Implementation of Firmware Load Protocol

Team Name: Masq-ware

Yogeshwaran Logashanmugam - 03606143 **Neha Mishra - 34969979**
Solomon Sagar Albert Jayaraj - 57219856 **Saurabh Jain – 39684083**

I. ABSTRACT

One of the many security objectives in modern systems-on chip (SoC) designs requires an unbreakable methodology to implement firmware upgrade. Although, efforts are made in this direction to approach towards developing a protocol which can counter all possible vulnerabilities, there are still certain scenarios where the implementation breaks. This document tries to describe some of the attacks which can undermine the secure firmware load implementation. Also, to overcome these flaws, an enhanced set of protocols, which builds upon the basic protocol are identified and implemented.

II. INTRODUCTION

Major concern while designing an SoC device is what all scenarios to be considered while updating the device firmware. Once, the device is deployed in field, firmware patching plays a very important role in life cycle of the device. Firmware update might be needed for bug fixing, requirement change or feature enhancement which are very crucial for a device to operate for a long-term without replacement. Therefore, firmware load protocol should be resilient enough to ensure that the firmware image is authenticated and attempts to activate a malicious firmware image are prevented. This project tries to achieve this requirement through few protocol designs which, under specified conditions provides hindrance against presence of malicious attacker. Also, this project showcases the pitfalls in the design of underlying protocol on the top of which more advanced and effective implementations are done.

III. TYPES OF PROTOCOLS AND THEIR IMPLEMENTATION

We created 3 processes – one each for driver, device and cryptographic engine. The inter process communication between the three were achieved through two levels of sockets. The driver connects to the device, which acts as a server and the device in turn sends a request to the cryptographic engine, which acts as a server to the device. Each server is multithreaded that implies that there will be a separate thread running for each client request at the server's side. The usual flow is – driver sends the firmware to the device through socket. Device receives the firmware, copies it to the Isolated Memory, which is the memory exclusive to the device and from where the firmware starts executing after the authentication is passed. After the device copies the firmware to the isolated memory, it sends a request to the cryptographic engine to authenticate the firmware. In our case we took string "abcd" as the authentic firmware. The cryptographic engine accepts the request from the device and processes the firmware contents and returns passed if the string was "abcd", else it returns failed. The device jumps to the isolated memory to start running from the updated firmware if the status from the cryptographic engine was passed. The implementation of all the 6 protocols were done using the above flow. However, for recreating the attack, we implemented a slightly different approach for the lock unlock and unlock lock protocols.

1. Basic Firmware load protocol - F_b

To recreate the attack, we initially sent the request from the driver for authentication with the correct firmware. While the request is being processed by the cryptographic engine, we send in another request

from the driver with a malicious firmware. This gets copied to the isolated memory by the time the first thread returns the authentication status as passed. Once the first request returns passed and the device is about to jump to the isolated memory to start executing the updated firmware, the firmware has been already corrupted by the second request. Now, instead of "abcd" the memory contains the malicious firmware and the device is running it. This is the exact scenario of how a TOCTOU attack would take place.

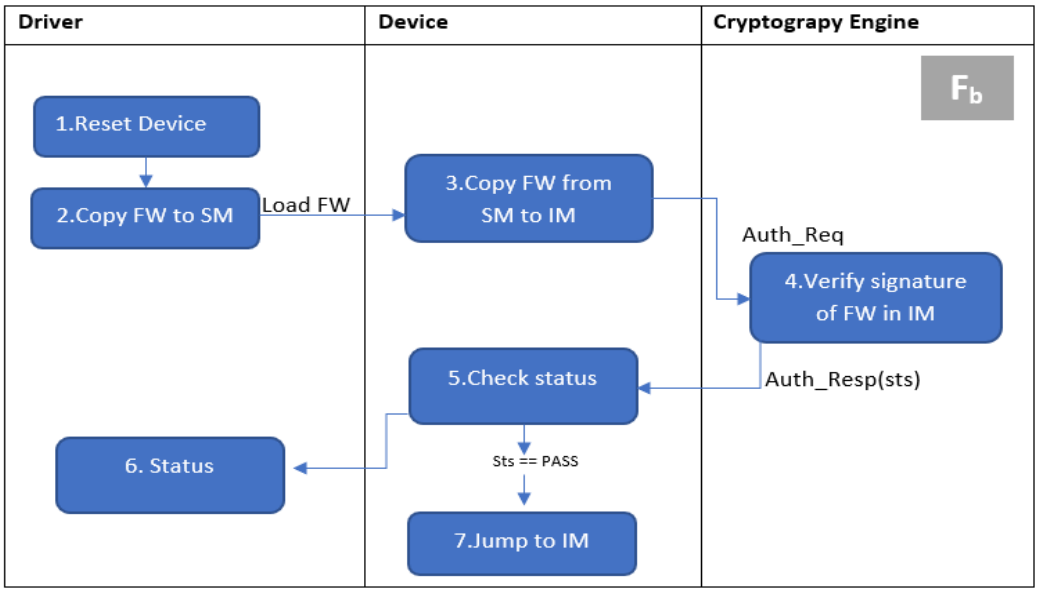

Figure.1. Basic Firmware load protocol

2.Protocol – F_a

In this protocol, there is a boolean that is set to false once the firmware is copied to the isolated memory and the firmware can be copied only if the boolean is true. This makes sure that the firmware can be copied only once during a whole cycle of the program. However, the flag is again set to true if there is a reset. This being an improved version than before, still could be possibly attacked. We pass the reset status initially from the driver along with the firmware. If the driver is reset again, the isolated memory can again be updated with the malicious firmware after sending in the authentic firmware "abcd" during the first run. We simulated this scenario and found that even with additional check of boolean variable, the implementation is compromised.

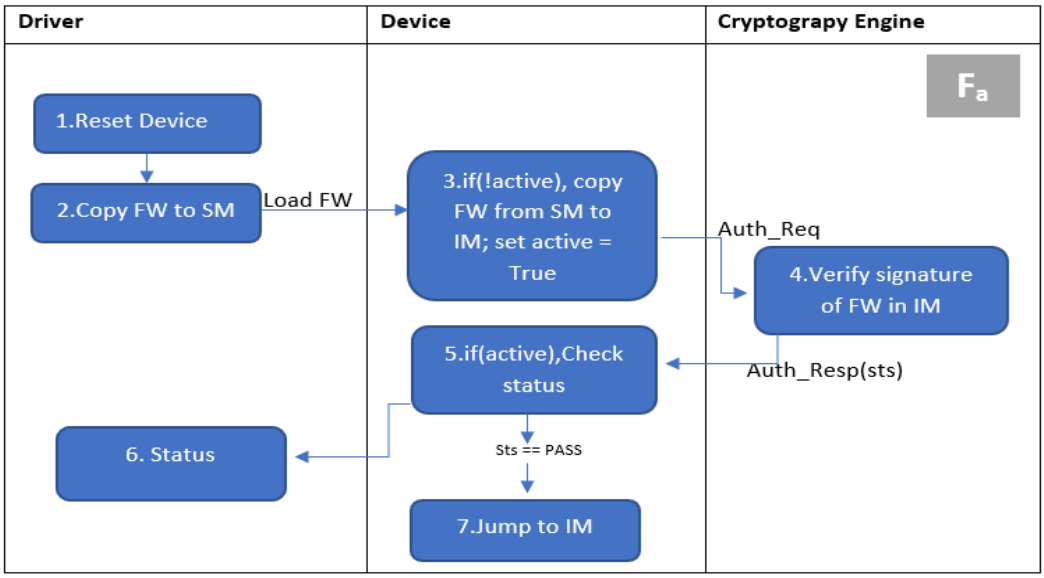

Fig.2. F_a -Attack scenario sequence

3. Protocol – F_{lu}

In this protocol, there is an additional layer of protection for the isolated memory. The isolated memory is locked and is not allowed to be updated until the authentication is fully performed. However, here as well, when the authentication has been performed and the isolated memory is unlocked, the malicious thread can come into action and copy its contents to the isolated memory. To recreate the attack, we followed a single multithreaded program wherein there are two threads – one for the authentic firmware and the other for the malicious one. The scheduling of these threads was done using locks and condition variables. The attack here was recreated by allowing the malicious thread to take over once the initial thread finishes of its authentication process and unlocks the isolated memory. By the time the first thread has its result passed and is intending to jump into the isolated memory, the isolated memory is already containing the corrupted firmware.

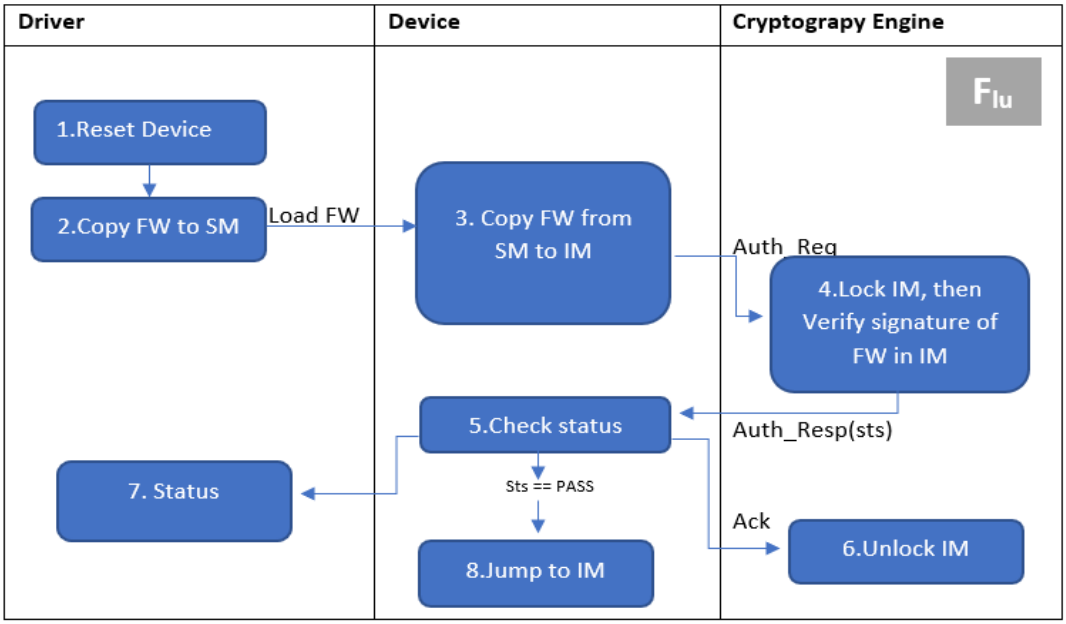

Fig.3. F_{lu} -Attack scenario sequence

4. Protocol – F_{ul}

This is similar to the previous protocol except for the difference in the locking scheme of the isolated memory. Here we first unlock the isolated memory, copy the firmware to it and then do the authentication and finally unlock the isolated memory. Here, the attack was recreated in a similar approach as mentioned for the lock unlock protocol. The thread containing the corrupted firmware was copied after the first thread's authentication is completed and the isolated memory is unlocked.

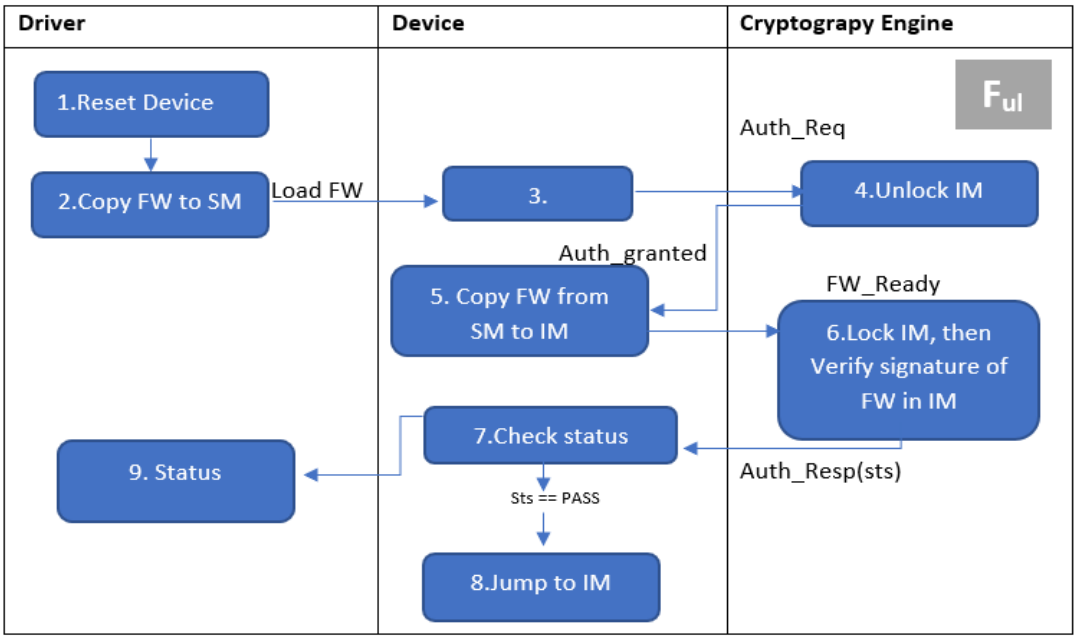

Fig.4. F_{ul} -Attack scenario sequence

5. Protocol – F_{aul} and F_{alu}

These build upon the previous protocols and combine them to provide more protection. We implemented these using the socket abstractions as mentioned above. However, we could not recreate the attack scenarios for these. It involved 3 threads to coordinate in an exact way for this attack to occur.

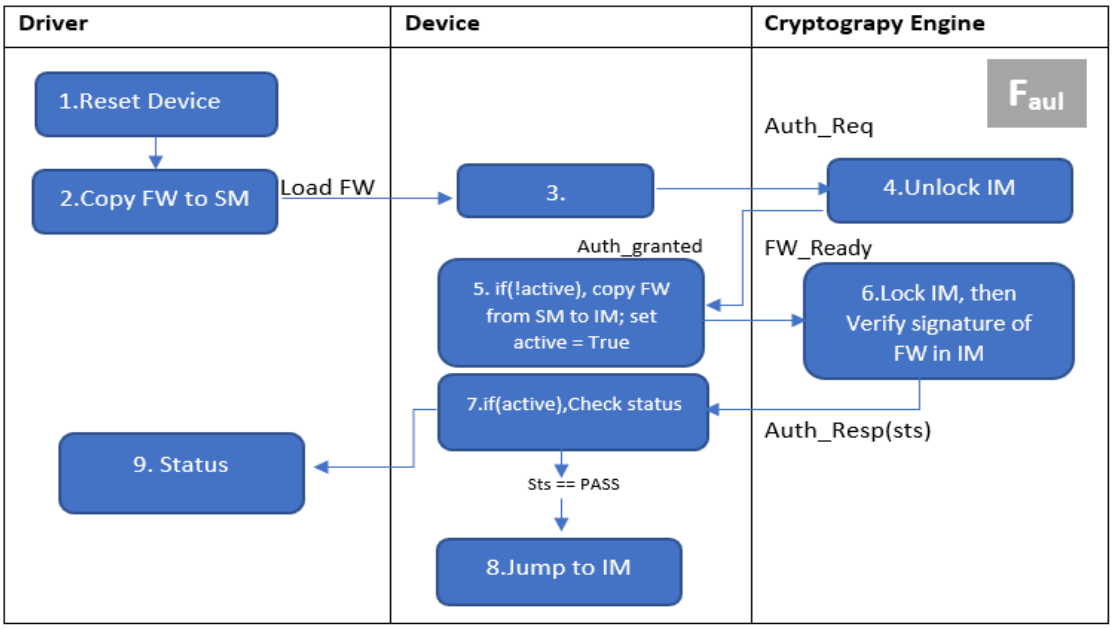

Fig.5. Faul – Attack scenario sequence

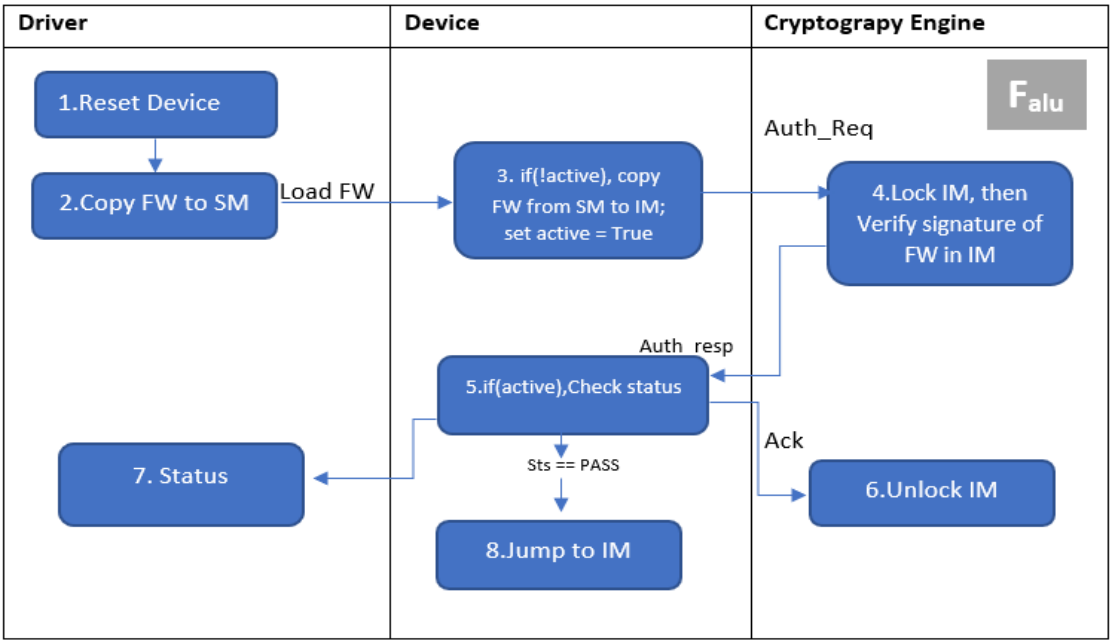

Fig.6. F_{alu} -Attack scenario sequence

IV. CONCLUSION

We tried to understand the TOCTOU attacks and the protocols that are governed to secure the firmware patching process. Starting on a basic protocol and improvising it to counter various ways of malicious attacks, we first tried to implement the protocol for single thread operation. With this condition, protocols seem to be working fine and the unintended firmware is blocked before getting activated. However, issue arises when multi-thread operations are accounted for, where all the six models seems to be prone to attack, especially when, considering the number of devices around the world that are connected through internet. Therefore, we need to dig in deeper to research about it and find out an efficient and the most secure way of patching the firmware over the air.

V. REFERENCES

[1] Sava Krsti´c, Jin Yang, David W. Palmer, Randy B. Osborne, Eran Talmor, "Security of Firmware Load protocols" 978-1-4799-4112-4/14/$31.00_c 2014 IEEE

[2] CollinMulliner and BenjaminMichele, "A mass-storage-based TOCTTOU attack"

[3] R.Elbaz, D.Champagne, C.H.Gebotys, R.B.Lee, N.R.Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. Transactions on Computational Science -Special Issue on Security in Computing, 4:1–22, 2009.

[4] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017

[5] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A Secure and Reliable Bootstrap Architecture. In Proceedings of the 1997 IEEE Symposium on Security and Privacy, SP '97, pages 65–, Washington, DC, USA, 1997.