Breaking all the Things — A Systematic Survey of Firmware Extraction Techniques for IoT Devices

Sebastian Vasile, David Oswald, and Tom Chothia

University of Birmingham
sxv512@student.bham.ac.uk, d.f.oswald@bham.ac.uk, t.p.chothia@bham.ac.uk

Abstract. In this paper, we systematically review and categorize different hardware-based firmware extraction techniques, using 24 examples of real, wide-spread products, e.g. smart voice assistants (in particular Amazon Echo devices), alarm and access control systems, as well as home automation devices. We show that in over 45% of the cases, an exposed UART interface is sufficient to obtain a firmware dump, while in other cases, more complicated, yet still low-cost methods (e.g. JTAG or eMMC readout) are needed. In this regard, we perform an in-depth investigation of the security concept of the Amazon Echo Plus, which contains significant protection methods against hardware-level attacks. Based on the results of our study, we give recommendations for countermeasures to mitigate the respective methods.

1 Introduction

Extracting the firmware from IoT devices is a crucial first step when analysing the security of such systems. From a designer's point of view, preventing the firmware from falling into the hands of an adversary is often desirable: for instance, to protect cryptographic keys that identify a device and to impede product counterfeit or IP theft. The large variety of IoT devices results in different approaches to firmware extraction, depending on the device in question.

Past work has looked at the state of security of IoT devices, e.g. [1,2,3]. Past work on the analysis of IoT firmware has found a wide range of vulnerabilities [4,5,6], and such vulnerabilities have been widely exploited [7]. Much of this work looks at firmware downloaded from the Internet, rather than taken from a device.

In contrast, not so much attention has been given to the hardware security of these devices. Having access to an embedded device's firmware can provide valuable insight into how the device operates and potential vulnerabilities it might have. Sensitive information, such as passwords and static keys can often be found in a firmware, which is indicative of insecure design and bad overall security. Besides, vectors used for firmware extraction also give write access to the device, enabling firmware *modification* as well.

Firmware extraction is not an exact science. The market is filled with a variety of IoT devices, each using one of the many embedded processors, with

their own settings and software stacks, making each device unique in its own way. Because of this, there is no one-glove-fits-all scenario when it comes to firmware extraction of IoT devices. At DEFCON 25, techniques to extract firmware from a range of IoT devices were presented [8], while Etemadieh et al. focused on the use of the eMMC interface (cf. Section 3.2). This work culminated into the `Exploitee.rs` project [9]. We include certain devices from [9] as part of our survey (cf. Section 4), but would like to note that we are not affiliated with that project.

The topic has only received relatively limited academic attention, with a first step towards a more systematic approach given by Schwartz et al. [10]. However, the authors of [10] focus on a relatively narrow class of low-cost devices (IP/baby cameras and doorbells). In this paper, we consider a significantly wider range of device types as well as extraction methods. Our case studies include popular smart voice assistants like the Amazon Echo product range and other extremely wide-spread IoT devices. Out of the devices included in the survey, our research suggests that UART is currently the most common and exploitable debugging interface found in IoT devices, with over 45% of the considered devices being vulnerable to firmware extraction via UART. However, direct access to flash memories (e.g. eMMC) is also becoming important for modern devices. Notably, in almost all cases where a hardware method is available for firmware extraction, the method also enables firmware modification and hence "rooting" of the device.

Contrary to the common opinion that security in the IoT is a lost cause, we also observed positive developments, with newer high-profile devices like the Amazon Echo offering a better level of protection compared to most other vendors.

The remainder of this paper is structured as follows: in Section 2, we present background information about the technologies and debugging interfaces in use in embedded systems. In Section 3, we present methodologies for firmware extraction using different techniques and interfaces. Then, in Section 4 we present case studies for the described methods. Particularly, in Section 4.4, we describe various measures implemented in new Amazon Echo devices, that—while they do not prevent firmware extraction—significantly raise the bar for malicious firmware modifications. Based on the case study, we recommend countermeasures for securing devices against firmware extraction and modification in Section 5, before concluding in Section 6.

2 Technical Background

Unlike the PC market, embedded systems are very diverse, each suited for particular applications of these devices. Such devices usually employ microcontrollers, that consist of one or more CPUs, along with their own memory and I/O peripherals. Common microcontroller architectures for IoT devices include ARM, MIPS, Freescale and Texas Instruments TI MSP.

Firmware In the context of embedded IoT devices, the firmware usually refers to the entire OS image that incorporates the kernel and file system, together

with different binaries and scripts running together, making up the device's functionality. On very low-end devices, such as TV remotes for example, the firmware can be a single binary handling the entire functionality of the device, from booting to transmitting RF signals. A full firmware image usually consists of bootloader(s) (2nd/3rd stage, often U-Boot), the kernel, and one or multiple file system images (e.g. SquashFS, JFFS2, UbiFS, etc.) containing custom binaries and scripts. While there are many custom operating systems made for specific devices, by far the most common is embedded Linux. There are many different distributions of embedded Linux currently used in embedded devices, but they are all very similar in functionality. Another notable commonly found OS in embedded devices is Android (cf. Section 4.4). There are also new operating systems specifically made to address the security and performance requirements of the IoT devices of today. These include Windows IoT Core, Kaspersky OS and Google Brillo OS (Android Things).

Bootloaders The bootloader is the first piece of software that runs on a system. The bootloader initializes hardware components such as RAM, flash storage, and I/O, and loads the kernel into memory for execution. In embedded systems, the boot process can be set up in one, two, or three stages, each stage having a different role during boot. In a three-stage process, the initial bootloader, which is usually located in ROM and is microcontroller-specific, handles the basic initialization of hardware components, and loads the second stage bootloader. The second stage bootloader, which typically resides on flash storage and is board-specific, handles the initialization of board-specific hardware. After the initialization, it loads the final bootloader, which copies the kernel into main memory, loads device drivers for the found hardware components, and runs the kernel code. One of the most widely used bootloader for embedded systems is U-Boot [11], which is used as a second stage bootloader. Aside from the booting process, U-Boot also has a command line interface.

Debug Interfaces Most microcontrollers offer on-chip debugging functions, usually used for IC fault-testing, direct memory access, and for programming integrated flash chips. Common interfaces include UART, JTAG, Serial Wire Debug (SWD) for ARM processors, as well as Background Debug Mode (BDM) in automotive processors. Other serial interfaces include SPI and I2C.

3 Firmware Extraction Techniques

Firmware extraction presents a couple of challenges for IoT device manufacturers. First, there is a risk of potential IP loss. More importantly however, firmware extraction can often lead to the discovery of new vulnerabilities in such devices. In some cases, this can have an effect not only on the analyzed device, but on all devices belonging to the manufacturer, due to critical vulnerabilities being discovered. We classify firmware extraction methods into three main categories:

- Leveraging debug interfaces to get local shell access or read memory contents;

– Performing a flash chip hardware memory dump;
– Using software methods to gain access to firmware (e.g. firmware updates, network services, etc.).

The execution of these methods usually varies from one device to another, adapting to the particularities of each device. Apart from the aforementioned methods, there is also a hardware method called bus snooping. This method inspects in-transit data between caches and controllers on a bus. A well-known example of this is the original XBox hack [12], which used hypertransport bus snooping to extract the firmware decryption key of the XBox. Aside from the hardware extraction methods, there are also software methods that can be leveraged for firmware extraction. One example are code execution vulnerabilities, which can be exploited to get shell access on the device. As most software runs as root on embedded devices, a successful exploit results in a root shell. In most cases where firmware extraction is possible, firmware modification can also be achieved using the same methodology [13]. In some cases, this can be even more harmful than firmware extraction [14]. Even if no vulnerabilities are found, an attacker might still implant a backdoor on a device such as the Amazon Echo and sell it online. An unsuspecting buyer would get a backdoored device, capable of spying via the microphone, or using the linked Amazon account to make fraudulent purchases. Besides, firmware modification is useful when dynamically analysing the firmware's behaviour, for instance by enabling live debug capabilities (e.g. through a disabled UART or `adb` interface).

3.1 Debug Interfaces

UART Firmware Extraction UART is often a straightforward way (see also [10]) of gaining access to an embedded device's firmware. An unrestricted root shell can often be found by simply connecting to UART. On Android-based devices, a root shell is sometimes accessible via the Android debug interface `adb`. Another method is to utilize the shell of a bootloader to enable root access or obtain a firmware image in cases where a root shell is not present during operation or is password-protected.

With root access, one way to dump the firmware is to perform a live internal dump of the entire filesystem, with all files bundled together in a tar or zip archive, or to dump the block devices available on the device using `dd` or `cat`. However, dumping block devices can cause problems since embedded systems use different types of flash storage with different filesystems. In general, we recommend to follow the following steps when performing UART firmware extraction:

1. Identify the UART interface through visual inspection, oscilloscope probing, and trial-and-error;
2. If an unprotected shell is available, image the device or download all files. Files can be downloaded using `netcat` (or similar tools) and a PC connected on the same network;

3. If the shell is password-protected, try common username/password pairs from a list (e.g. `root/root` etc.). If no shell is available or the credentials cannot be determined, attempt to interrupt the boot process and enter bootloader shell;
4. If the bootloader shell cannot be entered, try to temporarily disturb the flash interface (e.g. by grounding a data or clock pin) when the bootloader loads the kernel in order to fallback to the bootloader's shell. Image the device from the bootloader shell (e.g. using `nand dump` or `nand read` and `md` under U-Boot).

JTAG Firmware Extraction The JTAG port used during manufacturing for loading firmware can in some cases be used for reading the full memory of the chip. Reading the memory of a device via a JTAG port requires a suitable programmer that can receive the memory dump and transmit it to a computer. Some manufacturers lock the device from being read or reprogrammed after manufacturing. Leaving the JTAG interface connected and unlocked exposes the device to firmware extraction and firmware injection attacks. The general process of JTAG firmware extraction is:

1. Visually identify possible JTAG/SWD (and other) debug interfaces. SWD requires only two pins, while JTAG has a variety of different pin arrangements, ranging from 8 pins to 20. As a general rule, two rows of four or more pins are likely candidates for JTAG;
2. As with UART, first identify the ground pin using a multimeter;
3. To identify the pinout, the data sheet for the particular microcontroller is needed. If the data sheet is not available, use a tool like the JTAGulator [15] to identify possible pinouts;
4. After identifying all pins, a suitable JTAG/SWD programmer can be used to dump the internal memory if no readout protection is enabled.

Due to the large variety of different pinouts and proprietary pins, as well as different JTAG debuggers for different microprocessors and architecture types, firmware extraction via JTAG requires more effort than UART, as specialized hardware and software and information gathering are required.

3.2 Raw Flash Dump

The third and final hardware-based firmware extraction method considered in this paper is directly reading the flash storage. Reading older flash chips with parallel interfaces requires many connections to the target device, as well as a specialized programmer. Newer technologies such as eMMC however, require less connections and can be read with a standard SD card reader. Alternatively, specialised tools like easyJTAG Plus[1] or RiffBox[2] can be used. A deeper description of eMMC extraction can be e.g. found in [16]. The general steps for performing flash dumps are:

[1] `http://easy-jtag.com/`
[2] `http://www.riffbox.org/`

1. Identify the flash chip (by label, package type, number of connections to processor) and obtain a data sheet if possible;
2. Identify the pins, either by data sheet or oscilloscope. eMMC uses DAT0, CMD and CLK pins, as well as power and ground. CLK is a repetitive signal, while the CMD line has short data bursts, generally preceding data reads/writes on the DAT0 pin.
3. For eMMC: Disable access to eMMC from the processor, and connect pins to a generic SD card and use an SD reader to interface with it;
4. For other flash chips: Use a suitable adapter and programmer to read the chip contents, e.g. the MiniPro TL866[3];
5. If in-circuit dump is not possible, de-solder the flash chip and perform the dump with a suitable reader.

For in-circuit dumps, it is required to prevent accesses from the board's CPU while reading the memory. This can e.g. be achieved by temporarily cutting the clock line and re-connecting after the dump is completed. Sometimes, simply connecting an eMMC interface (e.g. easyJTAG Plus) prevents the CPU from booting, cf. Section 4.4. Alternatively, one can attempt to keep the processor in reset through the respective pin.

3.3 Software Methods

Software methods are a form of firmware extraction that does not require physical access to the device in some cases. Examples include:

1. Check the device manufacturer website for publicly available firmware;
2. Follow direct download links to firmware updates, analyzing the device's network traffic;
3. Intercept network traffic for firmware updates. If TLS is used, attempt to perform a man-in-the-middle attack using self-signed certificates to decrypt the traffic;
4. Identifying and using running services on the device, and exploiting known vulnerabilities in such software (e.g. default credentials).

Often, firmware update services provide packages containing only modified files. Therefore, this results in an incomplete image. There are however cases where firmware updates consist of full firmware images, making this method a simple and effective firmware extraction solution. In addition, it should be noted that in many cases, firmware images are packed or encrypted, sometimes in proprietary formats. Unpacking or decrypting such images is a challenge on its own.

4 Case Studies

Table 1 summarizes the results of the case studies presented in this paper together with other devices we analyzed (that are not described here for the sake

[3] `http://minipro.txt.si/`

Table 1. Survey of firmware extraction and modification techniques

Device	Reference	Debug Interfaces			HW Dump	SW Methods	Root achieved
		UART	JTAG	Other			
Accu-Chek Insulin Pump	[17], this paper		✓				?
Amazon Echo	[18,19], this paper			SD+UART	✓	✓	✓
Amazon Echo Dot	[19,20], this paper				✓	✓	?
Amazon Echo Plus	this paper						?
DroiBOX MXG	this paper	✓					✓
Hive Nano V2 Hub	this paper				✓	✓	✓
Infotainment ECU	this paper	✓					✓
KONX Video Doorbell	this paper	✓					✓
Phillips Hue Lights	[21], this paper	✓					✓
Samsung SHS-5230 Lock	this paper		✓				?
Smart-I Doorbell	this paper	✓					✓
Smart Rear View Mirror	this paper			adb			✓
Swann OneTouch Hub	this paper	✓					✓
WD My Cloud NAS	this paper					✓	
Yale Alarm	this paper		✓				?
Amazon Fire TV Stick	[9]				✓		✓
Amazon Fire TV	[9]				✓		✓
Amazon Tap	[9]	✓					✓
Asus OnHub	[9]	✓					✓
Google Nest	[9]					✓	✓
Google Chromecast	[9]	✓					✓
Google OnHub	[9]			USB			✓
LG Smart Refrigerator	[9]	✓			✓		✓
Samsung Allshare Cast	[9]	✓					✓
Total #	24	11	3	3	7	5	18
Total %		45.83%	12.50%	12.50%	29.16%	20.83%	75%

of space) and devices from other sources as indicated. We also indicated whether obtaining root access via a hardware method is possible, or if this has not been tested but should be possible (marked as "?"). These devices were not tested due to various reasons, as some devices are running monolithic firmware without an OS, or running Windows CE. The Amazon Echo Dot is very similar to the Amazon Echo Plus, so results from the Plus should transfer to the Dot. For the devices from [9], we list a selection of popular devices where i) a hardware method can be used to extract the firmware and ii) where it is clear that a firmware binary was actually obtained. We chose devices from the following major manufacturers: Google, Samsung, LG, Asus, and Amazon.

4.1 Custom Debug Interface: Amazon Echo

On the bottom of the device, a group of test points is exposed for debugging purposes. The pinout of the debug port has been documented in [18].

The Echo pinout shows the device has a UART interface and an MMC interface, which allows an external SD card to be connected. Connecting to the UART interface and booting up the device, we observe that the Echo uses a three-stage booting process. In the first stage, X-Loader tries to locate U-Boot in the boot partition of the internal memory card. Once U-Boot is loaded, it starts the final bootloader, found in the storage partition under the `/boot` directory. In the case of the Echo, the booting process cannot be stopped by sending UART characters, and any input after booting has finished is ignored. Since firmware extraction purely via UART is not possible, the debug interface is marked as "other" (combination of SD and UART) in Table 1.

The initial bootloader tries to boot from the external `mmc-0` (which fails) and then boots from `mmc-1`, the internal memory. As detailed in [18] and also explained in [22], it is possible to build a bootable SD card that can be connected to `mmc-0`. Then, the device can be configured to boot into a root shell, and imaged with `cat /dev/mmcblk1 > image.img`. Alternatively, we found that it is also possible to create an SD card that only contains U-Boot (but does not attempt to boot the kernel). This was also reported by independent research in [23]. From this card, we can drop into a U-Boot shell, from where the device can be imaged or configured to enable root access (by injecting an SSH service and running it on boot).

4.2 UART: Smart-I Doorbell

The Smart-I WiFi Doorbell is a WiFi-enabled unit that is installed outside the front door of a house. It has a camera, which is activated when a visitor presses the button, and can also be equipped with an optional door release. Using an Android/iOS app, the user can see and speak to the visitor and open the door remotely.

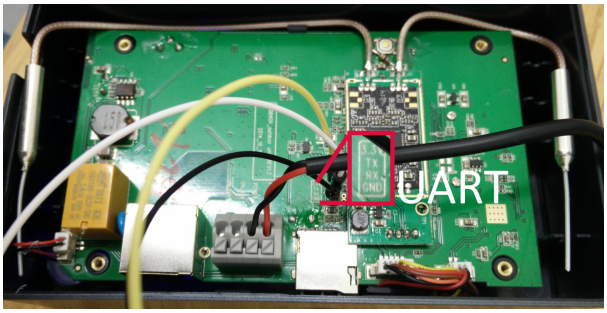

Fig. 1. Smart-I PCB with UART interface attached

Opening up the device, a UART port can be easily found, to which we can connect. Using the UART interface of the device (baud rate of 38400), we iden-

tified the presence of U-Boot as bootloader, and found an enabled root shell as well.

Furthermore, U-Boot has the bootdelay left at the default value of 3, which allows us to interrupt the booting process and drop to the U-Boot shell. The firmware can be extracted via the live filesystem using the root shell or via the U-Boot shell. For this device, the latter approach was used. The flash dump can be obtained with the commands in Listing 1.1:

Listing 1.1. Dumping Smart-I SPI flash from U-Boot

```
=> sf probe 0:0; sf read 0x8000000 0x0 0x800000; md.b    0x8000000 0x800000
#SF: Got idcode ef 40 17
8192 KiB W25Q64CV at 0:0 is now current device
##########################
08000000: 47 4d 38 31 32 36 00 00    GM8126..
          00 60 00 00 00 60 00 00    .‘...‘..
08000010: 00 60 0a 00 00 00 0e 00    .‘......
          00 00 00 00 00 00 00 00    ........
```

In order to save and process the dump, the serial output needs to be saved to a file. A simple Python script can then be used to convert the dump into a binary file. Further analysis with Binwalk [24] revealed that the image is LZMA-compressed. Decompression using Easylzma [25] results in a readable firmware image.

4.3 JTAG: Yale Easy Fit Smartphone Alarm

The Yale Easy Fit Smartphone Alarm is a wireless home alarm system that can be fully controlled from a mobile app. The alarm kit consists of motion sensors, wireless cameras, a keypad, a wireless remote, a central unit and a siren. We focused on the central unit as depicted in Fig. 2.

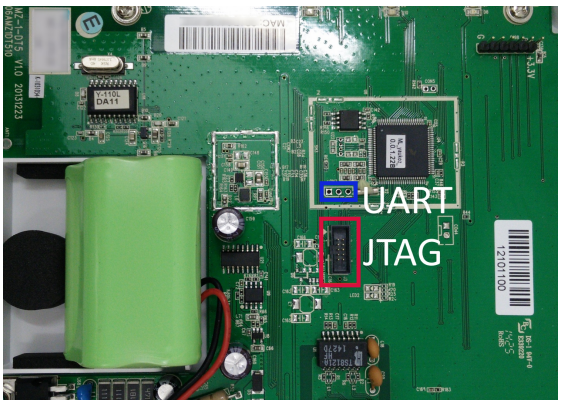

Fig. 2. Yale Easy Fit central unit PCB with UART and JTAG

Inspecting the disassembled device, we observe that the board employs an ARM Freescale MK60 CPU. The board has both UART and JTAG interfaces,

highlighted in blue and red, respectively, in Fig. 2. A UART interface was enabled on the device (baud rate 115200) but did not respond to user input. Its only purpose appears to be to output proprietary debug data. As user input was disabled, the bootloader could also not be bypassed. A standard JTAG header was found on the board, so further pinout reverse engineering was not needed. As the JTAG interface was not locked or disabled, the J-Link [26] programmer could be used to extract the complete firmware image using the J-Link proprietary software.

4.4 eMMC: Amazon Echo Plus

The new Echo Plus is similar to the second generation Echo Dot, as it runs an Android system, compared to the previous Amazon Echo, which runs a custom embedded Linux system (Section 4.1). We selected the Echo Plus as the most extensive case study in this paper, as it (in contrast to most other devices) employs a variety of rather effective security measures and is an example for best practices in the IoT. As with the previous Echo, debug pins are available on the bottom of the device. A UART interface is available, but only used for diagnostic output during the boot process, instead of giving shell access on the device. There are two different levels of debugging output, a lower level for the first stage ROM bootloader with a baud rate of 115200, as well as output for the following bootloaders with a baud rate of 912000.

Booting from a custom SD card image is no longer possible for the Echo Plus. Besides, we also could not find an `adb` shell on the device, consistent with the findings of [20]. However, in [19], the eMMC interface of the Echo Dot is documented. With minor modifications, we could connect to this interface using the easyJTAG Plus programmer. The pinout and the necessary connections are shown in Figure 3. It is noteworthy that the easyJTAG, when connected, prevents the Echo Plus from booting.

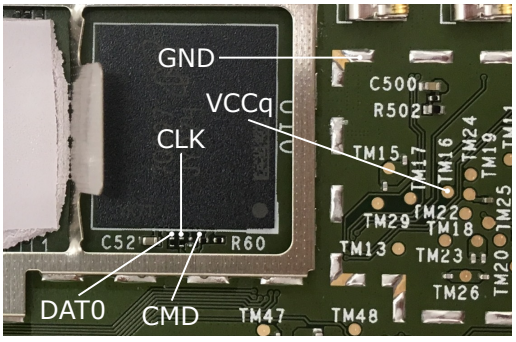

Fig. 3. eMMC pinout of Echo Plus. VCC is not connected to the easyJTAG; the eMMC chip is powered via the normal power supply of the board.

Therefore, we used an interface board (plugged into the easyJTAG), to which thin wires to the eMMC pads are soldered. The interface board can stay permanently connected; to boot the Echo Plus, the board is simply unplugged from the easyJTAG—yielding the ability to repeatedly read and write the firmware without (re-)soldering wires. Extracting the firmware via eMMC results in 16 separate partitions (following the Android standard layout). The Echo Plus has two separate `_a` and `_b` partitions each for the Little Kernel (LK), the actual kernel, and the system partition. The reason for having redundant partitions is the software update: the device e.g. boots from the `*_a` partitions, but then updates the `*_b` ones and switches to those when fully updated.

Secure Boot The boot process on the Echo Plus consists of three stages. The Boot ROM (BROM) embedded in the MT8163 processor boots the first stage bootloader. The preloader is present in the *bootloader_rom2.bin* partition. In the second stage, the preloader loads the LK, which resides in the *lk_a* and *lk_b* partitions. LK is the standard Android bootloader, which loads the kernel (version 3.18.19+) in the final stage. The kernel, which is present on the *boot_a* and *boot_b* partitions, loads the file system and initializes all system services. The kernel image holds all Android startup scripts, SELinux domain definitions, service definitions, kernel boot parameters and all other system configuration files. As evident from the boot log , the Echo Plus employs a secure boot process, where each bootloader verifies the subsequent stage. Furthermore, SELinux is enabled in *enforcing* mode [27]. To enable dynamic analysis of the running device (as achieved for the previous generation, see Section 4.1), we attempted to obtain shell access with full root privileges.

Our first attempt was to modify the kernel image to implant our own startup service with full root access. However, the Echo Plus employs a trusted boot chain, where each bootloader verifies a signature of the next boot stage. This means that we were unable to boot the device when changing the kernel image, LK, or preloader on the eMMC. The first stage (BROM) is stored in ROM and hence unchangeable.

Modifying the System Partition In contrast to the boot process, the system partition is not signed, and we did not find evidence for the use of cryptographic verification methods like `dm-verity`[4]. Having write access to the system partition, we first attempted to start a reverse shell from one of the several scripts (in the system partition) that get executed at boot time. For this, we added an ARM `netcat` binary in the `/system/bin` partition, as well as adding debug commands in each `.sh` file found, outputting different files to the `/cache` partition so we could identify which script files are executed on boot. After identifying the startup scripts, we found that SELinux was preventing us from running the `netcat` binary. The reason for this is that shell scripts on boot run in the restricted *init_shell* SELinux context. We are currently exploring further methods to run a binary with full admin privileges (SELinux context `su_exec`). Work on

[4] `https://source.android.com/security/verifiedboot/dm-verity`

this subject has recently been published at DEFCON 26 [28], where researchers have been able to successfully root the Echo Plus. We will examine this research in order to obtain root access on the Echo Plus.

5 Countermeasures

Based on the findings from our survey, we propose a set of countermeasures against firmware extraction (and sometimes modification). These measures increase the cost to an adversary per device analyzed to prevent wide-reaching, low-cost attacks. Although increasing the cost to an adversary might stop low-level attackers, the model of "security by obscurity" is never an adequate defense strategy against well-resourced attackers. As pointed out by Dullien [29], removing "inspectability" usually does not deter malicious adversaries, while creating obstacles for benign security researches and defenders. On the other hand, when physical access to a device is part of the threat model, leaving debug interfaces open may allow straightforward extraction of secrets and user data as well as malicious modifications. It is an open problem to balance these two aspects. A potential solution might be to provide device-specific debugging credentials to the device owner, or to implement an *auditable* mechanism (e.g. using write-once fuse bits) to put the device into a debugging mode.

UART, Bootloader, and Software Methods Debug interfaces can be disabled or protected post-production. For UART, the bootloader and kernel can be configured to disable the console to prevent access (as e.g. implemented in the Echo Plus). If a UART shell or remote SSH/Telnet access is required post-production, it should be password-protected, with a password unique for each device. This password could be made available in a secure way to the device owner to provide inspectability. All network communication with back-end services should be encrypted using TLS or a similar protocol, especially for firmware updates.

JTAG and other Debug Interfaces On most microcontrollers, JTAG (and other interfaces) can be either permanently disabled or protected with a password (if JTAG is not to be fully disabled for debugging or fault analysis). While these protections have been repeatedly shown to be vulnerable to fault injection (e.g. voltage and clock glitching) and similar physical attacks as well as logical attacks [30,31,32,33,34,35,36], simple read-out with an off-the-shelf programmer is prevented. Again, in case of password use, this password could be made available to the device owner.

Raw Flash Dump It is hard to prevent the direct dump of external flash memory, especially eMMC, which only requires a few connections and a low-cost SD card reader. Some processors provide means to encrypt the firmware stored in external flash, e.g. the ESP32 [37]. If such features are available, they should be activated. Otherwise, it may be at least possible to mitigate straightforward in-circuit dumps by routing all flash connections on inner layers of the PCB

Fig. 4. Covered PCB of an industrial IoT device

(without test pads) when BGA packages are used. Alternatively, the entire PCB can be covered in epoxy or other materials to prevent access to the flash chip as shown in Fig. 4. This thwarts direct access, but can still be removed with more effort using heat or chemicals.

Secure Boot and SELinux The Echo Plus is an example of an IoT device with stronger security measures compared to most other devices. Through the use of Android, trusted boot and SELinux, even though the firmware can be extracted, obtaining root access is difficult compared to other devices. It appears that SELinux, which is often considered hard to properly configure for a desktop system, might be suitable for IoT devices, which usually only provide limited and defined functionality. This is especially in light of the worrying practice to run all services with `root` permissions, which we encountered on many IoT devices. In addition, techniques to cryptographically verify the filesystem (e.g. `dm-verity`) or possibly also firmware encryption (if supported by the underlying processor, see e.g. [37]) should be considered for future IoT devices.

6 Conclusion

As shown in this paper, extracting firmware from IoT devices is possible through a variety of low-cost methods, with over 45% of the considered devices vulnerable to extraction through a simple UART connection. This problem exists throughout the industry, affecting high-profile devices like the first generation Echo as well as home hubs and alarm systems with significant security and privacy implications. Further details of all analyzed devices (notes, photographs, boot logs, etc.) are available at `https://github.com/david-oswald/iot-fw-extraction`.

We considered whether our work requires responsible disclosure to the affected manufacturers. However, our survey did not focus on the discovery of

vulnerabilities in the considered devices. Furthermore, in some cases, a similar technique had already been disclosed by a third party (e.g. [23,19,21]). Therefore, we decided not to engage in a formal disclosure process.

We plan to widen our survey, analysing additional devices and developing new methods for firmware extraction where necessary. An interesting approach in this regard is to analyse the low-level bootloaders integrated in the ROM of most modern processors w.r.t. to undocumented functions or implementation errors. Besides, it would also be interesting to better understand the susceptibility of firmware encryption mechanisms to physical attacks, e.g. side-channel analysis.

References

1. Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh, "IoT security: ongoing challenges and research opportunities," in *SOCA '14*. IEEE, 2014, pp. 230–234.
2. A. Riahi, Y. Challal, E. Natalizio, Z. Chtourou, and A. Bouabdallah, "A systemic approach for IoT security," in *DCOSS '15*. IEEE, 2013, pp. 351–355.
3. Y. H. Hwang, "IoT security & privacy: threats and challenges," in *IoTPTS '15*. ACM, 2015, pp. 1–1.
4. A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 95–110.
5. S. L. Thomas, T. Chothia, and F. D. Garcia, "Stringer: Measuring the importance of static data comparisons to detect backdoors and undocumented functionality," in *European Symposium on Research in Computer Security, ESORICS*, 2017.
6. S. L. Thomas, F. D. Garcia, and T. Chothia, "Humidify: A tool for hidden functionality detection in firmware," in *Detection of Intrusions and Malware, and Vulnerability Assessment DIMVA*, 2017.
7. B. Herzberg, D. Bekerman, and I. Zeifman, "Breaking Down Mirai: An IoT DDoS Botnet Analysis," https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html (visited on 25/05/2018.
8. Zenofex, 0x00string, CJ_000, Maximus64, "All Your Things Are Belong To Us," 2017, presentation at Defcon 2017.
9. exploitee.rs, "Exploitee.rs Wiki," https://www.exploitee.rs/ (visited on 20/05/2018).
10. O. Shwartz, Y. Mathov, M. Bohadana, Y. Elovici, and Y. Oren, "Opening Pandora's Box: Effective Techniques for Reverse Engineering IoT Devices," in *Smart Card Research and Advanced Applications — CARDIS '17*, T. Eisenbarth and Y. Teglia, Eds. Springer, 2017, pp. 1–21.
11. S. Glass, "Das U-Boot – the Universal Boot Loader," https://www.denx.de/wiki/U-Boot (visited on 20/05/2018).
12. A. Huang, "Hacking the Xbox: an introduction to reverse engineering," *no starch press*, 2002.
13. A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation." in *NDSS '13*. The Internet Society, 2013.
14. Z. Basnight, J. Butts, J. Lopez Jr, and T. Dube, "Firmware modification attacks on programmable logic controllers," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, pp. 76–84, 2013.

15. J. Grand, "JTAGulator," http://www.grandideastudio.com/jtagulator/ (visited on 20/05/2018).
16. A. Etemadieh, C. Heres, and K. Hoang, "Hacking Hardware with a $10 SD card reader," 2017, presentation at BlackHat USA 2017.
17. EthicalHacker523, "Hardware Hacking of Accu-Chek Performa Insight," https://hackaday.io/project/41162-hardware-hacking-of-accu-chek-performa-insight/ (visited on 20/05/2018).
18. I. Clinton, L. Cook, and S. Banik, "A Survey of Various Methods for Analyzing the Amazon Echo," 2016, https://vanderpot.com/Clinton_Cook_Paper.pdf (visited on 25/05/2018).
19. J. Hyde and B. Moran, "Alexa, are you Skynet?" *presentation at SANS DFIR Summit 2017*, 2017.
20. micaksica, "Exploring the Amazon Echo Dot, Part 2: Into MediaTek utility hell," https://medium.com/@micaksica/exploring-the-amazon-echo-dot-part-2-into-mediatek-utility-hell-b452f62e5e87 (visited on 10/05/2018).
21. OpenWRT forum, "Philips Hue Bridge v2 hacked (root access)," https://forum.openwrt.org/viewtopic.php?id=66346 (visited on 20/05/2018).
22. Texas Instruments, "How to Make 3 Partition SD Card," http://processors.wiki.ti.com/index.php/How_to_Make_3_Partition_SD_Card (visited on 20/05/2018).
23. M. Barnes, "Alexa, are you listening?" 2017, https://labs.mwrinfosecurity.com/blog/alexa-are-you-listening (visited on 25/05/2018).
24. "Binwalk," 2017, https://github.com/devttys0/binwalk (visited on 20/05/2018).
25. "Easylzma," https://github.com/lloyd/easylzma (visited on 20/05/2018).
26. Segger, "J-Link Debug Probes," 2017, https://www.segger.com/jlink-debug-probes.html (visited on 20/05/2018).
27. SELinux Wiki, "Guide/Mode — SELinux Wiki," https://selinuxproject.org/w/?title=Guide/Mode&oldid=808 (visited on 28/05/2018).
28. L. Yuxiang, Q. Wenxiang, and W. Huiyu, "Breaking Smart Speaker – Exploit Amazon Echo," https://github.com/tencentbladeteam/Exploit-Amazon-Echo (visited on 10/06/2018).
29. T. Dullien, "Closed, heterogenous platforms and the (defensive) reverse engineers dilemma," 2018, presentation at SSTIC 2018. [Online]. Available: https://www.sstic.org/2018/presentation/2018_ouverture/
30. S. Skorobogatov, "Copy Protection in Modern Microcontrollers," https://www.cl.cam.ac.uk/~sps32/mcu_lock.html (visited on 05/05/2018).
31. T. Goodspeed, "Side Channel Timing Attacks on MSP430 Microcontroller Firmware," 2008, presentation at BlackHat USA 2008.
32. D. Strobel, D. Oswald, B. Richter, F. Schellenberg, and C. Paar, "Microcontrollers as (In)Security Devices for Pervasive Computing Applications," *Proceedings of the IEEE*, vol. 102, pp. 1157–1173, 08 2014.
33. J. Obermaier and S. Tatschner, "Shedding too much light on a microcontroller's firmware protection," in *WOOT '17*. USENIX Association, 2017.
34. R. Pareja and N. Wierma, "Automotive microcontrollers. Safety != Security," *presentation at SHA2017*, 2017.
35. D. Nedospasov, "NXP LPC1343 Bootloader Bypass," https://toothless.co/blog/bootloader-bypass-part1/ (visited on 10/05/2018).
36. M. E. Scott, "The FaceWhisperer for USB Glitching; or, Reading RFID with ROP and a Wacom Tablet," *PoC||GTFO 0x13*, 2016.
37. ESP-IDF, "ESP32 Flash Encryption," https://esp-idf.readthedocs.io/en/latest/security/flash-encryption.html (visited on 20/05/2018).