Raspberry Pi 4 Malicious Bootloader

Andrew Rust
Columbia University
acr2213@columbia.edu

Emily Soto
Columbia University
es4060@columbia.edu

ABSTRACT

Hardware and firmware of computer systems pose critical yet often overlooked attack vectors when compared to mainstream concerns around software vulnerabilities. Systems exploited at the hardware level potentially allow for root-level access, remote code execution, and perhaps entire network access without privilege escalation required in software exploits. This paper details a procedure exploiting one such vector through a modified bootloader (U-Boot bootloader) on a Raspberry Pi 4 that perform a memory exfiltration attack. By manipulating the bootloaders hardware initialization and firmware code, we demonstrate the potential to covertly extract memory dumps over a network, revealing sensitive data before device initialization completes. This hypothetical attack outlines the security risk that naive bootloader configurations allow for, particularly among Internet of Things (IoT) devices, which typically lack robust security features like Secure Boot. The findings underscore the necessity for thoughtful security measures at the hardware level, such as secure boot mechanisms or hardened bootloader configurations, especially as increasingly ubiquitous "smart devices" are produced without security in mind.

ACM Reference Format:
Andrew Rust and Emily Soto. 2024. Raspberry Pi 4 Malicious Bootloader. In *Proceedings of* . ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

In the scene of cybersecurity, hardware or firmware attacks are generally more difficult to implement but are particularly dangerous because they can give attackers root access (level 0, kernel level) to devices usurping any software security in place. These attacks require knowledge of a device's hardware and booting software to properly interject any malicious code used as an exploit. System-on-chip (SoC) devices or Field-Programmable Gate Arrays (FPGAs) that use common bootloaders, like U-Boot or GRUB, are especially at risk because they need flexibility for their specific hardware, but often lack adequate scrutiny from a hardware security perspective that would be present in more prominent technologies like smartphones.

Embedded systems like these (SoCs and FPGAs), present an niche, but accessible attack surface given knowledge of their startup process. U-Boot, a widely utilized bootloader in these systems, is responsible for the initialization of hardware components and is the vector used in this paper alongside a nominally-used SoC: The Raspberry Pi 4 (RPi4). The U-Boot codebase, as well as documentation on the Raspberry Pi's hardware/firmware initialization is freely accessible and allows for our hypothetical attack to take place. The code used comprises of mostly assembly and C code as well as initialization scripts for a the RPi4. Given a niche product line using these technologies, and a lax security team, we demonstrate an attack where malicious code is hidden in seemingly benign, but obscure header and environment files, which allows for data exfiltration.

1.1 SCOPE & ARCHITECTURAL DETAILS

This research is conducted using a Raspberry Pi 4 with 8GB of memory, included with a modified U-Boot bootloader that exfiltrates memory on startup over the local network. The exploit take place temporally during the early stages of the boot process before the Linux kernel [LKM] takes over, we exfiltrate said data over the local network but also provide alternative means of exfiltration given different circumstances. The memory dumps themselves, expose potentially sensitive information, which we also analyse and categorize in this paper. Alongside benign application data(likely related to setup, telemetry, or background processes), sensitive files (.txt, .png) and encryption (ssh) keys are among what we show can be covertly accessed and transmitted in this attack. This attack mostly serves as a blueprint for other attacks following similar vectors of targeting hardware and firmware as opposed to exclusively software. We also provide directions for future research along these lines in an effort to advocate for robust security measures in and around the bootloader especially for devices less often under analysis.

2 RELATED WORK & OTHER ATTACKS

Embedded systems such at the Raspberry Pi often use specialized bootloaders due to their customized hardware configurations and limited resources. These bootloaders are designed to be lightweight and efficient, but they can sometimes lack the advanced security features found in more sophisticated computing environments. Consequently, vulnerabilities in these bootloaders can leave devices open to exploitation, potentially allowing attackers to control or disrupt critical functions.

One significant area of focus in embedded system security involves defending against cold boot attacks. These attacks exploit the tendency of RAM to retain data for a brief period after power has been removed. By quickly rebooting a system and dumping the contents of memory, an attacker can potentially recover sensitive information, including cryptographic keys and passwords, directly from RAM. Notably, in Won et. al[**1**], SDRAM information is recovered from the Raspberry Pi 4B+ with this particular methodology.

Working with the Raspberry Pi 4B+, we considered the possibility of an "evil maid" attack–a type of security threat where an attacker gains physical access to a target device, such as a laptop, smartphone, or hard drive. The attacker typically infiltrates the

device by either breaking into the target's room or gaining temporary access when the target is away. Once inside, the attacker can install malware or manipulate the device's hardware to gain unauthorized access to sensitive information. Ostensibly, given only a few minutes, with a new Raspberry Pi, one may reflash it to include a malicious bootloader. Compounded with this, there is certainly sensitive information to retrieve during boot–during the initial stages of the boot process, some decides load various encryption keys and credentials into the system's memory. These are essential for decrypting the file system and accessing encrypted data stored on the device. For systems with full disk encryption (FDE), master encryption keys are often retrieved from secure storage and loaded into memory only during the boot process to ensure they remain protected when the device is off. This perspective is explored further in Boursalian and Stamp [2], where they successfully pull off an attack using this methodology on macOS. Furthermore, there is a theoretical risk associated with the firmware utilized in systems like the Raspberry Pi 4B+, particularly considering the components sourced from manufacturers such as Broadcom. A prospective concern could arise if an individual working at Broadcom, or any other involved third-party, were to insert a covert backdoor within the firmware repository. Such an act could potentially allow unauthorized access to any device using this compromised firmware. A firmware backdoor embedded into the Raspberry Pi's GPU-related or boot process code—given these components' direct interaction with the system's core operations—could be especially hazardous. The potential actions from such exploitation might range from benign data gathering to disruptive system control, impacting users' privacy, security, and trust in the platform.

3 THREAT MODEL

Our threat model purposefully encompasses a wide array of potential adversaries, each with motivations that attempt to cover the attacks surfaces of embedded systems, and IoT-like devices, such as the Raspberry Pi used here. Notable actors in our model are state-sponsored entities, organized cybercriminal groups, opportunistic insiders, and security researchers each capable of the technical maneuvers we outlines here alongside the potential for more widespread and resource-intensive attacks - given enough importance on the target(s).

Firstly, state actors may target IoT devices or similar to establish covert surveillance channels or manipulate functionality within critical infrastructure, leveraging these devices as footholds in some broader cyber-espionage campaign or just as a simple surveillance proxy on a specific device. Organized criminals, presumably motivated by financial gain, might exploit these devices to access and exfiltrate corporate or government data, seeking to monetize stolen information through underground data markets, extortion, or ransom negotiations.

Related to organized groups, but perhaps with more political motives, insiders — either malicious employees or overreaching security researchers — could exploit physical or network access to experiment with embedded systems or devices on operating technology. Likely gathering data to be used for purposes of diagnostics in the best case to blackmail in the worst case. For curious tinkerers, perhaps vulnerabilities are inadvertently exposed only to allow later exploitations by other threat actors. We consider all potential threats as legitimate regardless of intent and seek to understand the technical capabilities that any motivation would allow for, given the assumed circumstances we outline that follow.

3.1 ASSUMPTIONS

The following assumptions underlie our threat model, framing the expected environment and conditions under which these threats may arise, which may be painfully broad:

- **Compromised Bootloader Installation:** We assume attackers have the means and opportunity to implant malicious firmware, either through direct physical access, supply chain compromises, or by leveraging previously established remote exploits.
- **Network Accessibility:** It is presumed that compromised devices maintain persistent network connectivity, which is necessary for data exfiltration. Also assumes minimal restrictions, simulating an environment where IoT devices are connected to broader networks without stringent segmentation or firewall policies, which is a common oversight in corporate, domestic, even governmental environments.
- **Security Configuration Oversight:** The model assumes that typical security configurations might not robustly partition sensitive network segments from general access, nor sufficiently secure hardware interfaces against unauthorized access. This reflects a plausibly realistic scenario where rapid deployment and scaling of IoT devices outpace the implementation of comprehensive security controls. This would manifest itself in malicious bootloader code that incorrectly passes internal audits or is never audited to begin with.

The above assumptions are purposefully generalized to encompass the broad range of potential attack surface that is commonly available to a successful attack at the bootloader level. Understandably, with robust security practices, many of these assumptions are nullified. Each of these too, apply specifically to our configuration with the Raspberry Pi, although our exfiltration was done in research - and not any real-world deployed setting. It is important to note too that in addition to these technical specifics on deployment of a malicious bootloader, it is likely to include an element of social engineering to allow for this code to go unnoticed or improperly trusted into production. We ignore this component of an attack but still wish to underscore its importance in the broader scene of a realistic attack.

4 BOOT PROCESS & METHODS

To outline our data exfiltration technique, we must first contextualize the scope, function, and sequence of the bootloader. For clarity's sake, the architecture will be simplified to only include sections notable to the exploit and not the entirety of U-Boot processes.

First, there are assembly scripts responsible for initializing generalized hardware, not specific to any board. This happens jointly across files inside `arch/arm/lib/` where modifications are made to `arch/arm/lib/crt0.S` such that the physical memory is modified and written to the end spaces in memory. On the RPi4 board, there is significant overlap with the video card and its memory as they share

After hardware setup is the memory virtualization, done after the environment is sufficiently prepared for C code to be executed. Here, modifications are made to the `arch/arm/mach-bcm38x/init.C` file where virtualization of the physical memory occurs. To do so however, it must be cleared which we wish to avoid in order to read the raw data on memory. So the designated blocks in memory are removed from the initialization process. Functionally this handicaps the amount of usable memory by the software during the boot process, which isn't problematic due to the low memory footprint of the bootloader anyway.

Lastly, to exfiltrate the data before the bootloader completes, we adjust the board-specific header file, located in `arch/arm/include/configs/rpi.h` and perform a memory dump after initializing a network connection with a local device. In this case, the local IP address is known, but in a practical instance there might need to be additional forensic measures taken place. We also outline several different alternative methods for exfiltration in the following section, including an ssh tunnel, simple https handoff or writing to a more permanent location in storage (SD card, SSD) to be accessed later.

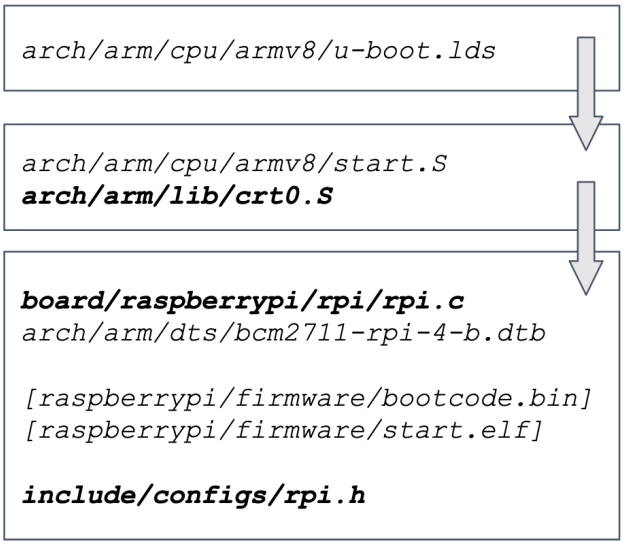

Figure 1: High-level booting process of U-Boot on Raspberry Pi 4, bolded files are modified in the malicious configuration

4.1 ALTERNATIVE EXFILTRATION METHODS

While the exfiltration method demonstrated here is transmitting the memory dump over the local network, alternative approaches can be employed depending on the attacker's resources, the target environment's security configuration, and any desire for stealth. The straightforward method used in this paper is the manipulation of environment variables within the bootloader to establish a connection to another device on the network and transmit using TFTP. This instead could redirect critical data to a benign-looking, but attacker-controlled, file in a specific folder. This file could be stored on more permanent media like an SD card or any attached storage (SSD/HDD).

Another technique would be to wait for the device to establish internet connectivity (via a successful DNS query or HTTPS connection), then pass the data over HTTPS to a attacker-controlled address. an HTTPS connection during its normal operation. At this point, exfiltrated data could be subtly intermixed with other outgoing traffic, depending on the level of suspected network monitoring done on the device. Augmenting this strategy, an attacker could use an SSH tunnel(s) to pipe out data to a remote server, especially if that is already the common means of transmission on the device, which is possible for IoT devices and the like.

5 DATA ANALYSIS

We found that the GPU-related code on the Raspberry Pi 4b+ was constructed with a proprietary assembly language Videocore IV– luckily, some work toward open-sourcing VC4 exists, and allowed us to analyze more thoroughly.

Continuing our exploration further into the Raspberry Pi 4B+'s proprietary systems, especially regarding the Videocore IV GPU-related code, it's evident that the developmental shift towards partial open-sourcing has been beneficial. This openness has provided us with clearer insights into the often opaque world of embedded GPU functionalities. The assembly language used, although proprietary, has started to reveal its structure thanks to the ongoing efforts towards open-sourcing. This allowed us a more in-depth glimpse into the crucial elements of GPU processing, particularly how encoding and decoding tasks are managed at a low level.

Within the files we analyzed, the transition of certain portions of the boot sequence storage to EEPROM in the Raspberry Pi 4 marks a design evolution from earlier models. This transition potentially enhances the robustness and flexibility of the firmware updates and management. Moreover, the ability to retain and execute bootcode independently of external storage media underscores a more resilient approach to maintaining critical operations, which are essential for reliability in varied application environments.

Using the binwalk -E utility, we created a histogram of the entropy in the two proprietary files of note, start4.elf and *bootcode.bin*, and we utilized a python script to output a histogram of the bytes of our dumped code. We did not find any encrypted sections of note. The absence of encrypted sections within these primary files—while facilitating easier exploration and academic critique—also points towards prospective security risks, specifically in environments where sensitive operations are conducted using the Raspberry Pi.

A particularly interesting string of bytes that appeared consistently across boots was loading "Micron PoP 44nm 4Gbit," which seems to confirm that our particular Raspberry Pi 4B+ was outfitted with this particular SDRAM. We were unable to find information readily available about the particular SDRAM model the Raspberry Pi 4B+ used anywhere else.

Most of our interesting extracted information, however, related to the setup of the graphics drivers. One of the essential aspects of the Raspberry Pi 4B+'s graphical enhancement is its encoding and decoding functionality. The GPU is equipped to handle H.265 (HEVC) video decoding natively, offering support for high-resolution video playback without burdening the CPU. Regarding encoding, the VideoCore VI also supports hardware-accelerated

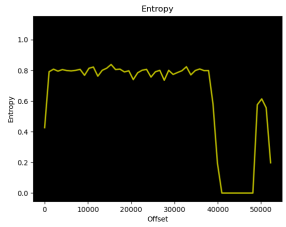

Figure 2: Entropy graph of bootcode.bin

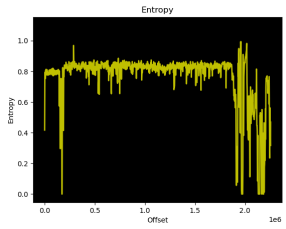

Figure 3: Entropy graph for start4.elf

video encoding (H.264). We were able to independently verify this functionality given the code. Interestingly enough, there seems to also be some power-related setup in the GPU setup, possibly an entry-way into a critical subsytem.

A more interesting section of note, from a security perspective, was .guard1 of start4.elf. It seemed that .guard1 contained, guard variables or canaries—special values placed in memory adjacent to buffers to detect and prevent buffer overflows. Ostensibly, these values are checked during boot to ensure they haven't been altered during execution; any change might indicate that a buffer overflow has occurred, which would seemingly prompt the system to halt operations/boot.

5.1 RESULTS & INTERPRETATIONS

Our investigation into the Raspberry Pi 4B+'s systems yielded substantial details that enhance our comprehension of its internal operations, particularly focusing on the proprietary files involved in its boot process and GPU-related functionalities.

From our entropy analysis using the binwalk -E utility, the lack of encryption in the files *start4.elf* and *bootcode.bin* suggests these components are more accessible for analysis than if they were encrypted. This potentially exposes them to higher security risks as it might allow malicious entities easier insights and manipulations, but it also simplifies the process for legitimate developmental modifications or academic scrutiny.

The examination also brought to light substantial insights into the loading mechanisms and behaviors during the boot process. For instance, examining the entropy and strings within these files helped identify regular patterns and key operations occurring at boot, such as initial hardware checks and driver setups.

Moreover, our analysis uncovered the robust nature of the Raspberry Pi 4's architecture concerning potential buffer overflow attacks. The presence of mechanisms like guard variables suggests a deliberate effort to incorporate security at the foundational level of firmware architecture. This is a crucial feature, particularly in a widely-used educational and developmental platform like the Raspberry Pi, which often handles diverse and potentially unsecured code.

Continuing our analysis, the findings from our examination have significant implications for both the maintenance and advancement of Raspberry Pi systems. The straightforward nature of the bootcode and its lack of robust encryption can provide an accessible platform for educators and developers to teach and experiment with low-level programming and system design. However, this openness also necessitates the implementation of enhanced security measures if the device is deployed in sensitive environments.

Further, the absence of sophisticated encryption in key operational files might imply lower processing overhead, leading to quicker boot times and less computational strain during startup. This characteristic is especially advantageous for educational purposes and experimental projects where simplicity and accessibility are priorities. Nevertheless, it also suggests the potential need for additional security measures when the Raspberry Pi is implemented in more sophisticated, security-sensitive projects, i.e, one which nation-state actors may take an interest in.

One possible mitigation for platforms such as the Raspberry Pi is to regularly allow security auditors to examine their proprietary code. They may identify and mitigate complex security issues that internal personnel might overlook, especially in intricate areas like boot processes and secure firmware loading.

Another effective mitigation strategy for combating bootloader backdoors is the implementation of **Hardware-based Security Modules (HSMs)**. HSMs store cryptographic keys securely and use these keys to decrypt and verify signatures on the bootloader and other critical startup code. If the validation fails—indicating tampering or unauthorized changes—the HSM can prevent the system from booting, thereby thwarting a potential attack exploiting the bootloader backdoor. HSMs provide a secure enclave for generating, storing, and managing cryptographic keys away from the vulnerabilities native to software-based key management solutions. This secure key management prevents attackers from extracting or manipulating keys to create forged signatures or bypass cryptographic verifications during the boot.

External auditors provide an independent verification of the security posture, ensuring that the system's defenses are robust and effective against actual threats. This independent assessment is crucial for trust and credibility, especially for devices deployed in sensitive or critical infrastructures where security cannot be compromised.

5.2 MAIN CONCLUSIONS

Bootloader attacks by way of core dumps are quite possible on SoCs where there is not robust booting procedures in place. It is not to say that open bootloaders like U-Boot are worse than secure boot, but simply that bootloaders are a viable entry point for attackers with any intent, potentially even manufacturers themselves with proprietary booting procedures. It is to show what is possible, not that any specific technology is any more or less vulnerable. At the

very least, U-boot's code is available and audit-able for security researchers to confirm or any suspicions that they may have.

In our case, the use of an open source bootloader made extracting the proprietary code far more efficient, but given that none of it was encrypted, grabbing the assembly was not overly difficult. On the contrary, the biggest hurdle to analyzing proprietary RPI code was Broadcom's VC4 assembly, which has not been entirely reverse engineered yet.

Open-source software comes with both costs and benefits. By gaining a deeper understanding of the firmware, open-source driver developers may work to enhance the performance and capabilities of the Raspberry Pi 4B+. Enhanced knowledge of hardware-specific functions allows for more efficient translations between the software applications and the physical components of the device, significantly reducing compatibility issues that can arise from miscommunication between hardware instructions and custom driver operations.

However, given their early and integral stage in the boot process, malicious versions of start4.elf and bootcode.bin hold the potential to be disastrous. With detailed knowledge of firmware operations, particularly if undocumented features or vulnerabilities are uncovered, malicious actors can craft targeted attacks that exploit specific weaknesses in the system. For example, they could exploit loopholes in the boot process or memory management functionalities identified through the analysis. These could be used to insert malicious code, create backdoors, or initiate privilege escalation attacks, where the attacker gains higher access privileges than originally intended. Given the widespread use of Raspberry Pi devices in various applications, including sensitive and critical environments, such exploitation could have far-reaching consequences.

5.3 FUTURE RESEARCH

To conclude our work, we would like to present related areas upon which further study is possible and invited. Firstly related to data analysis, a plethora of reverse engineering and binary analysis tools currently exist but perhaps additional focus on tools or methodologies for low-level initialization code or proprietary binaries for custom SoCs is of evergreen interest. Related to the attack outlined here, an improved infiltration technique or obfuscation strategy of the malicious bootloader may provide a more holistic red-team attack for device manufacturers looking to secure systems from cold boot attacks, malicious bootloaders, or any hardware/firmware-targeted vectors. Our approach was reductive, making assumptions about the security processes in place, so perhaps analysis on a system deployed in a more realistic, yet still a research environment would net a more substantive attack. In the wider context of secure booting, penetration-testing hardware-based security measures on more mainstream boards like trusted platform module (TPM) on Intel chips and platform security processors (PSP) on AMD chips would be substantive spaces for future research to be done. More specific to IoT devices would be security analysis on other bootloaders or hardware initialization code. A very lightweight bootloader for low-spec IoT devices might be a more secure solution than a bootloader for a whole operating system like U-Boot or GRUB. Lastly, smaller batch CPUs or smart-device specific processors could be potential exfiltration targets too instead of just physical memory as was the focus here. An addendum to our efforts on memory exfiltration could be CPU registers, caches, or colder storage locations like SD cards.

5.4 DEFENSIVE TECHNIQUES

To mitigate any vulnerabilities exposed by this paper and related bootloader-based attack vectors, several defensive measures could be implemented. Firstly, additional layers of encryption at the bootloader level can significantly enhance security by protecting sensitive information from being easily accessible during the boot process. Employing a robust, open-source secure boot protocol can also prevent unauthorized modifications to the bootloader, perhaps ensuring that only a signed and/or otherwise verified bootloader and accompanying binaries is executed. Another effective strategy could involve the use of a separate security chip that also manages memory, such as what is done with a TPM, which can store crypto keys away from any hardware components.

5.5 INSTRUCTIONS ON REPLICATION

For teams interested in replicating and extending research into bootloaders, the following steps outline the general process from initiation to completion, based on our experiences:

(1) Setup:
(2) Env modification
(3) code modification
(4) security analysis
(5)

5.6 ADDITIONAL LEARNINGS

This project has provided significant insights into the security landscape of embedded systems, particularly those that use open-source bootloaders like U-Boot. One of the key takeaways is the critical importance of security in the early stages of the device boot process. It highlighted the potential consequences of neglecting this aspect, demonstrating how vulnerabilities can be exploited to gain unauthorized access.

Furthermore, the project emphasized the value of open-source tools and community collaboration in identifying and mitigating security vulnerabilities. More concretely, over the course of the project we learned how to reliably and conveniently dump data from embedded devices as well as analyze binary files in Ghidra. Engaging with the open-source community can accelerate vulnerability discovery and patching, enhancing the security of widely used platforms. This project underscored the need for a balance between security and functionality in embedded systems. While it is vital to secure devices against potential threats, this should not come at the expense of their performance or usability. Finding this balance requires ongoing research and development, and a deep understanding of both hardware capabilities and security threats.

Were we to redo this project from scratch, we'd allocate much more time to dumping data from the bootloader–even if it was not the particular data we were looking for. Ultimately, getting data dumped consistently was one of the more considerably challenging parts of our setup, and we likely should have planned out exactly how were going to do it in advance.

6 REFERENCES

(1) T. Cannon and S. Bradford, "Into the Droid: Gaining Access to Android User Data," in *DefCon '12*, VIA Forensics, July 2012. [Online]. Available: https://www.usenix.org/system/files/conference/woot17/woot17-paper-hay.pdf

(2) S. Wright, "The Symantec Smartphone Honey Stick Project," Symantec Corporation, March 2012.

(3) Ponemon Institute LLC, "The Lost Smartphone Problem: Benchmark study of U.S. organizations," in *Ponemon Institute Research Report*, Sponsored by McAfee, October 2011.

(4) J. Rutkowska, "Evil Maid goes after TrueCrypt," *The Invisible Things Lab*, October 2009. [Online]. Available: http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html

(5) C. Fruhwirth, "LUKS On-Disk Format Specification Version 1.1.1," Linux Unified Key Setup, December 2008.

(6) J. Carbone, D. Bean, and J. Salois, "An in-depth analysis of the cold boot attack," *Technical Memorandum No. 2010-296*, DRDC Valcartier, Defence Research and Development, Canada, January 2011.

(7) T. Müller, M. Spreitzenbarth, and F. C. Freiling, "FROST: Frost Forensic Recovery of Scrambled Telephones," Friedrich-Alexander University of Erlangen-Nuremberg, October 2012. [Online]. Available: https://faui1-files.cs.fau.de/filepool/projects/frost/frost.pdf

(8) xda-developers, "Google Play Nexus not wiping after Bootloader Unlock," Thread No. 1650830, April 2012. [Online]. Available: http://forum.xda-developers.com

(9) "U-Boot Documentation for Raspberry Pi 4," *Documentation*, [Online]. Available: https://docs.u-boot.org/en/latest/board/broadcom/raspberrypi.html

(10) "Boot Sequence for ARM Chips," *ARMv8r64 Refstack Documentation*, [Online]. Available: https://armv8r64-refstack.docs.arm.com/en/v5.0/manual/components.html#boot-sequence

(11) "BCM2711 Datasheet," [Online]. Available: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf

(12) "Blogger Tinkering with U-Boot," Synacktiv, [Online]. Available: https://www.synacktiv.com/publications/i-hack-u-boot

(13) "CPU Memory Locations," in *Challenges in Computational Statistics and Data Mining*, Springer, 2020. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-29962-0_20

(14) "Technical Reference Manual (TRM) for A72 ARM Chip," [Online]. Available: https://documentation-service.arm.com/static/5e7b6c287158f500bd5bfb61

(15) "Related Paper on Warm Boot Attacks," *IEEE Xplore*, [Online]. Available: https://ieeexplore.ieee.org/document/10136888/

(16) "U-Boot on Raspberry Pi," eLinux.org, [Online]. Available: https://elinux.org/RPi_U-Boot

(17) "Rebooting RPi Running Linux," Sigmdel, [Online]. Available: https://sigmdel.ca/michel/ha/rpi/hard_soft_rpi_reboot_en.html#soft_restart

(18) "U-Boot on RPi Blogger," by H. Chao, December 2021. [Online]. Available: https://hechao.li/2021/12/20/Boot-Raspberry-Pi-4-Using-ub

(19) "ARMv8 Manual," [Online]. Available: https://armv8r64-refstack.docs.arm.com/en/v5.0/manual/boot.html