Design Document

This design document is a summary of our design implementation for the MITRE eCTF 2024 Competition to build and secure a medical device.

Tufts University
SPRING 2024

Kenneth Chua: Captain
Charlotte Lokere

Nicole Alexandru
Christopher Bann
Sam Chung
Jake Freedman
Douglas Lilly
Kamran Mirza
Hunter Purevbayar
Remy Ren
Aliyah Weiss

1. Overview

Our design includes the combination of hardware and software security by utilizing features of the MAX78000 boards. This includes encryption, use of True Random Number Generator (TRNG), and safe use of vulnerable functions within our software implementation. Our security principles include:

1. AES Encryption: To encrypt the attestation message containing the component's provisioned location, date, and customer information. AES is a symmetric encryption algorithm.
2. Bcrypt Hash: Implements a Bcrypt hash scheme to secure the attestation pin and replacement token at compile time so that these are not stored in plain text, and can then be verified during runtime, preventing brute force attacks, and increases the difficulty of side-channel attacks.
3. Public-Key Encryption: Used to share a one time use AES key for attestation data, and to share a randomly generated secret hash key for pre-boot functionality.
4. Secure AES Keys using TRNG: At each function call, a new random number is generated and used as a key for AES encryption.
5. Secure Send and Receive: Post-boot, the AP and components use a combination of hash and random number tracking to secure messages being sent.
6. End-to-End Encryption: The return response message to the attestation command is protected using AES and end-to-end encrypted.
7. Data Protection: Data is kept secret by a combination of the above principles in order to protect the messages and attestation pins.
8. Hardware Verification: To verify the connected AP and components are MAX78000 boards, the Unique Serial Number is checked pre-boot.

The following keys will be used for encryption and ensuring the above principles are met.

- K_{AES} = AES Key, symmetric
- K_{hash} = Hash Key for authentication

2. Build Environment

Our project uses a combination of Python, Assembly, and C language given from the reference implementation, as well as our added implementation in C as well. Using *nix-shell* to enter the Nix environment, the project can access packages and libraries necessary for building. Using the nix build environment allows the build environment to be the same across devices (i.e. Mac vs Windows) and ensure that the build environment we develop in is the same as the one used by MITRE to compile our code for testing and distribution.

Build Host Tools

To build the host tools, the implementation provided by MITRE allows us to access them in the Nix environment by running the provided build ectf-tools. The build tools are a set of scripts provided to implement functionality. This relies on the Poetry shell, and is accessible in the nix-shell by running poetry install. These tools can also be executed outside of the Poetry shell by including poetry run before the script name.

The build tool reference calls include:

1. Build Deployment: Running ectf_build_depl calls the Makefile in deployment to run. This will generate the global secrets and ectf_params header file in the application processor containing the provided pin and token in plaintext. The use of -d flag allows the addendum of the correct design file path to build the deployment.
2. Build Application Processor: Running ectf_build_ap calls the Makefile in application processor to run. This then calls a python script to find and replace the plaintext pin and token in the application proccesor params header file and hashes them into bcrypt hashes. It then compiles the application processor code and creates a build img for the AP, which will later be flashed to a MAX78000 board. This tool requires a multitude of flags to call the design (-d), the output name (-on), the pin (-p), token (-t), components (-c), the component IDs (-ids), and the boot message (-b).

3. Build Component: Running ectf_build_comp calls the Makefile in component to run. This will generate the header file containing the component prams, and then build the .img for the component, which will later be flashed to a MAX78000 board. Similar to Build AP, this tool requires a multitude of flags, including output name (-on) output directory (-od), component id (-id), boot message (-b), attestation location (-al), attestation date (-ad), and attestation customer (-ac).

4. Update Tool: In order to flash onto the MAX78000 boards, the bootloader needs to be in update mode. Holding the bootloader button on the boards while resetting will cause the onboard LED to flash blue, indicating update mode has been activated. Running the update command will flash and update the boards.

MISC Tool Reference Calls

The MISC Tools are provided by MITRE and allow communication between the host computer and the application processor, which is then able to communicate to the components. This allows for functionality requirements to be met.
The following tools can only be accessed **pre-boot**:

1. List Tool: When called, List should return the IDs of the attached and components

   ```
   poetry run ectf_list -a /dev/<PORT>
   ```

2. Boot Tool: One of the most important functionalities of the MISC tool is to boot the Medical Device. During this process, the MISC must first ensure the integrity of the device and the Components connected to it. If this integrity check fails, the boot process will be aborted. Otherwise, the MISC should print a boot message and hand off control of the AP and Controllers to the target software that will then run the Medical Device.

   ```
   poetry run ectf_boot -a /dev/<PORT>
   ```

3. Replace Tool: In the case that one of the Components fails, the MISC should allow an authorized user to replace it with a new, valid component. This should only occur if the user is able to provide a valid replacement token and if the new Component is authentic

   ```
   poetry run ectf_replace -a /dev/<PORT> -t
   <REPLACEMENT TOKEN> -i <NEW ID> -o <OLD
   ID>
   ```

4. Attestation Tool: To debug and validate the integrity of the Components, the MISC must allow an authorized user to retrieve the Attestation Data that was stored on the Components during the build process. This should only occur if the user is able to provide a valid Attestation PIN

   ```
   poetry run ectf_attestation -a /dev/<PORT> -p <AT-
   TESTATION PIN> <COMPONENT ID>
   ```

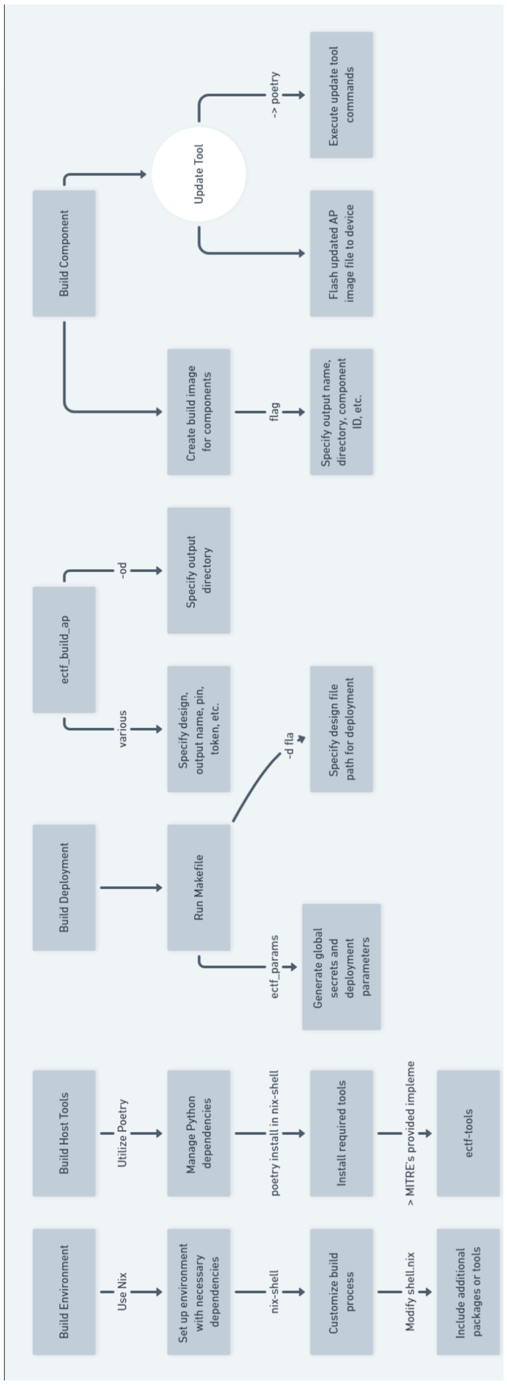

Build Environment
Use Nix
Set up environment with necessary dependencies
nix-shell
Customize build process
Modify shell.nix
Include additional packages or tools
Build Host Tools
Utilize Poetry
Manage Python dependencies
poetry install in nix-shell
Install required tools
> MITRE's provided impleme
ectf-tools
Build Deployment
Run Makefile
ectf_params
Generate global secrets and deployment parameters
-d fla
Specify design file path for deployment
ectf_build_ap
various
Specify design, output name, pin, token, etc.
-od
Specify output directory
Build Component
Create build image for components
flag
Specify output name, directory, component ID, etc.
Update Tool
Flash updated AP image file to device
--> poetry
Execute update tool commands

Figure 1: MISC Tools Flow Chart

3. Cryptography

To ensure messages are not sent in plain text, multiple cryptographic tools are implemented.

AES Encryption

AES (Advanced Encryption Standard) is a widely used encryption algorithm designed to secure sensitive data. It operates through a process of substitution and permutation, utilizing a fixed block size of 128 bits. AES employs a symmetric key system, meaning the same key is used for both encryption and decryption.

In our implementation the AP generates a random AES key for every attestation call. It then shares this with the component using public key encryption, and the components save the AES key to their device registers. This process is reliant on the MAX78000 hardware implementation, but allows us to complete encryption much faster than a software implementation would. The process involves multiple rounds of transformation, including byte substitution, shifting rows, mixing columns, and adding round keys, which collectively provide strong cryptographic security.

To further enhance the security of our design, AES256 is used, meaning the key length is 256 bits, and used to secure the attestation data sent after the attestation pin is verified. This completes security requirement 4.

bCrypt Hashing

To secure the attestation pin and replacement token, our design uses the bcrypt hashing algorithm to alter the ectf_params.h file, where the attestation pin and replacement token were originally stored in plaintext. They're replaced with bcrypt hashes using a python script bcrypt_pin.py that is called by the makefile for the application processor at compile time.

Meaning that in order for the attestation of the device to occur now, an end user provided plaintext pin or token is hashed using bcrypt. The resulting is hashes is compared to the original correct hash using a modified string compare function that checks the entire string character by character before returning if the strings match or not. This should help mitigate side channel attacks, as the regular implementation of string compare will return the moment the strings don't match, leaving it more vulnerable to side channel analysis

4. Security

Secure Send and Receive

After a successful boot, the MISC must provide a secure communications channel for the AP and Components to use. This channel will allow the AP and Components to securely send and receive messages. To securely send messages, an 256 byte packet will be first sent to the component from the AP. To ensure the integrity and authenticity of the data, a hash of the concatenation of the data, secret hash key, and random number will be included in the packet along with the original data, length of data, and chosen random number. After receiving the packet over I2C communication channel, the Component will attempt to recreate the hash based on the received plain text data and random number, as well as the shared secret hash key. This hash will be compared with the received hash using a constant-time compare function. If the hashes match, the packet will be accepted. Otherwise, an error will be returned.

AP Secure Send

Secure send creates an authentication hash composed of data to send, a secret key, and a random number. It then sends the data to the components securely, ensuring data integrity and authenticity by including the authentication hash and the random number in the packet. The AP keeps track of what random number was sent to each component, and expects to receive the same number.

AP Secure Receive

Secure receive unpacks the received packet methodically and extracts the random number, authentication hash, and original data. The AP checks if the received random number from the component equals the random number sent. It then checks if the received authentication hash is valid by comparing it with a generated hash using the extracted data, key, and expected random number.

Component Secure Send

Secure send creates an authentication hash composed of the data to send, a secret key, and the random number the component previously received from the latest communication with the AP. The component then sends data to the AP securely, ensuring integrity and authenticity by including the hash and the latest received random number in the packet.

Component Secure Receive

Secure receive extracts the random number given, the authentication hash, and the original data. It checks if the received data from the AP is valid by recreating the authentication hash from the extracted data, random number, and known secret key and compares the created hash to the extracted hash.

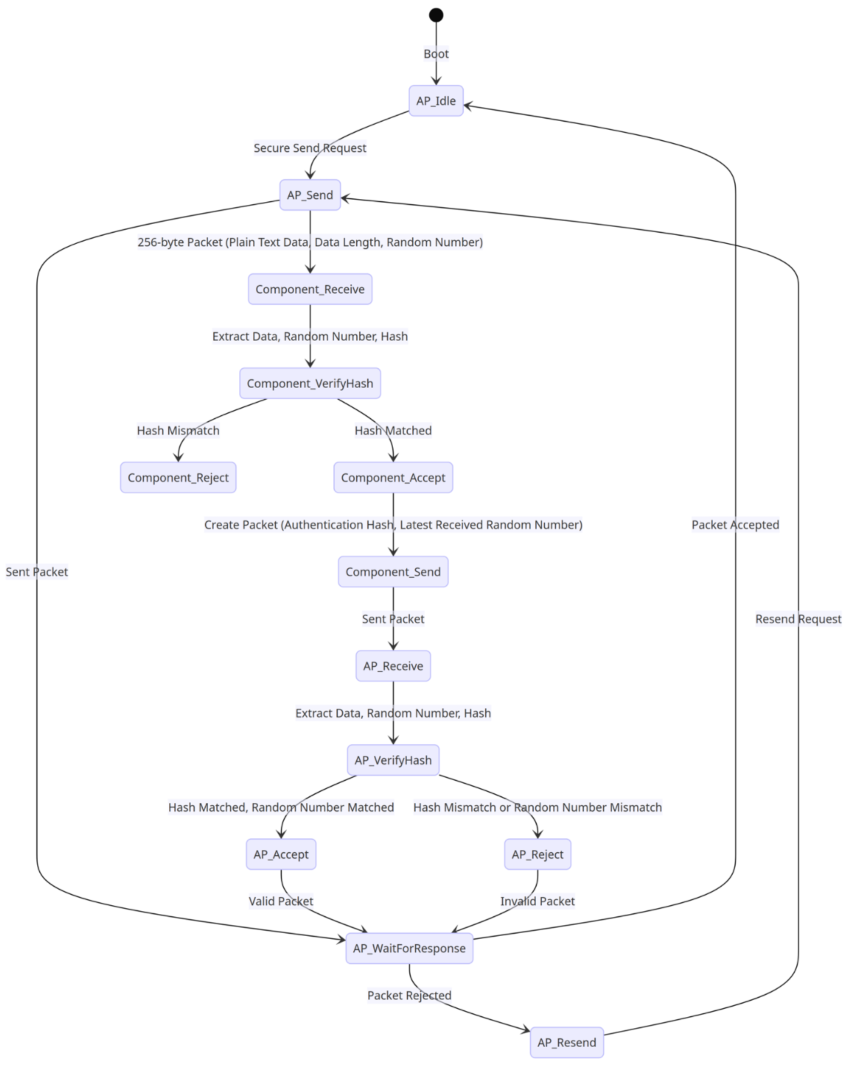


Figure 2: Secure Send and Receive

True Random Number Generator

True Random Number Generation (TRNG) implementation is facilitated through the use of the Analog Devices MSDK for the MAX78000FTHR microcontroller. This approach leverages the onboard TRNG (True Random Number Generator) capabilities provided by the SDK to generate unique random values to bolster the secure communications between the AP and Components.

This integration involves attaching a unique random number to each command sent to the component and ensuring that number is the same if and when a response message is sent back, which helps guard against side-channel and replay attacks. Since every message is unique, it cannot be used again by an attacker, even if it is intercepted. This is essential for maintaining a secure communication channel. TRNG is used to prevent replay attacks for PREBOOT command requests sent from the AP to the components. Each PREBOOT function command packet from the AP includes a TRNG number that is then expected in the request response packet for the specific command call from the component.

End-to-End Encryption

Attestation data is encrypted using AES. Given that the only sensitive data to be sent is the attestation data of a device — which will only be provided if the correct attestation pin is given to the application processor, we've implemented a hardware-based AES encryption scheme that allows the attestation request to be sent in plain text to the component, while the return data will be encrypted using AES, using a randomly generated 128 bit AES key that is generated at runtime for every single transaction through the true random number generator on the board — making it harder to influence considering that it's driven by entropy. This key is then shared through a public-key encryption scheme utilizing the X25519 elliptical curve encryption scheme.

Public Key Encryption

Public key encryption is used for all pre-boot functionality. RSA is widely considered the gold standard when it comes to public-key encryption. However, given modern advances in public-key encryption, we opted to utilize a different public-key encryption scheme known as ED 25519 and its counterpart X25519, which are based on the 25519 elliptic curve.

The encryption scheme works by having both the sender and receiver generate a public key, and a private key utilizing the 25519 curve. The public keys are then exchanged. In our code, this occurs by first generating the public and private key on the application processor. The application processor then sends it's public key to the component, which is used as the signal for the component to generate it's own public and private key pair. The component then sends it's public key back to the application processor. The benefit of the 25519 curve is that a shared secret key can be made from any combination of a receiver's public key and the sender's private key.

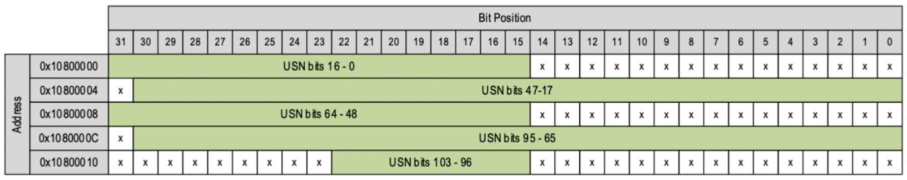
From there, the shared secret key is then used to encrypt the previously generated AES key through a XOR operation — encrypting the AES key. This encrypted AES key is then sent through the i2c channel to the recipient, where it can then be decrypted using an XOR operation against the shared secret key. The AES key can then be provided to the AES engine on both devices so that the following message can be decrypted.

Authentication Hash

An authentication hash, using a hash of a shared secret key between the AP and Components, is used to ensure the authenticity of the communication between the AP and Components. Each Preboot command packet sent from the AP to the components and the command request response packet sent from the component to the AP includes an authentication hash generated from a known secret hash key. The secret hash key is exchanged securely using Public Key Encryption for a selection of PREBOOT function calls to ensure the validity of the devices.

Hardware Verification

In order to verify the connected AP and components are not fraudulent devices, prior to boot, a hardware verification is conducted and a function is called to read the Unique Serial Number of the board from the registers of the MAX78000. This specific function should be able to only read those registers if the board is a MAX78000. If nothing is returned, we can conclude the board isn't correct and the boot sequence will reset, protecting the integrity of the device.



		Bit Position																															
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Address	0x10800000	USN bits 16 - 0																	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	0x10800004	x	USN bits 47-17																														
	0x10800008	USN bits 64 - 48																	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	0x1080000C	x	USN bits 95 - 65																														
	0x10800010	x	x	x	x	x	x	x	x	x	USN bits 103 - 96								x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 3: MAX78000 Hardware Map

5. Security Analysis

bCrypt

Given how the hashing algorithm works and generates its salt, we're reasonably certain that given the same string, there would be a different hash generated each time at compile time — ideally meaning that breaking the hash at one point of the attack phase wouldn't guarantee knowledge of the pin (assuming the same pin was used) in a different scenario (assuming the build that was compiled at a different time) as opposed to something like an md5 or other repeatable hash that would guarantee the same hash every time.

Given the constraints on the pin alone, we already know the pin will be no more than 6 characters (comprised of the alpha numeric characters between 0-9 and a-f), meaning that at worst a brute force attack would only need to try a maximum of 16^6 or 16,777,216 possible combinations before the correct pin is found — making this extremely susceptible to brute force attack on modern day computers.

Bcrypt as a hash was selected because it was designed to be is resilient to brute force attacks, as the cost function specified in the hash itself means each hash undergoes a predetermined number (6) of rounds of hashing before a final hash is outputted, and in plotting the time to generate a hash, research shows that the time required to generate a single hash scales exponentially as the cost function increases. As such, we have tuned the hash computation time to be within the competition parameters in the hopes that attempting a brute force attack against this string will slow down the hashing process.

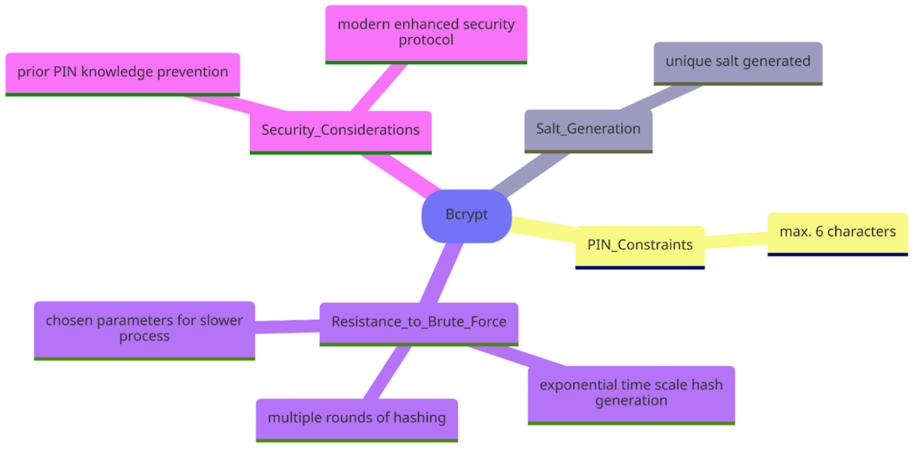

modern enhanced security protocol
prior PIN knowledge prevention
unique salt generated
Security_Considerations
Salt_Generation
Bcrypt
PIN_Constraints
max. 6 characters
chosen parameters for slower process
Resistance_to_Brute_Force
exponential time scale hash generation
multiple rounds of hashing

Figure 4: bCrypt Security

Data Protection

AES is being used to encrypt the attestation message and keep this data confidential. Given the vulnerabilities posed by placing the AES key in the global secrets file, we opted to generate and send the key required to hash the function through the public-key encryption. This takes place before computing the MD5 hash that will allow us to verify the authenticity of the packet.

We hope that by generating new keys for every single transaction, the amount of work required to actually decrypt an entire conversation or craft and inject a single message will deter most adversaries, as using side channel analysis or a memory dump to determine a single AES key or hashing key will only work for that one operation. Since these keys change with every operation, an adversary would have to find a way to determine these keys before they change, or pause the function and craft a very specific package utilizing these keys, and then freeze the receive window before the window closes.

Another potential attack vector of freezing the random number generator could work as hardware — and entropy driven random number applications tend to become more stable as the temperature decreases. However, we protect against this type of attack by keeping a list of the previous random numbers generated for hashing so that if the same random number is sent, the packet will be invalidated. Ordinarily this should be safe in normal circumstances, as the true random number generator should be able to produce a near infinite stream of random numbers that will never repeat until the competition has concluded.

End-to-End Encryption (AES)

We opted for the hardware-based AES implementation because we knew that separating it from the software implementation on the main processor would allow us a bit more security in that an attacker would actually have to do side channel analysis on the AES module itself as opposed to the arm chip to retrieve the AES key. This is also substantially faster than the software implementation provided through the simplecrypt and wolfssl libraries, as the AES module has a dedicated XOR gate to handle the encryption and decryption.

Given that the attestation command must be completed in three seconds, the hardware implementation also seemed to be the smartest solution as this would allow us to increase the work function of bcrypt as much as possible, while also allowing the use AES — which could depending on the size of the key and the message be computationally expensive to deterministically complete within the three second window provided by the specifications.

Public Key Encryption

To reduce the risk of keys being discovered or taken from the binary by storing them in plain text within the global secret header file or elsewhere within the code, we opted to use a public-key encryption scheme. GHIDRA analysis can easily retrieve plain-text keys stored in global secrets, but by using ED25519, we secure our design against this, as no keys are ever stored in plain text, and are now generated at run time and stored in memory.

Instead of RSA, we choose ED25519 and its counterpart X25519. We chose this scheme instead of RSA because it's hardened against brute force attacks, while still being incredibly fast. Given the potential for side channel analysis created by using an XOR operation on the processor itself, we've also added an extra provision within the XOR operation that will force the entire operation to always complete in a constant amount of time, meaning that timing alone won't be enough to decipher the key.

5. Future Implementations

PUF

PUF stands for Physically Unclonable Function. It is a security concept which leverages the inherent physical variations present in electronic devices and components to generate unique identifiers or cryptographic keys. By utilizing these variations, PUFs generate a fingerprint or unique response for each device, which can be used for authentication, device identification, or key generation. PUFs are particularly useful in scenarios where secure authentication or key generation is needed without relying on stored secrets, making them resistant to traditional attack methods like reverse engineering or cloning, as the key itself is unique to the board itself and is not something that can be replicated or stolen from software or binary exploitation.

For this competition we wanted to use AES for encrypting data, and through PUF had hoped to use a unique key across all boards that could be used for encryption and decryption without it being statically present. Thus, we attempted to use SRAM PUF to "generate" a unique 128-bit key that is synced up across all three boards.

First, we attempt to use the SRAM as our PUF source. To ensure it is theoretically possible, we checked the size of the MAX78000 SRAM. The SRAM is 128KB which is 1024000 bits, which results in the probability for 128-bits indexes lining up across all three boards to be ${\frac{1}{8}}^{128}/1024000$. This amounts to less than 2% of the total number of possible bits in memory, making PUF an extremely feasible strategy.

Current PUF implementations generally allow for about a 10% variation between registers on the off chance that a register flips — or has some unexpected behavior during measurement, resulting in a mismatched key. Given the importance of the key and the inability to easily brute force a possible key given a 10% error, to ensure a matching key, we realized that we would need to implement some sort of error correction scheme. Current error correction methods generally utilize 3 parity bits, but we could expand on that by using 5 parity bits, guaranteeing a greater majority of the bits for the overall PUF key would have a higher chance of being correct. This would increase the number of bits needed from 128 to 384 matching bits(128 * 3) in the event that we used a simple bit correction scheme, or 640 matching bits in the event of the 5 bit error correction scheme (128 * 5).

With a 3-bit error correction scheme, we can allow up to one-bit being "accidentally" flipped without breaking the PUF implementation. Calculating the probability of finding a 128-bit key with 3 bits of parity is $\frac{1}{8}^{128*3}/1024000 = 0.9992$ or 99.92%.

However, if we want to allow for more room for error with SRAM PUF, we can use a 5-bit parity and if the majority match up, verification is complete. The probability that 640 bits line up across all 3 boards is: $\frac{1}{8}^{128*5}/1024000 = 0.9987$ or 99.87%.

Doing more research, we unfortunately found out that not all bits of the SRAM are "stable" upon boot. This means some bits are too predictable to be considered for a SRAM implementation of PUF, and it's possible we'd never be able to find the variation needed. Given the time constraints of the competition, while this seemed originally like a great idea, we quickly realized that in order to implement something like this, we would need substantially more time.

Another challenge we found was that temperature greatly affects the "stability" of bits upon boot— in which case an attacker could freeze the bits in place i.e. using cold temperatures which will improve the stability of the bits, then discern what bits we were, and formulate the key out the values of the specific memory registers.

For the allocation of the bits, we came up with 2 possible schemes:
In our first scheme we wanted all the bits to just match. So the idea was to just go register by register on each of the boards, compare the values, and assuming a value was found against all three of the boards that match, add that register to a string, and use that bit as part of the key.

One glaring issue we realized in doing it this way is that bits could be closer to the lower numbers of memory as opposed to being randomly spread throughout the entire memory register, making it easy to get the key. One obvious solution used in industry would be to randomly determine a bit to check using the true random number generator of each of the boards. However, this would increase the amount of time to generate a key as we now need to generate up to 128,000 different numbers and track that a number hasn't already been seen.

But even then, we quickly realized that without a secure channel there would be no way for the boards to synchronize without divulging the key, as an attacker could just listen in on the channel during the synchronization process. At which point they would be able to determine what the memory locations were used or even get the actual value being compared without needing to measure the memory location.

In our second scheme, we considered randomly generating an AES key, and passing that key to the boards. For there, each board would then find the bits that would correspond to this key — meaning that every single board's memory mapping / locations of the bits that build the key would be different — while still ensuring that the key remains the same. This approach better protects the identity as a single efficient message would be used to distribute the key to the components presumably during boot time. However, the initial message could be intercepted and the corresponding AES broken to discern with the second key was and would render the entire PUF useless yet again.

That said we then ran into the issue of how much time would this take? Given that the boot time has to be completed within three seconds, we didn't think it was reasonable for any of these algorithms to complete in that time given that:

- a key would have to be generated and distributed
- that potentially the entire board's memory locations would need to be searched in the PUF state before the key is found.

This means that the only way for this encryption scheme to actually work is if somehow, this entire process happened at compile time or at build time — or at some point in time outside of when the boards were operational. The time this takes though could be substantial considering that there's at least 128 memory registers to search in the second scheme, or 128 x 3 registers a search of the first scheme (Note, the second scheme could parallelized this process which means it could complete faster).

Aside from the physical bit counts needed, finding the bits could be an issue, as using I2C communication to establish which bits to use would render PUF useless. We realized that this communication would have to be done at compile time before the item gets shipped out, so that no man-in-the-middle (MITM) attacks can be performed on boot as the key is being "generated" or as the matching bits are found.

Another concern is how bits are searched for and their parity. Searching and assigning bits starting from the beginning of the SRAM, "going down the list" and having them as our key is dangerous in the case that threat actors get their hands on our specific hardware. In theory, they could decipher the number of bits we are using as parity, perform a test of the SRAM to find all the "worthy bits" and put together a good guess to what our key is.

PUF seemed to be a secure way to secure the key, as reversing the process to generate values would be very difficult. The only way would be to individually measure memory locations, monitor their voltage and arbitration setup, and use machine learning and a trained linear regression model to determine what the output could have been. Otherwise, side channel analysis could be used to determine the AES key we use PUF to salt. Ultimately, due to technical difficulties, we decided to leave the implementation of a PUF for a future competition.

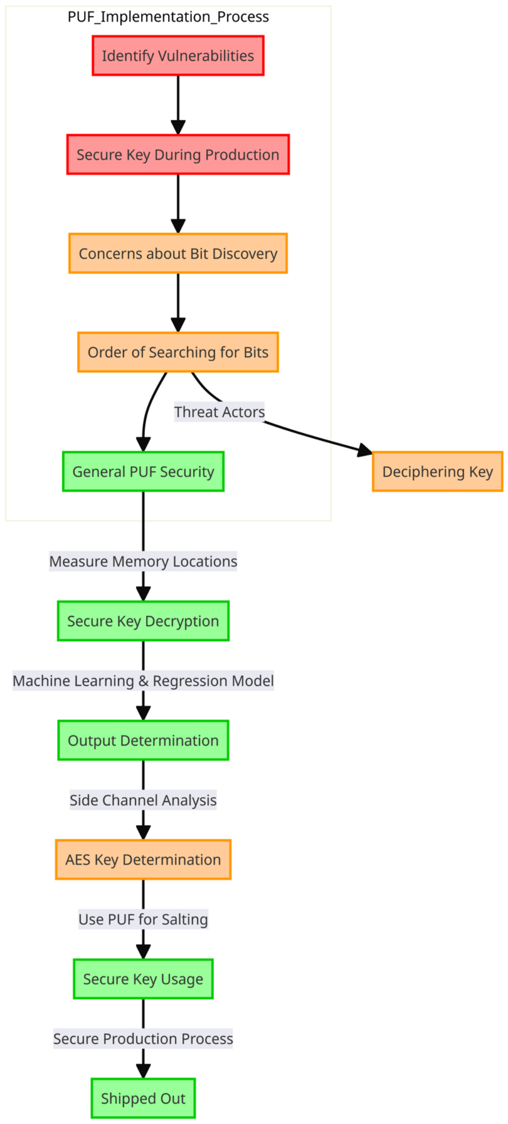


Figure 5: PUF Implementation Plan