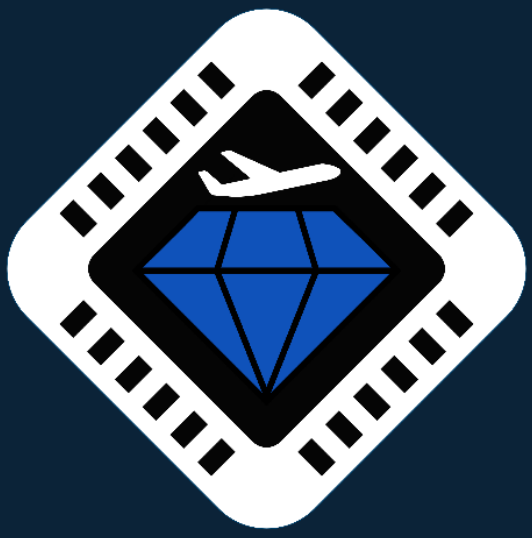
Technical Specifications:

SAFFIRe: a Secure Avionics Flight Firmware Installation Routine



MITRE | SOLVING PROBLEMS FOR A SAFER WORLD™

Contents

1 System Implementation

1.1 System Architecture

The reference SAFFIRe design provided by the organizers uses Docker to build container images for the host computer and avionic device. The Docker container architecture has been designed to run on both the MITRE-hosted development servers and local computers that will interact with the physical hardware. The emulated and physical set-ups are shown in Figure 1 and Figure 2, respectively.

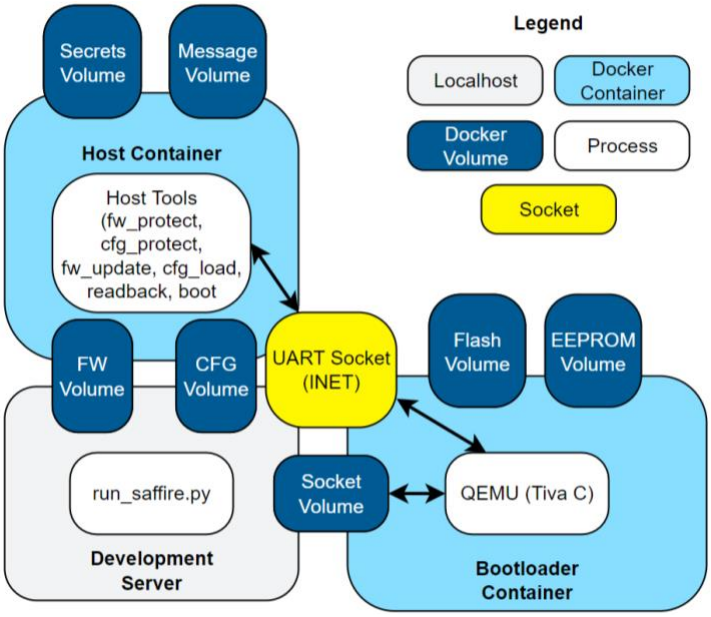


Figure 1 Emulated System Architecture

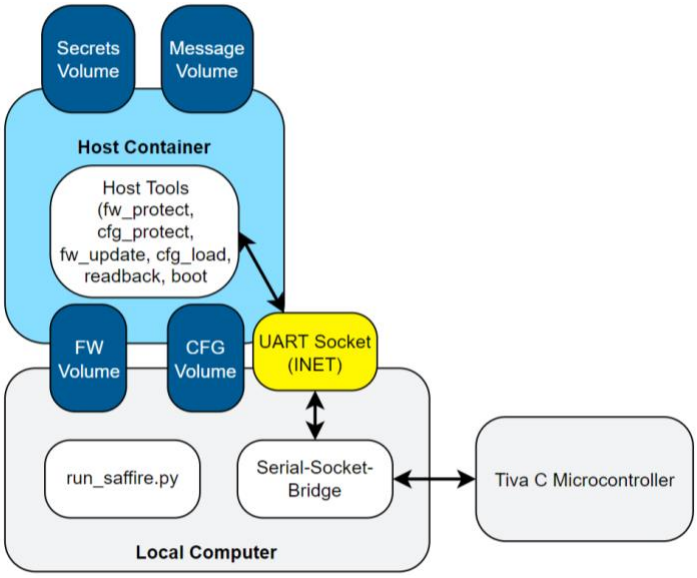


Figure 2 Physical System Architecture

1.1.1 Docker Containers

SAFFIRe is implemented in two main Docker containers: the host container, and the bootloader container. The host container is built first in the system and holds all the final host tool programs for use in the system, the host secrets, and the compiled bootloader. The bootloader container holds all scripts needed to launch the emulated device, as well as additional binaries needed to load the physical and emulated devices with the SAFFIRe bootloader.

1.1.2 Docker Volumes

The emulated and physical systems both use Docker volumes to share files across containers and enable the containers to save their state between run cycles.

1.1.2.1 Firmware and Configuration Volumes

Unprotected firmware and configuration files should be placed in this volume on the host computer so they can be used as inputs to the firmware and configuration protect host tools. The host tools will place all protected output files here so they can be used later to perform firmware updates and configuration loads.

1.1.2.2 Secrets Volume

This volume is hidden from the user to simplify system use, and holds the secrets generated during a SAFFIRe system build. If a host tool needs to modify the secrets during a protect or update command, the new secrets contents are available on the next host tool execution. This volume is cleared when building a new system.

1.1.2.3 Message Volume

This volume is hidden from the user to simply system use. The message volume stores release messages received by the device boot tool to be accessed by the aircraft simulation later. This volume is cleared when building a new system.

1.1.2.4 Socket Volume

This volume is only used in the emulated system and contains sockets for attaching to GDB on and collecting side-channel traces from the emulated device. This volume does ***not*** contain the UART socket for communication between the host tools and device; the UART socket is an INET type socket that is shared between the localhost and Docker networks.

1.1.2.5 Flash and EEPROM Volumes

These volumes are hidden from the user to simplify system use. The Flash and EEPROM volumes contain emulator-generated files for preserving the contents of Flash and EEPROM memory, respectively, across executions of the emulator. These volumes are cleared when loading a new SAFFIRe bootloader onto the emulated device.

1.2 Repository Structure

Every design repository must have the following structure. Files and folders in ***bold italics*** cannot be modified in a submission; These files implement the SAFFIRe system architecture and are managed by the organizers. You may modify them to help your development and testing, but the organizers will use unmodified versions for testing and attack phase operation.

- bootloader – code for the SAFFIRe bootloader

    - inc – bootloader header files
    - lib – external libraries
        - tivaware – Tivaware driver library for the Tiva C microcontroller
            - bootloader.ld – bootloader linker script
            - makedefs – common definitions for the bootloader and Tivaware
    - src – bootloader source files
    - Makefile – SAFFIRe bootloader makefile
    - README.md
- dockerfiles – dockerfiles for building deployment images. ***Note:*** *these files may be modified, but their names must be kept the same*.
    - 1_build_saffire.Dockerfile – creates a provisioned SAFFIRe system image including the host tools and bootloader
    - ***2_create_device.Dockerfile*** – creates a standalone avionic device image with SAFFIRe bootloader and system startup binaries
- ***firmware*** – folder to store firmware images
    - ***example_fw.bin*** – example unprotected firmware
- ***configuration*** – folder to store mission configuration images
    - ***example_cfg.bin*** – example unprotected mission configuration
- host-tools – code for the SAFFIRe host tools
    - boot – device boot tool
    - cfg_load – mission configuration load tool
    - cfg_protect – mission configuration protect tool
    - fw_protect – firmware protect tool
    - fw_update – firmware update tool
    - generate_secrets – deployment secret generator
    - ***monitor*** – example firmware monitor
    - readback – readback tool
- ***platform*** – code for launching the bootloader
- ***tools*** – code for running the deployment
    - ***run_saffire.py*** – top-level script for running SAFFIRe operations
    - ***serial_socket_bridge.py*** – script for connecting a physical device to host tools containers

1.3 SAFFIRe Bootloader

1.3.1 Tiva Tm4C123G Emulator Support

The Tiva C TM4C123G microcontroller is emulated using a custom QEMU machine. The emulator implements several key peripherals that teams may use in their SAFFIRe bootloaders. Although designs will be run on the physical microcontroller, the SAFFIRe bootloader must also work in the emulated system and therefore may not rely on peripherals that are not implemented in QEMU. The following peripherals/registers are supported in the emulated device:

- SYSCTL Peripheral Ready Registers
    - 16/32-bit timers, GPIO ports, UART interfaces, watchdog timer[1], EEPROM

[1] Although the watchdog timer claims to be implemented, it does not correctly generate timeout interrupts or flags

- SYSCTL Peripheral Present Registers
  - 16/32-bit timers, GPIO ports, UART interfaces, watchdog timer[1], EEPROM
- SYSCTL Software Reset Registers[2]
  - 16/32-bit timers, GPIO ports, UART interfaces, watchdog timer[1], EEPROM
- SYSCTL Run-mode Clock Gate Control (RCGC) Registers
  - 16/32-bit timers, GPIO ports, UART interfaces, watchdog timer[1], EEPROM
- Flash Memory Controller
  - FMC Register[3]
  - FMA Register
  - FMD Register
  - FMPREx Registers
- EEPROM
  - EESIZE Register
  - EEBLOCK Register
  - EEOFFSET Register
  - EERDWR Register
  - EERDWRINC Register
  - EEDONE Register
  - EEHIDE Register
  - EEDBGME Register

If there is functionality missing from the emulator that multiple teams would benefit from, the eCTF organizers may add it to the emulator during the Design Phase.

1.3.2 Bootstrap and Reset Interface

The microcontroller is programmed with a bootstrapping application that places the SAFFIRe bootloader in Flash at address 0x5800 and loads the EEPROM data into EEPROM memory. To provide teams with a way to quickly restart the SAFFIRe bootloader, the bootstrapper provides a "soft reset" interface that skips the bootstrapper update logic and immediately restarts the installed SAFFIRe bootloader. On the emulated device, this reset is triggered by externally sending data over UART1, and on the physical device this reset is triggered by pressing switch button SW2 on the Tiva C development board. The button is connected to GPIO F0, so setting the corresponding general-purpose header low will also trigger the reset.

The soft reset on the physical board also provides a handshake mechanism for controlling when the SAFFIRe bootloader begins executing. When asserting the reset (by setting GPIO F0 low or holding button S2), a quick system setup function runs and then turns the on-board LED blue by setting GPIO F2 high (this pin is also routed to a general-purpose header on the development board). The reset function will wait until the input signal on GPIO F0 is released before executing the bootloader.

[2] The Peripheral Software Reset registers can be written and read to, but they do not alter the state of the peripherals; only whether or not they appear as "ready". One exception to this is SREEPROM, which does clear all EEPROM hidden block settings

[3] The FMC2 register is ***not*** implemented

1.4 Requirements and Restrictions

1.4.1 Time Requirements

The SAFFIRe bootloader must complete each operation within the following time limits:

Table 1 SAFFIRe Time Limits

Operation	Max Time for Completion[4]
Start Up	5 seconds
Firmware Update	5 seconds
Configuration Load	10 seconds
Boot	5 seconds
Readback	10 seconds

1.4.2 Size Requirements

The SAFFIRe Bootloader must support the following sized components:

Table 2 SAFFIRE Size Limits

Operation	Range for Size
Compiled SAFFIRe Bootloader	See 1.4.3
Firmware	(0B, 16KB[5]]
Configuration	(0B, 64KB]
Release Message	[0B, 1024B]
Readback	10 seconds

1.4.3 Bootloader, Firmware, and Configuration Addresses

When booting the avionic firmware, the firmware must be loaded into SRAM at address 0x20004000. The flight configuration must be stored in Flash at address 0x00030000.

The SAFFIRe bootloader must be compiled so its first instruction is at address 0x5800. The bootloader may not use any Flash memory located below this address.

[4] Time requirements apply to physical hardware only
[5] 1KB = 1024B

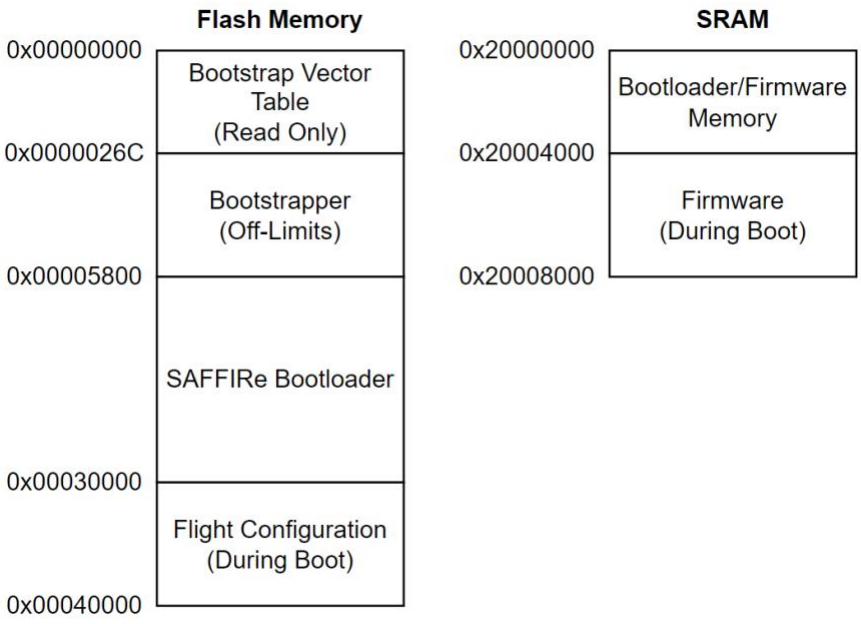


Figure 3 Avionic Device Memory Map

1.4.4 Bootloader Reset Handling

As specified in the Rules Document, the bootloader must run in a continuous loop and accept multiple firmware updates, configuration loads, and readback commands in a row. Additionally, the bootloader must also continue to operate after the soft reset is triggered (See 1.3.2). However, once the bootloader boots the installed firmware and configuration, it may depend on a full power cycle to run again.

1.4.5 Flash Memory Protections

SAFFIRe bootloaders ***may not*** permanently commit Flash memory write protections. This is to ensure that keyed attack-phase devices can be used for multiple designs. SAFFIRe bootloaders ***may*** set Flash memory write protections on each boot as long as it does not prevent the design from working after a soft reset.

1.4.6 EEPROM Block Hiding

SAFFIRe bootloaders ***may*** use EEPROM block hiding as long as it does not prevent the design from working after a soft reset.

1.4.7 Interrupt Vector Table

If a team wants to use interrupts in their SAFFIRe bootloader, they must place a copy of the vector table at address 0 in SRAM or Flash after the start of their bootloader. This functionality is implemented in the Tivaware driver library in `tivaware/driverlib/interrupt.c:IntRegister()`.

2 Handoff Phase Submission

When submitting your design to either the automated testing service or the organizers for verification, your design must reside on a public facing git repository (github, gitlab, etc.). If your school provides a private git server, you are welcome to use it for development. However, the server does not support access from public networks, you will have to make it available through a public service. When you are ready to submit to either the automated testing service or the organizers, the organizers will provide an account that you must give access to your source code.

Each version of your design that is submitted must be tagged with a version number starting from "v1.0". If the tests fail while evaluating the submission, you will be informed and given a log of the test process. When submitting the next version of your code, you must increase the major version number (e.g., "v1.0 to "v2.0"). Minor version numbers are reserved for cases at the organizers' discretion.

When testing a submission, the evaluation process will automatically replace any folders that are not allowed to be modified with a clean version. Therefore, you may change files in non-modifiable folders for the purposes of testing your design, but any changes or new files will not be included when the organizers test your submission. See the repository structure for a list of folders that are not modifiable.

3 Functional Requirements

The Rules Document introduced each functional step in SAFFIRe, and how they should affect firmware images, mission configuration images, and the microcontroller state. This section formally defines the inputs and outputs of each step. Each step in the system is executed through a script (tools/`run_saffire.py`) either which executes either a Docker build or run command. We define the requirements for each SAFFIRe step by its invocation with the top-level script.

The following sections will describe the API for each host tool.

3.1 System Build

Create a Docker container from scratch containing the SAFFIRe host tools and bootloader. Your team may modify `host_tools/1_create_host_tools.Dockerfile` to generate any secrets, build them into the bootloader, and save them for later use with the host tools.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py build-system
        (--emulated|--physical)
        --sysname <NAME>
        --oldest-allowed-version <VER>
```

INPUTS:

- <NAME>: The name of the SAFFIRe system. Use different names for different developers to avoid clobbering each other's containers
- <VER>: The oldest allowed version of firmware for the bootloader to accept

OUTPUTS:

- Container tagged <DEPL>/host_tools:latest
  - /host_tools: A directory containing the host tools
    - fw_protect
    - cfg_protect
    - fw_update
    - cfg_load
    - readback
    - boot
  - /secrets: A directory containing any needed secrets
  - /bootloader: A directory containing bootloader binaries
    - bootloader.axf: ELF executable of the bootloader
    - bootloader.bin: Raw binary of the bootloader
    - eeprom.bin: Data to load into EEPROM immediately after bootloader installation

3.2 Load Device

Load the SAFFIRe bootloader and EEPROM binaries into the microcontroller. Teams may not modify any code for this step; it is provided only as a reference for the command line arguments.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py load-device
        (--emulated|--physical)
        --sysname <NAME>
```

INPUTS:

- <NAME>: The name of the SAFFIRe deployment

OUTPUTS:

- Emulated device image or physical microcontroller with the newest SAFFIRe bootloader and EEPROM binaries installed

3.3 Launch Bootloader

Launch the SAFFIRe bootloader created during the system build phase. Teams may not modify any code for this step; it is provided only as a reference for the command line arguments.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py launch-bootloader
        (--emulated|--physical)
        --sysname <NAME>
        --sock-root <SOCK_ROOT>
        --uart-sock <UART_SOCK>
```

INPUTS:

- <NAME>: The name of the SAFFIRe deployment
- <SOCK_ROOT >: Local folder to mount that will contain the sockets to communicate with the bootloader
- <UART_SOCK>: Internet socket port number for the bootloader UART

OUTPUTS:

- Additional sockets for side-channel emulator and emulated debugger in <SOCK_ROOT>/

3.4 Firmware Protect

Protect a firmware image using the host tools and secrets created during system build. The firmware protection tool must accept a raw firmware binary, firmware version, and release message which are packaged into one protected firmware image. The tool may access the `/secrets` folder.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py fw-protect
        (--emulated|--physical)
        --sysname <NAME>
        --fw-root <FW_ROOT>
        --raw-fw-file <RAW_FW>
        --protected-fw-file <PROTECTED_FW>
        --fw-version <FW_VERSION>
        --fw-message <RELEASE_MSG>
```

RESULTING HOST TOOLS CALL:

```
$ /host_tools/fw_protect
        --firmware <RAW_FW>
        --version <FW_VERSION>
        --release-message <RELEASE_MSG>
        --output-file <PROTECTED_FW>
```

INPUTS:

- <NAME>: The name of the SAFFIRe deployment
- <FW_ROOT >: Local folder to mount as the firmware volume
- <RAW_FW>: Raw input firmware file name (must locally be in <FW_ROOT>)
- <PROTECTED_FW>: Protected output firmware file name (will be placed locally in <FW_ROOT>)
- <FW_VERSION>: Version number to package with the firmware (16b unsigned int)
- <RELEASE_MSG>: Release message to package with the firmware (Between 0 and 1024B long)

OUTPUTS:

- Protected firmware image located in <FW_ROOT>/<PROTECTED_FW>

3.5 Mission Configuration Protect

Protect a mission configuration image using the host tools and secrets created during system build. The mission configuration protection tool must accept a raw configuration binary and package it into one protected configuration image. The tool may access the `/secrets` folder.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py cfg-protect
        --sysname <NAME>
        --cfg-root <CFG_ROOT>
        --raw-cfg-file <RAW_CFG>
        --protected-cfg-file <PROTECTED_CFG>
```

RESULTING HOST TOOLS CALL:

```
$ /host_tools/cfg_protect
        --input-file  <RAW_CFG>
        --output-file  <PROTECTED_CFG>
```

INPUTS:

- <NAME>: The name of the SAFFIRe deployment
- <CFG_ROOT >: Local folder to mount as the configuration volume
- <RAW_CFG>: Raw input mission configuration file name (must locally be in <CFG_ROOT>)
- <PROTECTED_CFG>: Protected output mission configuration file name (will be placed into <CFG_ROOT>)

OUTPUTS:

- Protected mission configuration image located in <CFG_ROOT>/<PROTECTED_CFG>

3.6 Firmware Update

Send a protected firmware image to the bootloader for installation. The firmware update tool must accept a protected firmware image and socket to connect to the bootloader with. The tool may **not** access the contents of the `/secrets` folder.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py fw-update
        (--emulated|--physical)
        --sysname <NAME>
        --fw-root <FW_ROOT>
        --protected-fw-file <PROTECTED_FW>
        --uart-sock <UART_SOCK>
```

RESULTING HOST TOOLS CALL:

```
$ /host_tools/fw_update
        --socket <UART_SOCK>
        --firmware-file <PROTECTED_FW>
```

INPUTS:

- <NAME>: The name of the SAFFIRe deployment
- <FW_ROOT >: Local folder to mount as the firmware volume
- <PROTECTED_FW>: Protected input firmware file name (must locally be in <FW_ROOT>)
- <UART_SOCK>: Internet socket port number for the bootloader UART

3.7 Mission Configuration Load

Send a protected mission configuration image to the bootloader for installation. The configuration update tool must accept a protected configuration image and socket to connect to the bootloader with. The tool may **not** access the contents of the `/secrets` folder.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py cfg-load
        --sysname <NAME>
        --cfg-root <CFG_ROOT>
        --protected-cfg-file <PROTECTED_CFG>
        --uart-sock <UART_SOCK>
```

RESULTING HOST TOOLS CALL:

```
$ /host_tools/cfg_load
        --socket <UART_SOCK>
        --configuration-file <PROTECTED_CFG>
```

INPUTS:

- <NAME>: The name of the SAFFIRe deployment
- <CFG_ROOT >: Local folder to mount as the configuration volume
- <PROTECTED_CFG>: Protected input configuration file name (must locally be in <CFG_ROOT>
- <UART_SOCK>: Internet socket port number for the bootloader UART

3.8 Readback

Request a region of the installed firmware or configuration from the bootloader. The readback tool must accept a data type (firmware or configuration), a number of bytes to read, image and socket to connect to the bootloader with. The tool may access the `/secrets` folder.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py readback
        --sysname <NAME>
        --uart-sock <UART_SOCK>
        --rb-region <RB_REGION>
        --rb_len <RB_LEN>
```

RESULTING HOST TOOLS CALL:

```
$ /host_tools/readback
        --socket <UART_SOCK>
        --region <RB_REGION>
        --num-bytes <RB_LEN>
```

INPUTS:

- <DEPL>: The name of the SAFFIRe deployment
- <UART_SOCK>: Internet socket port number for the bootloader UART
- <RB_REGION>: Image region to read back. Must be either “fw” or “cfg”
- <RB_LEN>: Number of bytes to read from the region

3.9 Device Boot

Request the bootloader to boot the installed images in preparation for talking to the aircraft. The boot tool must accept a file to store the returned release message in and a socket to connect to the bootloader with. The tool may **not** access the contents of the `/secrets` folder.

SAFFIRe CALL:

```
$ python3 tools/run_saffire.py device-boot
        --sysname <NAME>
        --msg-root <MSG_ROOT>
        --boot_msg_file <BOOT_MSG_FILE>
        --uart-sock <UART_SOCK>
```

RESULTING HOST TOOLS CALL:

```
$ /host_tools/boot
        --socket <UART_SOCK>
        --release-message-file <BOOT_MSG_FILE>
```

INPUTS:

- <NAME>: The name of the SAFFIRe deployment
- <MSG_ROOT>: Local folder to mount as the release message volume
- <BOOT_MSG_FILE>: Release message output file name (will be placed locally in <MSG_ROOT>
- <UART_SOCK>: Internet socket number for the bootloader UART

4 Hardware Trojans

During the Attack Phase, attacking teams will have the opportunity to insert their own hardware trojans into the Flash Memory Controller of emulated avionic devices. Trojans will be able to read and modify the Flash memory with the following constraints:

- **Trojans are subject to the bit-flip restrictions of Flash memory.** If a bit of the Flash memory has already been programmed from a 1 to a 0, a trojan may only set that bit back to 1 by erasing[6] the entire 1024-byte block containing that bit. This restriction does not apply for programming individual bits to 0.
- **Trojans are NOT subject to read/write-protections placed on the Flash memory.** If the SAFFIRe bootloader has marked regions of Flash as write-protected, a trojan may bypass those protections.
- **Trojans may be triggered upon any write to the Flash FMC, FMD, and FMA registers.** Trojans are always executed when one of these registers is accessed by software. A trojan may choose whether to act based on the state of the registers and the Flash memory.
- **Trojans receive the values of the Flash FMC, FMD, and FMA registers.**
- **Trojan functionality is memory-limited.** Trojan function will be provided a fixed amount of memory or "registers" to use for computation.

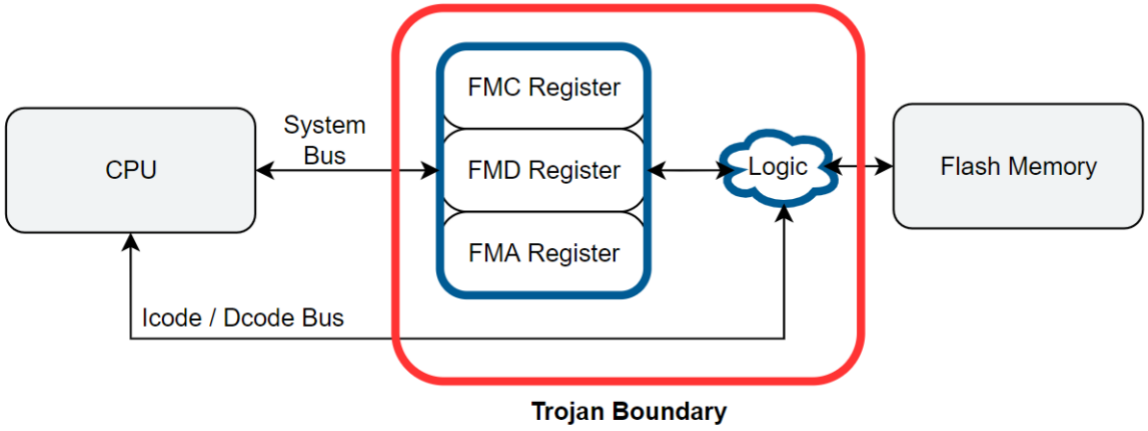


Figure 4 Flash memory controller trojan location

[6] Flash erase sets all bits to 1 and executes on blocks of 1024-bytes