decrypted as such:

$$\text{keystream} \oplus \text{ciphertext} = \text{plaintext}$$

This can be done for the encrypted content on channel 2, 3 and 4 as well as the recording playback to get the expired subscription, pirated subscription, no subscription and recording playback flags.

7.2. SEMO

SEMO uses AES-CBC to encode their messages and use the same key and IV for every channel's encryption. They have their AES-CBC key and IV in their gen_secrets.py file in plaintext. With this, a python script can be written utilizing this key and IV to decrypt the encoded messages sent through channels 2, 3 and 4 as well as the recorded playback in order to receive the expired subscription, pirated subscription, no subscription and recording playback flags. In reverse, a python script can also be written to use their keys to forge a frame to obtain the pesky neighbor flag.

7.3. Binghamton

Binghamton uses a simple double-layer AES-ECB scheme with a channel key for encrypting and decrypting frame payload data, while a master key encrypts and decrypts the entire frame. The subscription generation process is just **Enc**(**DeviceID** $\|$ **Start** $\|$ **End** $\|$ **Channel**), so this allows us to easily modify the subscriptions on a per block basis. For the pirated flag, we replaced the first 16 bytes of our pirated subscription with our valid subscription to get a valid start timestamp, although the last 4 bytes of the end timestamp don't get changed, which was not a problem. We then used this modified subscription to get the pirated flag. For the playback flag, we noticed that the frame header is exactly 16 bytes, so we could just replace it with a valid header from channel 1 which we have a valid subscription for. We send the modified frame to the decoder to get the playback flag.

7.4. CA

CA only have an HMAC key to verify the integrity of their message but do not achieve confidentiality in their design as they do not encrypt their frames sent through the channels in any way. On top of this, the decoder.c file logic does not check whether the decoder is subscribed to the channel it is reading from, meaning all channels print their decoded contents to every decoder anyways. Due to this, the expired subscription, pirated subscription and no subscription flags can be read in plaintext by running the TV on channels 2, 3 and 4 and the recording playback flag can be obtained by converting the hex for any encoded message to plaintext as there is no encryption. Similarly, the pesky neighbor flag was extremely simple to obtain since no encryption was present. We could simply send a packet of all 0's and get the pesky neighbor flag back.

7.5. ETSU

ETSU have no checks for their generate subscription tool which is typically enforced in

the gen_subscription.py file. This means that the generate subscription tool can be used to generate a subscription to every channel for our attacker decoder by using a fake secrets.json file for which the contents do not matter, 0 for the lowest time stamp accepted and a very large value as the highest time stamp accepted as arguments to the tool. The final tool usage will look like this: `python -m ectf25_design.gen_subscription .\fake_secrets.json chan_4.sub 0x4d5d363c 0 10000000000000000 2`. This can be done for every channel so that the decoder accepts the subscription and properly decodes the encoded content to print the flags for the expired subscription, pirated subscription and no subscription channels of 2, 3 and 4.

7.6. USCGA

USCGA validates a device ID when you subscribe to a channel. Channel 0, however, is always subscribed by default, so sending a frame on channel 0 bypasses both the subscription and device-ID checks. Combined with USCGA's hard-coded XOR key, we can forge a valid channel 0 packet. That packet will be decoded successfully, revealing the Pesky Neighbor flag.

8. Individual Contributions

8.1. Tracy Guo

At the beginning of the project, Tracy struggled to fully understand the security requirements and how different components of the system interacted. To address this, she revisited the official website, researched the system components, and discussed implementation details with the team to develop a clearer understanding. She contributed to Security Requirement 1 by researching encryption methods, including nonce-based encryption and HMAC authentication. She reviewed Nicolas' proof of concept and Kevin's design to understand how hashing and HMAC authentication were being used, which reinforced her understanding of the team's plan. Tracy attended MITRE's workshops on rules, reference design, boot reference, Docker, OpenOCD debugging, and Attack the Reference, taking notes to help with steps to obtain the flags and navigate throughout the competition. She also contributed to the initial design document and kept up with team meetings.

To help with the final secure design, Tracy attended team meetings for implementing Security Requirement 1 with Nicolas, Mahir, and Denny, where she was able to get her environment completely set up and waited to test the encoder. She also began the first draft of the system design diagram to finalize the design document before submitting to handoff. As she was waiting for the handoff, she watched all the videos and read all the articles that Kevin linked in the ectf readme, including AES encryption, session vs token authentication, and others. Once the team entered attack phase, Tracy attended the Pesky Neighbor workshop to learn how to obtain the pesky neighbor flags and reviewed her teams' attack writeups to see where to get started.

8.2. Tony Tanory

At the beginning of the project, Tony focused on understanding the functional requirements of the design by reviewing the MITRE documentation and exploring the components of the system. Tony used the design board to boot the reference design on the hardware, testing its functionality and how the provided functions work. After successfully booting the reference design, Tony proceeded to test the simple_crypto algorithm to evaluate its implementation and security properties. Later, Tony used the attack package to analyze simple vulnerabilities in the system, walking through the process of capturing the Attack the Reference Design flags across channels 0-4 with other members of the team. For the design, the team decided the best way to implement the security requirements for the design was to split up into sub-teams where each team would work on one of the three different security requirements. Tony worked alongside Shayan and Daniel to implement security requirement 2 which states: "The decoder should only decode valid TV frames generated by the Satellite System the decoder was provisioned for". The team did this by implementing an HMAC verification system in which the encoder encrypts the packet to be sent, and creating a function that computes an HMAC with SHA-256 and prepends it to the encrypted message. Upon completing the design, the team needed to update the design document before submitting the final design. For the design document, Tony and Tracy worked together to create Sequence Diagrams that embody our design. After submitting the design, the team entered the attack phase. Tony faced a multitude of challenges with how to approach the attack phase with the team only having 3 attack boards, and 3 days to collect flags from the attack phase. Tony tried his best to contribute to the attack phase without hardware by analyzing code from other designs and learning about how other teams approached this challenge. Overall, Tony learned a lot from the semester working on this eCTF challenge, from working with a 7 person team to designing a system requiring knowledge on unfamiliar topics. Tony is looking forward to the potential opportunity to compete in a similar eCTF challenge with the same VIP team in the coming Fall semester.

8.3. Daniel Winsness

Being that it was Daniel's first time doing a CTF challenge, there was a significant learning curve he had

to overcome. Additionally, being a student at a different university from the rest of the team meant that he would have limited access to the labs and equipment. To address these issues, Daniel planned a lab meeting early into the project where he and a few other members were able to meet and discuss the competition rules, design requirements, security requirements, possible design concepts, and successfully flash design boards to use during development. After this Daniel took one design board back with him to allow him to work on the teams design at a distance. From this point on Daniel helped the team draft the design design documentation and conceptualize the design and how it would work at a high level. Though he did personal research on all three of the security requirements, his main contributions in documentation were in security requirements one and two. After the documentation was completed, he, Tony, and Shayan began working on an implementation for security requirement number two. Once attack phase began Daniel worked primarily on writing scripts to get pesky neighbor flags. He was successfully able to obtain pesky neighbor flags from SEMO, CA, and USCGA. Common vulnerabilites Daniel looked to exploit were hard-coded secrets and in proper security on channel 0. Overall, Daniel was able to get unique hands on experience with embedded systems, cryptography, and reverse engineering.

8.4. Shayan Aqeel

Shayan has helped guide the team as team leader through the different phases of the MITRE eCTF competition through his experience from last year's eCTF and outside the VIP program. In the beginning stages of the competition, he was able to eventually fully understand the functionality and security requirements of this year's design while also working with the hardware to boot, debug, test and attack the reference design provided through the development and attack MAX78000FTHR boards. Building off of this, he was able to propose a solution to security requirement 2 which guarantees TV frame and Satellite System integrity through the use of HMACs. From here, Shayan, Tony and Daniel began implementing this solution within the reference design and helped to fully secure the vulnerability. With all of this, he helped submit many flags such as the boot reference design, debugger, testing service and four attack flags in the defense portion of the competition. Moving into the handoff and attack phase of the competition, he helped complete the design document outlining specifics of the secure design submitted for the handoff. In the attack phase, he was able to target weaker teams and in analyzing their designs was able to obtain their expired subscription, pirated subscription, no subscription and recording playback flags. With this, he was able to successfully attack OC, SEMO, CA and ETSU, earning attack phase points for the team. This concluded this semester's competition from which Shayan was able to take away valuable theoretical and practical experience.

8.5. Nicolas Amato

Given this was Nicolas' first time participating in a team-based CTF as well as the VIP program, he had some trouble understanding the general requirements and workflow of the CTF challenge in addition to the VIP. He started off by creating and managing the teamwide Github repository, forked from the eCTF insecure design, where the team's main design would be worked on. Nicolas then found and submitted the boot flag. Nicolas and a few other members planned a lab meeting early on to gain hands-on experience with the insecure reference design and flashing the MAX78000FTHR boards with the insecure design, as well as discussing the challenge to further understand the requirements and propose possible solutions. Nicolas took a design board to finish flashing the reference design and to start developing the software for the design. He had discussions with Shayan on how to implement Security Requirements 1, in which we finally settled on a design using shared subscription update keys derived from the hash of the Decoder ID and a salt value for symmetric encryption to securely share channel keys. Using this design, he created a proof-of-concept illustrating the implementation of Security Requirements 1 and 2 to allow other team members to easily comprehend the design and to serve as a guide for implementing our design in software. He then took the responsibility of implementing the design for the decoder and encoder. Throughout this process, he realized a multitude of changes needed to be made such that the design could properly function. This included utilizing new primitives such as PBKDF2 to address the issue of *gen_subscription* and encoder processes being stateless, in addition to other various tweaks and changes. After this, he underwent a difficult debugging phase to fix major bugs, tune the design, implement last minute security mitigations, and polish and document the code. During this process, he attempted to use the board's AES module for speed and further security against side-channel attacks. However, he failed at this endeavor and the design had to use the WolfSSL AES implementation. At this point the design was complete and all that had to be done was updating the design document. He also took the responsibility of overhauling the design document with the new formalized design. During the limited attack phase, he was able to obtain the Binghamton pirated and playback flags. Overall, Nicolas gained invaluable experience in cryptography, CTFs, and embedded systems.

8.6. Mahir Riki

At the beginning of the project, Mahir faced significant challenges, largely due to his inexperience with

CTFs. To overcome this gap, he spent a lot of time studying the official site's documentation and watching workshop videos, which provided him with a solid foundation in understanding the functional requirements of the design. During the design phase, Mahir took a design board home to test whether he had flashed it correctly, as well as learning from the mistakes he made while flashing the hardware. He also contributed to the design document by working on the build environment and attending group meetings. In addition, he contributed to Security Requirements 1 and 3. During a sub-meeting with Tracy, Nicolas, and Denny, he provided some ideas from his research from reviewing Nicolas's proof of concept and Kevin's design for Security Requirement 1. For Security Requirement 3, he researched on how a timestamp should be calculated using the MITRE rules website and Slack. He was able to implement a version of Security Requirement 3, which was later modified by the other team members to improve the implementation. After entering the attack phase, Mahir joined a sub-meeting with Denny, Shayan, and Daniel, where everyone attempted to attack SEMO and OC but struggled due to a lack of experience with the attack board and how an attack should be initiated. Additionally, Mahir attempted to capture the Pesky Neighbor flags of several teams. He watched the Pesky Neighbor MITRE workshop and learned how to obtain the flags as well as begin writing a script to help with the flags. However, he struggled with this and was unable to capture the Pesky Neighbor flags alone. This concluded the competition for this semester, and Mahir learned a lot about how eCTFs operated and gained valuable experience in participating in MITRE's eCTF competition.

8.7. Denny Hall

As this was Denny's first Capture The Flag (CTF) competition and VIP experience, he encountered several challenges, particularly due to the steep learning curve. To build the necessary foundational knowledge, he studied MITRE documentation, researched system components, and reviewed educational videos. This preparation enabled him to understand system interactions and component relationships more effectively. Denny also contributed to the design document by developing Security Requirement 3 and actively participating in all group meetings. For security requirement 1, he worked closely with Nicholas, Tracy, and Mahir to discuss solutions for Security Requirement 1 using Nicholas's proof of concept and Kevin's design. During the short attack phase, he worked together with Daniel, Shayan, and Mahir to obtain expired subscription, no subscription, recording playback, and pirated subscription flags for SEMO. However without an attack board, Denny watched the Pesky Neighbor Workshop video, attempted to write a script to try and obtain the flag but failed. This resulted in him not getting any pesky neighbor flags.

Appendix A — Definition of Terms

Term	Definition
Frame	Actual TV image data. Guaranteed to be a maximum of 64 bytes.
Encoder	Device that generates and broadcasts encoded frames to be received by Decoders.
Decoder	Device that is responsible for receiving and decoding the frames accordingly.
Host	Trusted system designed to contain Encoder infrastructure and broadcast messages to the Decoder via UART.
Global Secrets	Shared information between the Encoder and Decoder. This is meant to contain cryptographic secrets and other values to facilitate secure communication. Valid decoders maintain Global Secrets upon provisioning.
AES-CBC	AES (Advanced Encryption Standard) using the CBC mode of operation
SHA	Secure Hashing Algorithm
HMAC-SHA256	HMAC (Hash-based Message Authentication Code) using SHA256 as the hash function
K_c	Channel specific symmetric key
K_s	Subscription update symmetric key
K_h	HMAC key
S_s	Subscription update salt
S_c	Channel key PBKDF2 salt

References

[1] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997. [Online]. Available: https://www.rfc-editor.org/info/rfc2104

[2] Wikipedia contributors, "Block cipher mode of operation — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=1279034580, 2025, [Online; accessed 7-April-2025].

[3] S. Frankel, K. R. Glenn, and S. G. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec," RFC 3602, Sep. 2003. [Online]. Available: https://www.rfc-editor.org/info/rfc3602

[4] B. Kaliski, "PKCS #7: Cryptographic Message Syntax Version 1.5," RFC 2315, Mar. 1998. [Online]. Available: https://www.rfc-editor.org/info/rfc2315

[5] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/info/rfc8018