Anteaters

University of California, Irvine

Jalen Chuang, Kush Dave, Jacob Huang, Zhuoyi Yang, William Jeon, Jeein Kim, Chenhan Lyu, Lucas Chang
Advised by: Dr. Ian Harris, PhD

Design Overview

Hardware Security Features
- ARM Cortex Memory Protection Unit to prevent code execution from unexpected areas of memory
- Code ran in volatile memory protected from hardware trojan

Software Security Features
- Encryption via masked AES-CBC 256 to provide confidentiality on the firmware and configuration [1]
- Digital signatures via Ed25519 to ensure that data has not been tampered with [2]
- Challenge-Response based authentication to verify trusted hosts to allow readback
- Integrity self check and host tools challenge to detect trojans
- Link-time object location randomization for binary hardening

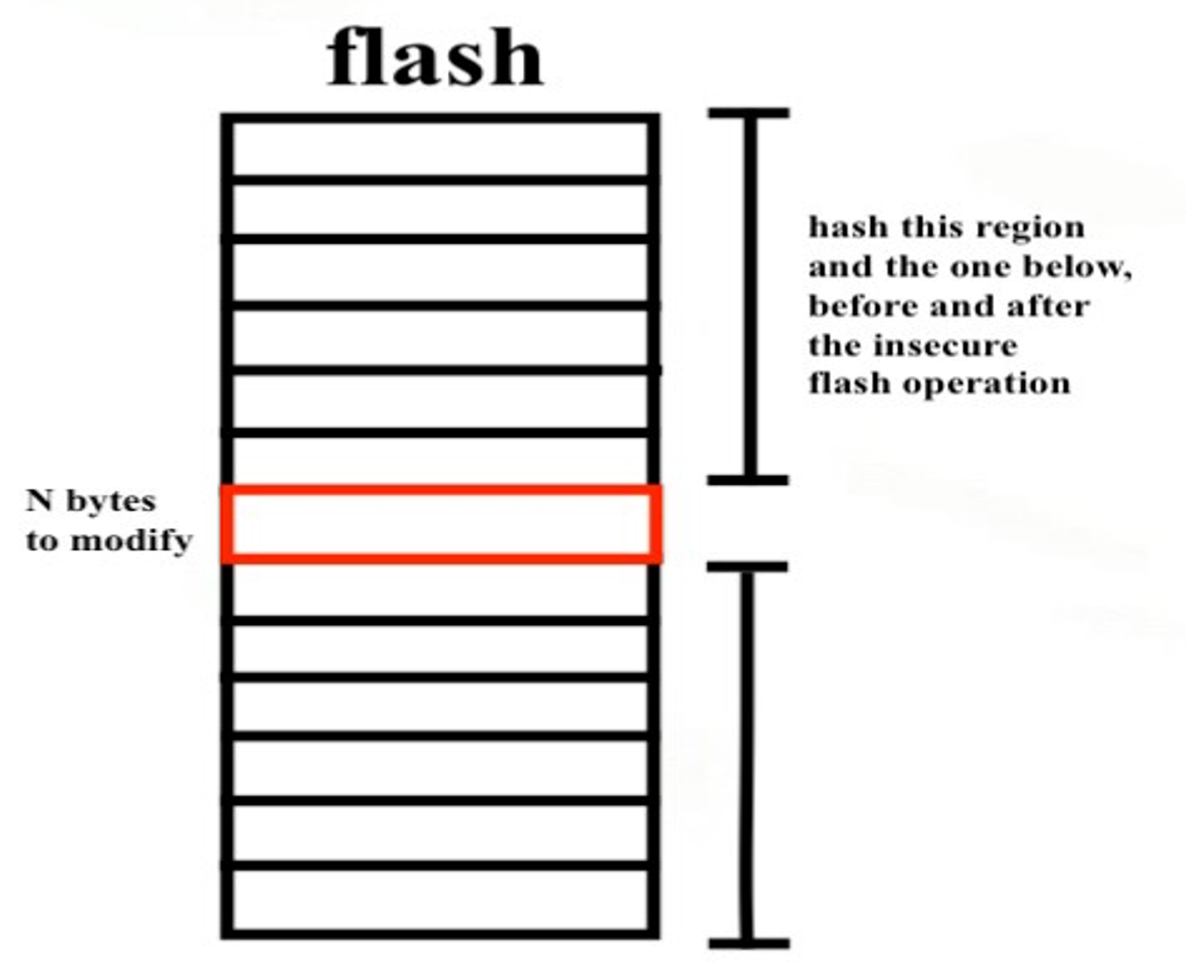
1	Confidentiality	● Encryption ● Side channel resistant implementations
2	Integrity	● Digital Signatures ● Integrity Challenge
3	Authentication	● Randomized Challenge-Response
4	Trojan Defense	● Memory Protection ● Secure Flash Writes ● Linker Randomization

Defensive Highlight

A core defense is secure flash memory writes. The trojan can be triggered on any flash write, and some safety-critical branches can be skipped or altered.

We protect our flash memory by running code in **SRAM** - a region that the trojan cannot change. This region acts as a sort of trusted zone.

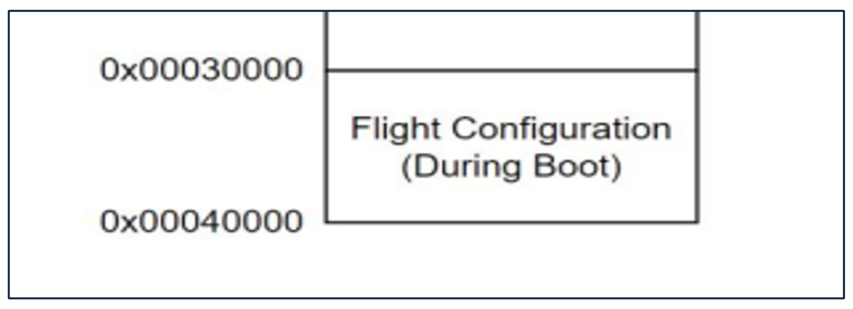
Any flash writes are surrounded with an integrity check barrier which computes a cryptographic hash of memory that is not being changed (seen below). It also verifies each word is written as expected. If the check fails, secrets are deleted.


flash
hash this region and the one below, before and after the insecure flash operation
N bytes to modify

An area to be improved in this defense is that it does not validate that the final contents of a batch write to flash are as expected. This oversight allows the trojan to arbitrary data to memory when a batch write occurs.

Offensive Highlight

All designs need to write the configuration in **plaintext** to flash at some point. This can be exploited to leak the flight configuration for many designs!



The configuration needs to be written to a constant area during boot

A trojan can take advantage of this write and another property of the SAFFIRe system - hard resets. A hard reset can occur and returns bootloader execution to the start.
The attack follows this outline:
- The bootloader writes the plaintext to flash.
- The trojan activates some time after (possibly immediately), preventing future writes from flashing and locking the configuration in place.
- The trojan overwrites the start of the bootloader with its code.
- The attacker performs a hard reset, executing the trojan's newly written code!
- The code exfiltrated the flight configuration and/or other sensitive data over UART.

We failed to implement this attack due to time constraints, as it took too long for us to implement and test the trojan.

Protections for this attack could involve randomization, stopping the trojan from knowing when to activate:
- Writing to the configuration area a random amount of times.
- Writing random fake data to the configuration area first.

References

1. *C implementation of AES with masking technique to prevent side-channel analysis attacks.* GitHub. (2020, October 7). Retrieved April 23, 2022, from https://github.com/CENSUS/masked-aes-c

2. *ED25519: High-speed high-security signatures.* (2017, January 22). Retrieved April 23, 2022, from https://ed25519.cr.yp.to/

Overview of the design.

This year's MITRE eCTF challenge was to implement a secure bootloader. The provided insecure design didn't give any security guarantees. Therefore we decided to implement the following features:

- To protect IP such as proprietary firmware and configuration, we used AES-256 in CBC mode during our firmware and configuration update. This ensured that the firmware and configuration were encrypted on the device and an adversary couldn't have access to the secrets even if the device was under attack. The AES implementation also used random masking to prevent against power side channels.
- To ensure that correct firmware and configuration images were loaded into the memory, we implemented digital signatures using the Ed25519 signature scheme. This scheme was chosen because of its resistance to side channel attacks. Signatures would be computed at firmware/configuration encryption time and loaded onto the device during firmware/configuration update. Additionally, we also signed the firmware version number with the firmware to ensure that the version number wasn't tampered with to prevent firmware rollback attacks.
- A big part of designing our secure bootloaders was to detect/prevent trojan based attacks. The condition under which the trojan was activated was during a flash write. Hence we performed integrity checks over Flash every time we interacted with the bootloader to ensure that the trojan hadn't modified any part of our bootloader..
- We disabled the J-TAG interface to prevent adversaries from attaching a debugger to our bootloader and possibly interfere with our code and recover protected secrets.
- We also made sure that sensitive code which performed security checks such as integrity checking, decryption and signature verification was loaded into the SRAM to prevent the trojan from modifying this sensitive code while it was running.
- To ensure that the host tools were a valid authority, we performed authentication using the Challenge Handshake Authentication Protocol. We performed this by sending a random string to the host device, generating the SHA-256 hash of the string concatenated with a symmetric key using the host tool, and then verifying the hash with the bootloader.
- To prevent adversaries from recovering our encryption keys after the cryptographic primitive was performed, we manually zero the keys stored on the stack.
- We enabled the Memory Protection Unit and removed certain permissions from regions of flash where the bootloader code/data is stored. This is to prevent the bootloader code to be overwritten, the data section to be non executable and the unused part of the flash to be unusable for both writing and execution.
- We applied link time randomization to prevent adversaries from reverse engineering our code and prevent ROP based attacks on our system. Moreover, randomization makes it very difficult to craft generic attacks for multiple builds.

Defensive Highlight

Our defensive highlight is our secure flash write and erase functions located in SRAM. These security functions are necessary because a hardware Trojan in the flash controller can potentially activate whenever a flash write or erase (but not read) occurs. It can write to arbitrary addresses in the flash and bypass flash memory protections.

Our secure flash functions ensure the integrity of flash operations by providing that the flash controller only writes the expected number of bytes (N) to the anticipated addresses in the flash. They achieve this goal by computing a SHA-256 hash of all the flash memory except for the N bytes at the expected addresses before and after the unsafe flash operation. If the two hash values aren't equal, the bootloader assumes the hardware Trojan activated and writes to locations in the flash it wasn't supposed to.

However, there is a limitation to the integrity that these functions provide. Because they ignore the flash addresses where the N bytes are expected to be written, a Trojan can modify the N bytes of data that get written to flash without causing the bootloader to panic. This oversight could be correct by changing the way the hashes are computed. The "before" hash should be computed by hashing the entire flash region, but with the old N bytes replaced with the new N bytes at the expected addresses. The insecure flash operation would be called, and then the "after" hash should be computed by hashing the entire flash region. Then, the bootloader can verify the two hashes are equal just as before. This modification would ensure the integrity of the whole flash memory, assuming the Trojan in the flash controller doesn't affect the flow of control of the bootloader.

Offensive Highlight

We came up with a theoretical attack on the confidentiality of the flight configuration that is difficult to defend taking advantage of the requirements of the firmware. Because the configuration needs to be written in plaintext to flash by the bootloader at some point, a trojan can take advantage of this.

The attack works with an hardware trojan that triggers some time after unprotected configuration is written to flash. If the trojan then blocks all further writes by the bootloader to flash, the configuration is forced to be left unprotected. The trojan can then write code to the beginning of the bootloader space.

After then, if the technician performs a hard reset, the trojan's code will be run and can exfiltrate the contents of the configuration over UART. This attack is difficult to defend against since a hard reset cannot be disabled and the provided bootstrapping code cannot be modified, meaning that the trojan can always run code of some kind after a hard reset. With more individual planning on designs, this technique can be used to leak more sensitive information such as keys.

We were not able to perform this attack because we were unable to prepare trojans in the attack phase's time constraints. In addition, although this attack is quite general, it requires some amount of customization for different designs that store unprotected configurations at different points.

A protection against this attack would be to write a randomized number of times to the regions of memory where the unprotected configuration is stored. In this way, it would be difficult for the trojan to know when to activate after the real configuration is written. This defense is not perfect, as if there is any way for a trojan to differentiate a legitimate configuration write then the trojan can determine when to trigger. The trojan can also simply pick a random time to activate, which provides a random chance of working in contrast to how much time the bootstrapper is willing to spend writing fake data to flash.