# BatteringRAM

## University of Connecticut

**Michael Benevenuto, Zachary DiMeglio, Blake Fulton, Aniruddah Manikandan, Kevin Marquis, Kevin Romero**
**Advised by: Dr. John Chandy, PhD**

## Design Overview

The system functions with a master key encryption scheme. A random and unique signature key is generated for each signature. This signature key is used for encrypting the SHA256 hash of the data to be signed. The signature key is appended to this hash and encrypted with a master data key.

Three signatures need to be verified for the system to properly boot. The metadata of the configuration and firmware, and the unencrypted firmware itself.

## Defensive Highlight

The primary defense mechanism was a master key encryption scheme. By using a master key encryption scheme, this allowed us to use synchronous encryption algorithms and reduce the overhead of the bootloader. A signature was generated for a given dataset by hashing the data, encrypting the derived hash, appending the signature key and encrypting the entire block of data. This consumed 88 bytes of data and allowed us to use any synchronous algorithm desired. To minimize the design further, we chose the ARX cipher, ChaCha20, to improve security and reduce the processing time consumed to encrypt/decrypt data. Because encryption/decryption utilize the same function, this reduced the software overhead further.

The signature scheme worked, however securing our data was a pitfall in our design. When loading the datakey from EEPROM, the key remained on stack space no longer used and could be viewed from GDB easily. To increase security, the ChaCha20 initialization algorithm could interface directly with the EEPROM instead of loading the data from stack memory, as this would reduce the number of locations the key is stored from two to one, and would ensure security as the initial key data is mutated while generating cipher data.

Another vulnerability lied within the firmware booting process. The bootloader decrypted the firmware into SRAM for execution and verified the data using a signature loaded into the device during the firmware update process. If the signature was verified unsuccessfully, the faulty data remained in SRAM. This gave a known encryption vector and could potentially allow one to derive the secret key.

## Offensive Highlight

One planned attack was intended to take advantage of the flash hardware trojan. The attack, while not actually implemented due to time constraints, was planned to make use of the trojan to switch the clocking system of the microcontroller. By changing the clock in use from the (default) PLL to the external oscillator, an attacker could disrupt the system by potentially causing it to skip certain steps and open up major security vulnerabilities.

Another planned attack was the SCA differential power analysis, where we would collect the leaked power output from the bootloader and analyze the power consumption to find the RSA key bits. To this, we came up with an algorithm that would use a few different inputs to get traces, which would be aligned together automatically. The part we were working on implementing was trying to find the correct bits based on this information received from the traces. Due to time constraints, this was not fully implemented. This attack can be deterred by having the channel emit lots of noise to mask the actual power outputs. This would make it infeasible to try and break the RSA key by DPA. Another fix would be to vary the internal clock frequency of the device, which would also stop DPA.

The final planned attack was reverse engineering the binaries of the firmware images to gain information about the various IDs (Nav, Control, Altimeter, and Autopilot) needed to crash the aircraft. To do this, we were going to dump the binaries and find the IDs and use them to send false information to the autopilot or hinder it from receiving information from the altimeter entirely. We were planning on doing this either by giving a false height value or shutdown the altimeter and stop the bus controller from turning it back one. This attack can be fixed by obfuscating the code either by making it complicated to read or by encrypting the code to make it harder to read the binary.