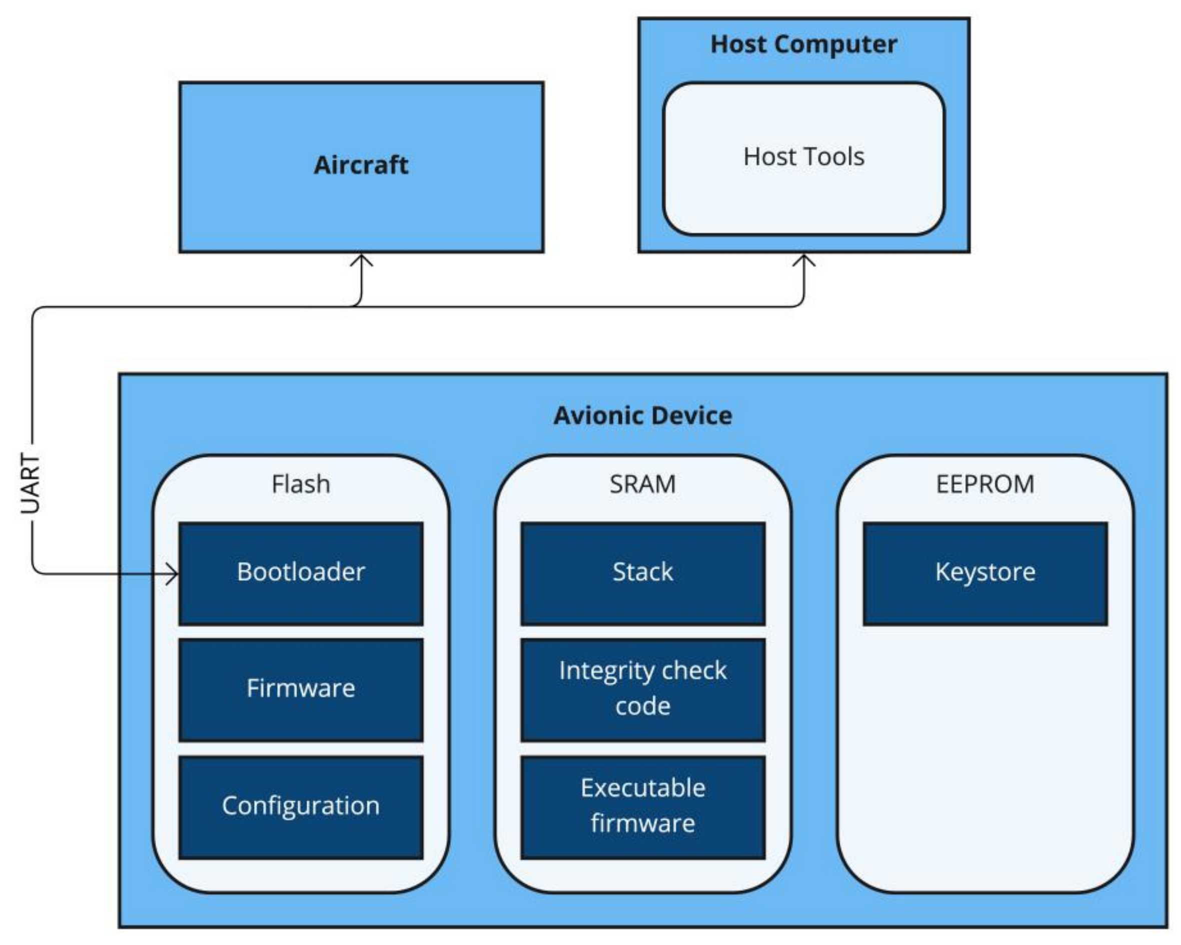
Plaid Parliament of Pwning 2022 eCTF Team

Carnegie Mellon University

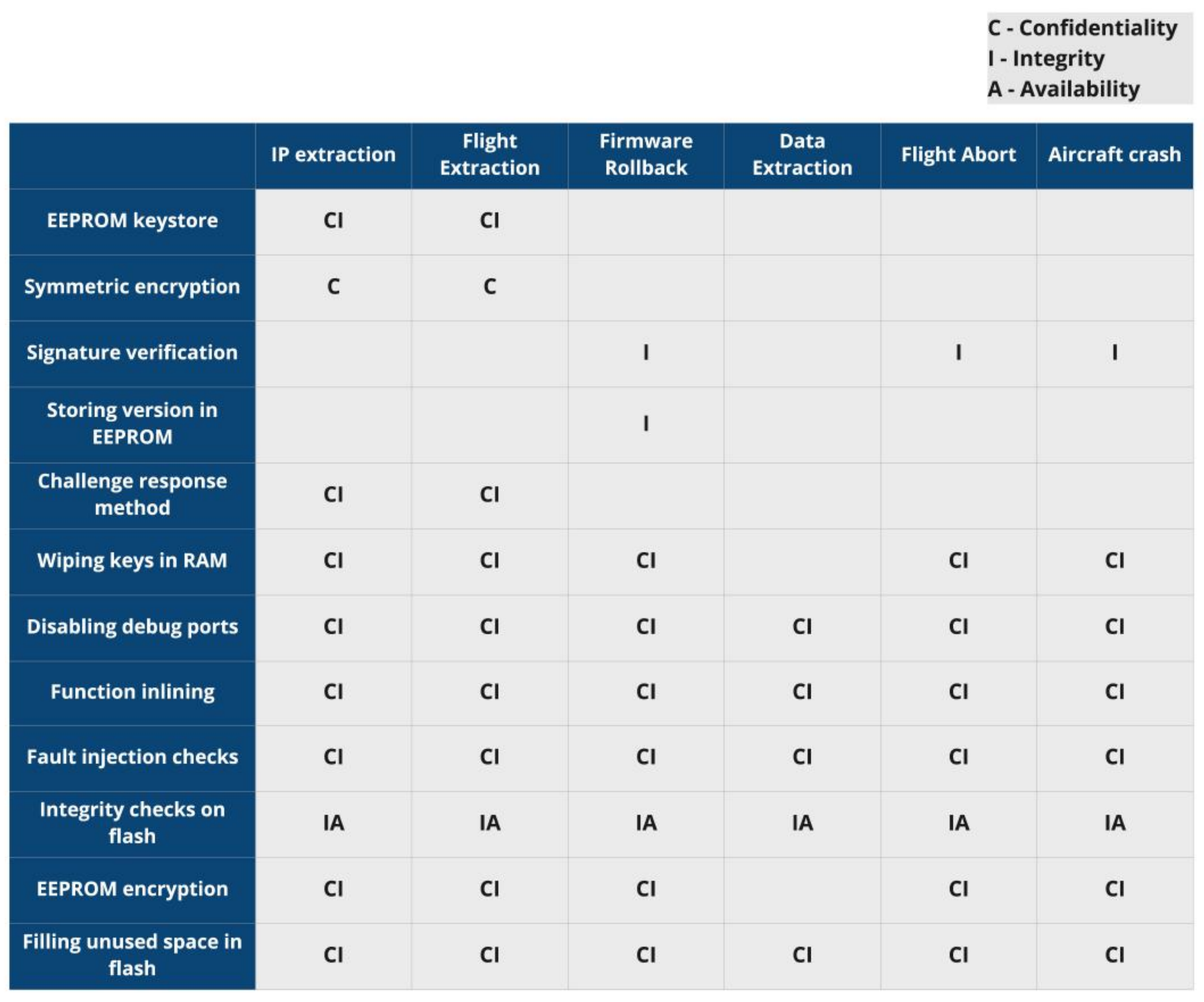
Neel Bhavsar, Robert Chen, Henry Howland, William Luca, Antonio Martorana, Palash Oswal, Nishant Arun Poorswani, Anish Singhani, Suma Thota, Hunter Wodzenski, and Maverick Woo (Faculty Advisor)

Design Overview

SAFFIRe System Architecture



Security Control to Flag Protection Mapping

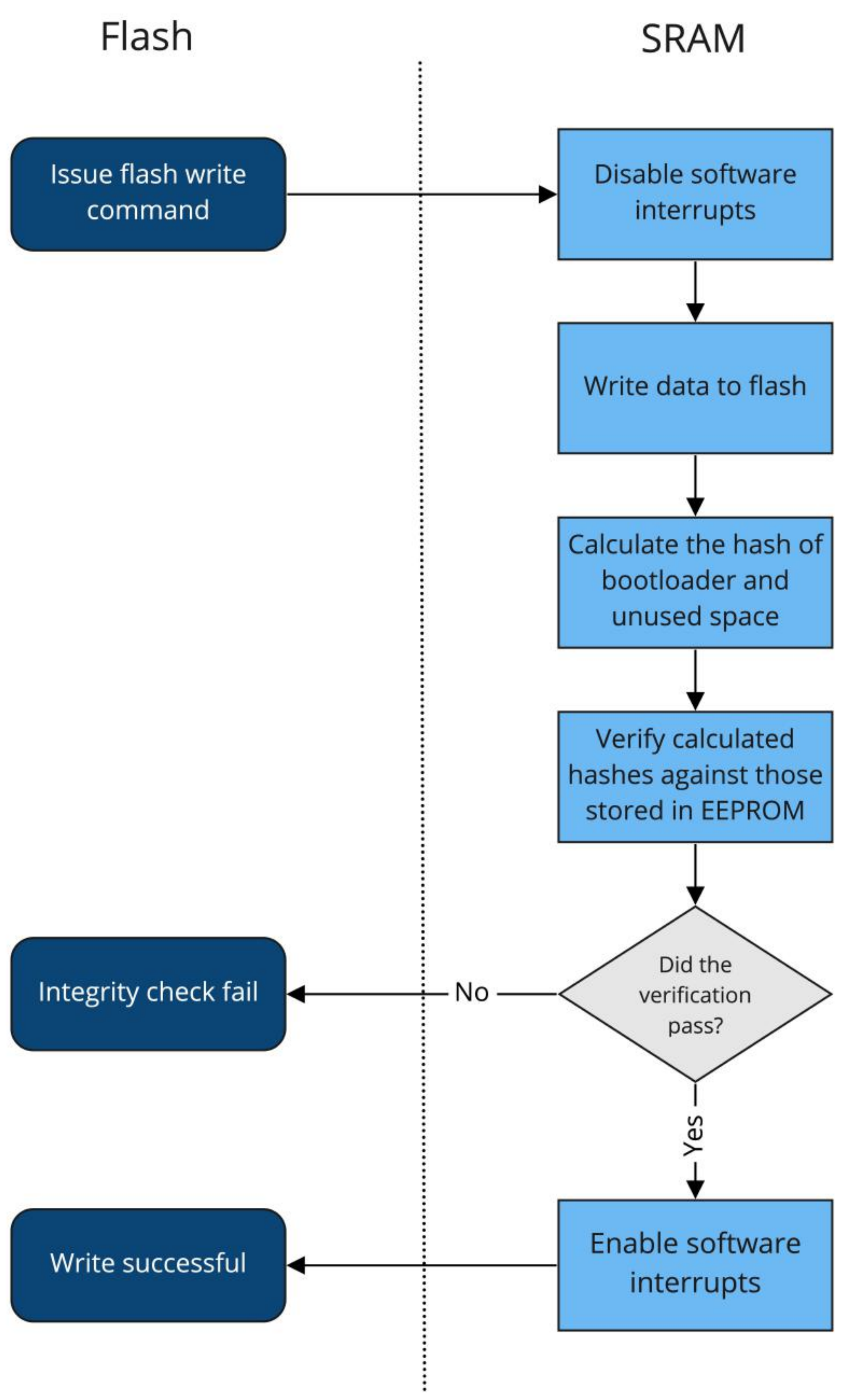


C - Confidentiality
I - Integrity
A - Availability

	IP extraction	Flight Extraction	Firmware Rollback	Data Extraction	Flight Abort	Aircraft crash
EEPROM keystore	CI	CI				
Symmetric encryption	C	C				
Signature verification			I		I	I
Storing version in EEPROM			I			
Challenge response method	CI	CI				
Wiping keys in RAM	CI	CI	CI		CI	CI
Disabling debug ports	CI	CI	CI	CI	CI	CI
Function inlining	CI	CI	CI	CI	CI	CI
Fault injection checks	CI	CI	CI	CI	CI	CI
Integrity checks on flash	IA	IA	IA	IA	IA	IA
EEPROM encryption	CI	CI	CI		CI	CI
Filling unused space in flash	CI	CI	CI	CI	CI	CI

Defensive Highlight

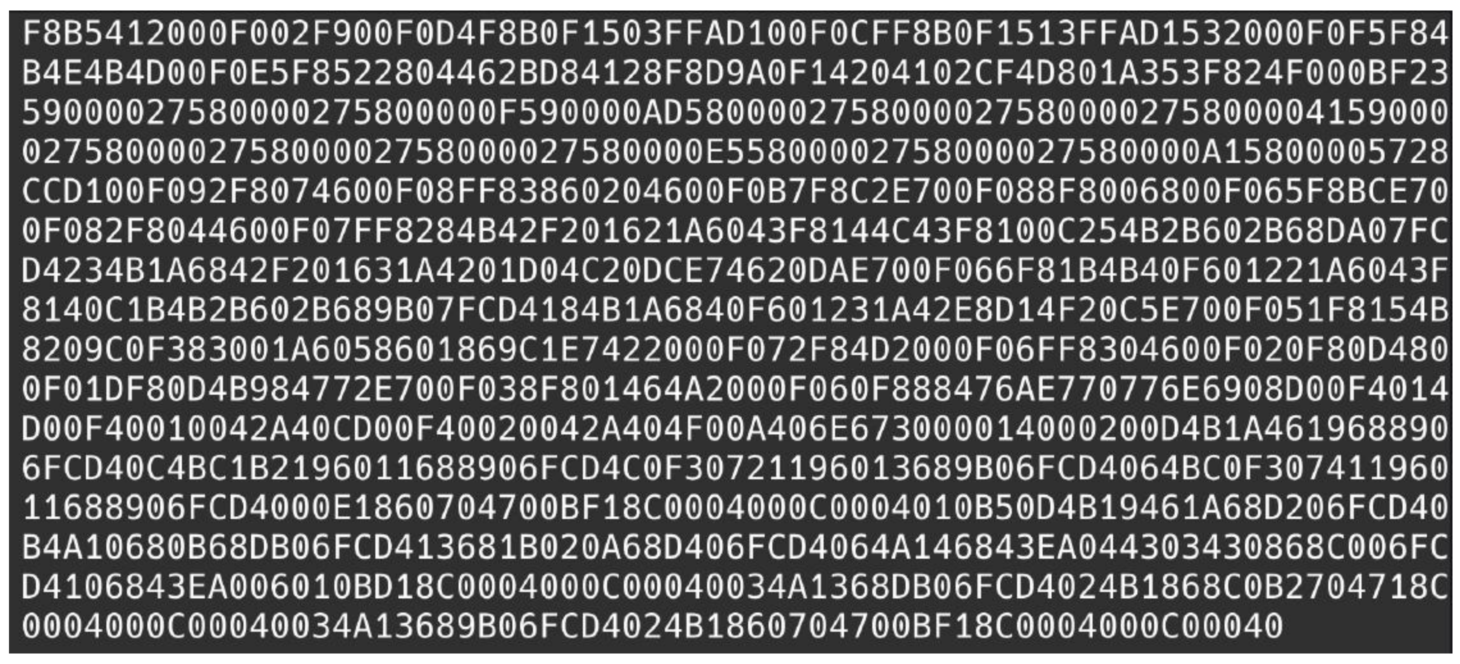
Bootloader Integrity Checking from SRAM


Flash
SRAM
Issue flash write command
Disable software interrupts
Write data to flash
Calculate the hash of bootloader and unused space
Verify calculated hashes against those stored in EEPROM
Did the verification pass?
No
Yes
Integrity check fail
Enable software interrupts
Write successful

Offensive Highlight

Trojan + uFIRE Mini Bootloader Shellcode

uFIRE's entire binary shown below:



```
F8B5412000F002F900F0D4F8B0F1503FFAD100F0CFF8B0F1513FFAD1532000F0F5F84
B4E4B4D00F0E5F8522804462BD84128F8D9A0F14204102CF4D801A353F824F000BF23
5900002758000027580000F590000AD58000027580000275800002758000004159000
0275800002758000027580000275800000E558000027580000275800000A15800005728
CCD100F092F8074600F08FF83860204600F0B7F8C2E700F088F8006800F065F8BCE70
0F082F8044600F07FF8284B42F201621A6043F8144C43F8100C254B2B602B68DA07FC
D4234B1A6842F201631A4201D04C20DCE74620DAE700F066F81B4B40F601221A6043F
8140C1B4B2B602B689B07FCD4184B1A6840F601231A42E8D14F20C5E700F051F8154B
8209C0F383001A6058601869C1E7422000F072F84D2000F06FF8304600F020F80D480
0F01DF80D4B984772E700F038F801464A2000F060F888476AE770776E6908D00F4014
D00F40010042A40CD00F40020042A404F00A406E673000014000200D4B1A461968890
6FCD40C4BC1B2196011688906FCD4C0F30721196013689B06FCD4064BC0F307411960
11688906FCD4000E1860704700BF18C0004000C0004010B50D4B19461A68D206FCD40
B4A10680B68DB06FCD413681B020A68D406FCD4064A146843EA04430343086 8C006FC
D4106843EA006010BD18C0004000C00040034A1368DB06FCD4024B1868C0B2704718C
0004000C00040034A13689B06FCD4024B1860704700BF18C0004000C00040
```

Features:

- Automated attacks for 6/6 flags
- 4 full pwns within 10 minutes of trojan build
- Replicate much of SAFFIRe's core functionality
- The entire bootloader is less than 600 bytes
- Sleep functionality defeats anti-trojan encryption
- uFIRE host tools add additional capabilities

Design Overview

SAFFIRe is a bootloader and a set of host tools that implement a secure firmware and configuration update for an avionic device. It provides confidentiality and integrity guarantees for a device that provides critical information to an aircraft. The bootloader is flashed on a microcontroller and are controlled through the host tools on a Linux machine over UART. The host tools support the build processes for the device bootloader and implement the firmware protection and configuration protection.

SAFFIRe uses cryptography to provide confidentiality and integrity for the data in transit and at rest. To securely manage the cryptographic keys, a keystore was implemented using the EEPROM. The keystore is locked before handing execution to the firmware which only unlocks after a full power cycle. The keys are generated by the host tools and initialized in the EEPROM. The device makes sure that the keys in the RAM are securely set to zero after usage.

Confidentiality and integrity is ensured using encryption and signatures whenever the data is sent from the host tools to the bootloader. Our implementation uses AES for the payload encryption and ed25519 for signature verification. The firmware and configuration updates are provided by encrypting and signing the payload. The device verifies the signature first and upon verification failure of the signature it refuses to decrypt the payload. The device also verifies the signature of the stored firmware and configuration on every command issued by host tools. The encryption and signature verification provides protection for the IP extraction and flight extraction flags.

Strict firmware versioning is implemented by storing the current version in EEPROM. The device refuses to flash any version that is older than the current installed version except for the version #0. The source code was reviewed for this feature with potential integer conversions and integer overflows in mind. This, along with the signature verification, protects the firmware rollback flag.

The readback function is authenticated using a challenge response method. The private keys are only accessible to legitimate users of the device. The device generates a challenge and sends it to the host tools. The host tools respond by signing the challenge using the private key. After successful authentication, the device allows one readback operation to continue.

On a boot command the device hands the control over to the firmware. A potential attack on the booted firmware may leak sensitive keys used by the bootloader. The design prevents these attacks by wiping out the RAM before passing the control. Before booting firmware, SAFFIRe performs various integrity checks to verify that the firmware and configurations are untampered. This protects the flight abort and aircraft crash flags. If an integrity violation is detected then the device sends an error to the host tools, locks the keystore and refuses any further commands. The integrity checks ensure that the flash contents are not tempered with by a trojan.

The bootloader build system makes use of function inlining to avoid using stack return addresses in order to prevent stack overflow attacks. To thwart fault injection attacks the build system adds a random amount of delay and redundant comparisons at compile time. The bootloader also performs an integrity check of its own code in an attempt to detect code execution. These protections aim to protect the data extraction flag that requires unauthorized code execution.

To further deter attackers, the EEPROM contents are encrypted with a one-time pad generated at compile time. The unused space of the flash is also filled with random data which is verified as part of the integrity checks. This allows the bootloader to detect the presence of a trojan even if the trojan attempts to write malicious code in the unused portion of the flash.

Defense Highlight

Our flash_write_and_verify function will write/erase flash, and then verify the integrity of the bootloader code against a precomputed hash. This function is relocated to an unused region of SRAM on SAFFIRe startup. NVIC Interrupts are disabled at the entry-point and re-enabled afterwards before return. This prevents a malicious trojan from modifying the flash instructions at the bootloader entry-point (0x5800) by leveraging a software reset to jump to that location before we perform critical integrity checks. Performing integrity checks of the bootloader code allow us to verify the integrity of the flash after a potential flash modification operation. The trojan with the flash controller cannot tamper with the SRAM, and the function cannot be interrupted (with software resets). This window allows us to establish a root of trust for performing flash writes and erases. Integrity of flash code is verified before control flow returns to flash as long as the system is running, thus preventing the trojan from hijacking control flow by overwriting code. This security feature is defeated if the trojan performs a delay in operation followed by hard-resetting the system, an attack vector we used. Future improvements to this function would ensure more interrupt coverage to prevent side channels from soft resetting the system.

Attack Highlight

Attackers have the ability to insert a “flash trojan” into the microcontroller, which would trigger on flash-memory writes and could tamper with the contents of flash. To avoid tripping integrity-checks during provisioning or runtime, we wrote a trojan which would trigger after 10 unique writes to the config-version pointer, and then write a piece of shellcode (**uFIRE**) starting at 0x5800 (the entry-point) in flash, allowing us to reset the device to jump into our shellcode.

The shellcode we developed for this purpose is called **uFIRE** ("micro-Firmware Installation RoutinE"). The fully-assembled uFIRE bootloader fits in ~600 bytes of ARM-Thumb code (small enough that it can fit within the trojan itself) and allows us to get 6/6 flags against any design (usually within just 10-15 minutes after trojan provisioning)!

We do this by minimizing the set of features in the shellcode itself:

- UART interface
- Read-memory
- Write-memory
- Write-flash
- Erase-flash
- Read-EEPROM
- Jump-arbitrary
- Boot (with fake release message)

And implementing higher-level functionalities in our Python-based host scripts:

- *Unlock* - Sends the unlock sequence and waits for an acknowledgment.
- *Unhide EEPROM* - Sets the MMIO registers to enable and then reset the EEPROM.
- *Dump EEPROM* - Reads out the entire contents of EEPROM.
- *Load firmware* - Takes unencrypted firmware + config, writes it to the relevant locations in RAM/flash, then reads it back to verify it was written correctly.
- *Boot and Monitor* - Jumps to the installed firmware and then monitors the UART to see what data is written out.
- *Pivot* - Allows us to write and load a second-level shellcode which fakes the team's specific boot process (i.e. one team had a integrity check during boot, but since the integrity check was static, we were able to just hardcode it into our second-level shellcode and pivot to it) including a fake release message.

How to get each flag using uFIRE:

- *Flight Extraction* - Use dump-EEPROM to read encryption keys from EEPROM, then decrypt their protected configuration files.
- *IP Extraction* - Same as Flight Extraction
- *Firmware Rollback* - After decrypting the v1 firmware (see "IP Extraction"), use load-firmware script to load, then boot it.
- *Data Extraction* - Dump EEPROM; read plaintext flag
- *Flight Abort* - Use the load-firmware script to load the decrypted v2 firmware (see the "IP Extraction" bullet point) with a nonsense configuration, and then pivot to a bit of code that fakes the team’s “boot” process (with a fake release message to satisfy the “boot” host-tool) and then start the aircraft.
- *Aircraft Crash* - Use the load-firmware script to load our special firmware that exploits the altimeter in order to crash the aircraft. Then, boot it using the same process as Flight Abort.

Given the constraints of the competition, we don’t know of any way to defend against this attack using purely software-based measures.

1) We cannot prevent trojan from running - sufficiently-complex trigger logic makes it possible to build a trojan that goes undetected during provisioning; and hanging the system bus after writing uFIRE would allow an attacker to reset the device before any subsequent integrity checks run.

2) We cannot protect secrets from the trojan - in order for the device to continue functioning across reboots, all secrets needed to decrypt the firmware must be stored in such a way that can be accessed by code running on the microcontroller, which means uFIRE can access it too (the solution to this would be to use a dedicated hardware coprocessor with higher security expectations).

3) Since uFIRE is written permanently into flash, any transient memory protection, EEPROM hiding, etc. is worthless because it all goes away after a reboot.

4) We can imagine a system where the boot command uses a protocol to attest its correctness to the boot host-tool. However, the uFIRE could be modified to make a backup of any data it modifies and use that to spoof whatever the integrity check is.