Chapter 11
Finding Software Bugs in Embedded Devices

Aurélien Francillon, Sam L. Thomas, and Andrei Costin

Abstract The goal of this chapter is to introduce the reader to the domain of bug discovery in embedded systems which are at the core of the Internet of Things. Embedded software has a number of particularities which makes it slightly different to general purpose software. In particular, embedded devices are more exposed to software attacks but have lower defense levels and are often left unattended. At the same time, analyzing their security is more difficult because they are very "opaque", while the execution of custom and embedded software is often entangled with the hardware and peripherals. These differences have an impact on our ability to find software bugs in such systems. This chapter discusses how software vulnerabilities can be identified, at different stages of the software life-cycle, for example during development, during integration of the different components, during testing, during the deployment of the device, or in the field by third parties.

11.1 The Challenges of Embedded Devices and Software

We argue that the problem of embedded software security is due to multiple factors, including a systematic lack of transparency, control, and resistance to attacks. A particular way to improve this is to analyze the software of these devices, with the particular goal of identifying software vulnerabilities in order to correct them as early as possible.

A. Francillon (✉)
EURECOM, Sophia Antipolis, France
e-mail: aurelien.francillon@eurecom.fr

S. L. Thomas
University of Birmingham, Birmingham, United Kingdom

A. Costin
Faculty of Information Technology, University of Jyväskylä, Jyväskylä, Finland

© The Author(s) 2021

G. Avoine, J. Hernandez-Castro (eds.), *Security of Ubiquitous Computing Systems*,
https://doi.org/10.1007/978-3-030-10591-4_11

11.1.1 Lack of Transparency

Today, many smart devices are compromised during massive attacks, and may be abused to form large *botnets* (networks of compromised devices). Record-high Distributed Denial of Service (DDoS) attacks (i.e., network flooding) reportedly generated between 620 Gbps and 1 Tbps of traffic [241, 344]. These DDoS attacks were reported to use several hundred thousand compromised embedded/smart devices, comprising dozens of different models of Commercial Off-The-Shelf (COTS) products like IP/CCTV cameras and home routers. Most of those devices were compromised using default or hard-coded credentials set by the manufacturer [345]. Malware running on such devices has complete control over the traffic that is generated, and most smart devices do not embed any infection detection or prevention mechanism. Worse yet, the users or owners of the device are often not aware of the problem, and unable to solve it. In fact, devices are not designed to be inspected and modified by end-users (e.g., to perform forensics as discussed in Chap. 13).

11.1.2 Lack of Control

Another important problem is that smart devices are generally provided as a fixed software (i.e., firmware) and hardware platform, often tied to a cloud service and bundled together as a closed system that the user has little control over. An example of the negative consequences of this customer lock-out is the *Revolv* smart thermostat. *Revolv*'s manufacturer was acquired by its competitor Nest, and after a year Nest stopped the cloud service, rendering the *Revolv* thermostats installed in homes impossible to use [271]. Users often have no choice regarding which software the device should run, or which cloud service to use, or what the device should do. Choosing, installing and using alternative software for such devices is difficult, if not impossible, often due to the single-purpose nature of the hardware and software design, the lack of public documentation, in addition to any anti-tampering measures imposed by the manufacturer.

11.1.3 Lack of Resistance to Attacks

In practice, Internet scanning botnets are active enough that some devices will be compromised within a few minutes after being connected to the Internet [344]. To be considered trustworthy, devices need to have a certain level of resistance to attacks. This is astonishing, because in essence many of the recurring security issues with smart devices have already been "solved" for many years. If vulnerabilities and corresponding attack situations could ultimately be avoided, it is important to ask who is responsible for the damage caused by compromised smart devices, beyond the malware author. The device owner may be legally responsible, but often the end-user does not have any means to detect or prevent such compromises, or to apply a secure configuration. On the other hand, the manufacturers currently often have no legal liability, and thus no incentive (e.g., economic, legal) to prevent a potential vulnerability and compromise.

11.1.4 Organization of This Chapter

Solving these problems requires analyzing the software and firmware for the embedded devices, and identifying and fixing their vulnerabilities. This chapter describes the possible steps to systematically and consistently achieve this goal. We first provide a classification of embedded systems that is well adapted to their analysis. We then describe the possible steps for their analysis. We start with ways to obtain the software to analyze, which is often a challenge in itself for embedded devices. We then describe how to perform static analysis on the firmware packages obtained, which has many advantages such as speed and scalability. We then describe techniques which can be used to dynamically analyze the firmware, which in contrast to static analysis has the advantage of larger code coverage and lower false positive rates.

11.1.5 Classification of Embedded Systems

A general definition of embedded systems is hard to establish [261]. However, two widely accepted differences separate embedded devices from modern general-purpose computers, such as ordinary desktop PCs or smartphones, namely: (a) they are designed to fulfill a *specific purpose*, and (b) they heavily interact with the physical world via *peripherals*. The aforementioned two criteria cover a wide variety of devices, ranging from hard-disk controllers to home routers, from digital cameras to Programmable Logic Controllers (PLCs). These families can be further classified according to several aspects, such as their actual computing power [171], the extent to which they interact with their computing and physical environment, their field of usage, or the timing constraints imposed on them.

Unfortunately, these classifications tell us very little about the type of security mechanisms that are available on a given device. Muench et al. [430] classifies embedded systems according to the type of operating system (OS) they use. While the operating system is certainly not the only source of security features, it provides several security primitives, handles recovery from faulty states, and often serves as a building block for additional and more complex security mechanisms. We therefore classify embedded devices using the following taxonomy:

Type-0: Multipurpose / non-embedded systems.
We use Type-0 in order to reference traditional general-purpose systems.

Type-I: General purpose OS-based devices (e.g., Linux-based).
The Linux OS kernel is widely used in the embedded world. However, in comparison to the traditional GNU/Linux found on desktops and servers, embedded systems typically follow more minimalist approaches. For example, a very common configuration that can be found in consumer-oriented products as well as in Industrial Control Systems (ICS) is based on the Linux kernel coupled with `BusyBox` and `uClibc`.

Type-II: Embedded OS-based devices.
These dedicated operating systems targeted at embedded devices systems are particularly suitable for devices with low computation power, which is typically enforced on embedded systems for cost reasons. Operating systems such as `uClinux` or `FreeRTOS` are suitable for systems without a Memory Management Unit (MMU) and are usually adopted on single-purpose user electronics, such as IP cameras, DVD players and Set-Top Boxes (STB).

Type-III: Devices without OS-Abstraction.
These devices adopt a so called "monolithic firmware", whose operation is typically based on a single control loop and interrupts triggered by the peripherals in order to handle external events. Monolithic firmware can be found in a large variety of controllers of hardware components, such as CD-readers, WiFi-cards or GPS-dongles.

11.2 Obtaining Firmware and Its Components

Even though complete black box analysis of embedded devices is possible to some degree and in certain situations, obtaining the firmware significantly helps and makes more advanced analyses possible. There are two main ways to obtain the firmware for a given device—as a firmware package (e.g., online, support media) and through extraction from the device itself.

11.2.1 Collecting Firmware Packages

The environments in which embedded systems are deployed are heterogeneous, spanning a variety of devices, vendors, CPU/hardware architectures, instruction sets, operating systems, and custom components. This makes the task of compiling a *representative* and *balanced* dataset of firmware packages a difficult problem to solve. The lack of centralized points of collection, such as the ones provided by software/app marketplaces, antivirus vendors, or public sandboxes in the malware analysis field, makes it difficult for researchers to gather large and well triaged datasets. Firmware often needs to be downloaded from vendor Web pages and FTP sites, and it is not always simple, even for a human, to tell whether or not two firmware packages are for the same physical device.

One challenge often encountered in firmware analysis and reverse engineering processes is the difficulty of reliably extracting meta-data from a firmware package. This meta-data might include, the device's vendor, its product code and purpose, its firmware version, or its processor architecture, among countless other details.

11.2.2 Extracting Firmware from Devices

Obtaining the firmware from an online repository as a firmware package is convenient and thus preferred, however it is not always possible. First, the firmware may not be available, e.g., because there is no update yet, nor one planned. Sometimes the firmware is only distributed through authorized and qualified maintenance agents, e.g., in case of industrial or critical systems. It is also common that the firmware is not distributed at all in an attempt to prevent counterfeit products, reverse engineering of the software or protecting its security.

In such cases the best (and sometimes the only) solution is to extract the firmware from the device itself. There are multiple possible ways to approach this ([529] and [564] provide a detailed overview of the process), each approach having its own set of benefits and issues. In the simplest case, the firmware can be extracted by connecting to a debug interface (e.g., JTAG, and serial ports such as UART, SPI, I2C). It is important to note that JTAG is a low level protocol and many different mechanisms can be implemented on top of it. Debug mechanisms allow dumping some memories (e.g., ROM, RAM or Flash memories behind a Flash controller), but not necessarily others. When Flash memory is soldered onto a Printed Circuit Board (PCB) and is independent from the processor, it is possible to de-solder it and extract its contents using a Flash programmer/reader. Unfortunately, the variety of Flash memory standards, types and pinouts is huge. One can design their own Flash chip adapter for reading and dumping the memory contents (e.g., code, data) [75]. However, some cheap universal programmers may be sufficient for dumping sufficiently many models of Flash memories [38]. Finally, the advanced Flash programmers support even hundreds of thousands of different Flash memory models [198].

However, when the device is a Flash microcontroller, the Flash memory is integrated within the microcontroller and is typically not directly accessible. In such cases, the microcontrollers themselves provide mechanisms to access Flash memory areas, but often such mechanisms come with some Flash area protection mechanisms, which are often arbitrary and microcontroller specific. Such protection mechanisms can sometimes be bypassed due to vulnerabilities in the implementation of the protections themselves [447, 556]. However, such attacks may not always succeed, and one may be left with using more costly invasive hardware attacks such as Linear Code Extraction (LCE) [549] or direct memory readout using a microscope [158] as the only option available.

11.2.3 Unpacking Firmware

The next step towards the analysis of a firmware package is to unpack and extract the files or resources it contains. The output of this phase largely depends on the type of firmware, as well as the unpacking and extraction tools employed. In some examples, executable code and resources (such as graphics files or HTML code) might be embedded directly into a binary blob that is designed to be directly copied into memory by a bootloader and then executed. Some other firmware packages are distributed in a compressed and obfuscated package which contains a block-by-block image copy of the Flash memory. Such an image may consist of several partitions containing a bootloader, a kernel, a file system, or any combination of these.

11.2.4 Firmware Unpacking Frameworks

The main tools to unpack arbitrary firmware packages are: binwalk [263], FRAK [161], Binary Analysis Toolkit (BAT) [558] and Firmware.RE [155] (Table 11.1 compares the performance of each framework):

- Binwalk is perhaps the best known and most used firmware unpacking tool developed by Craig Heffner [263]. It uses pattern matching to locate and carve files from a binary blob. Additionally, it also extracts some meta-data such as license strings.
- FRAK is an unpacking toolkit first presented by Cui et al. [162]. It reportedly[1] supports a limited number of device vendors and models, such as HP printers and Multi-Function Peripherals (MFP).
- The Binary Analysis Toolkit (BAT), formerly known as GPLtool, was originally designed by Armijn Hemel and Tjaldur software in order to detect GPL license violations [269, 558]. To do so, it recursively extracts files from a binary blob and matches strings with a database of known strings from GPL projects and licenses. BAT also supports file carving similar to binwalk, as well as a very flexible plugin-oriented extension interface.

[1]Even though the authors mention that the tool would be made publicly available, it has yet to be released.

Table 11.1 Comparison of the unpacking performance of Binwalk, BAT, FRAK and Firmware.RE on a few example firmware packages (according to [155])

Device	Vendor	OS	Binwalk	BAT	FRAK	Firmware.RE
PC	Intel	BIOS	✗	✗	✗	✗
Camera	STL	Linux	✗	✓	✗	✓
Router	Bintec	–	✗	✗	✗	✗
ADSL gateway	Zyxel	ZynOS	✓	✓	✗	✓
PLC	Siemens	–	✓	✓	✗	✓
DSLAM	–	–	✓	✓	✗	✓
PC	Intel	BIOS	✓	✓	✗	✓
ISDN server	Planet	–	✓	✓	✗	✓
Voip	Asotel	Vxworks	✓	✓	✗	✓
Modem	–	–	✗	✗	✗	✓
Home automation	Belkin	Linux	✗	✗	✗	✓
			55%	64%	0%	82%

- Firmware.RE [155] extends BAT with additional unpacking methods and specific analyses to perform automated large-scale analyses. When released, it achieved a lower false positive rate when unpacking firmware compared to binwalk.

11.2.5 Modifying and Repacking Firmware

Modifying and repacking a firmware could be one optional step during the analysis of the firmware and device security. The modifications could be performed either at the level of the entire firmware package, or at the level of individually unpacked files (that are finally repacked back into a firmware package). Such a step could be useful in testing several things. First, it can check whether a particular firmware has error, modification and authenticity checks for new versions of firmware. If such checks are missing or improperly implemented, the firmware update mechanism can then be used as an attack vector, or as a way to perform further analysis of the system [57, 162]. Second, it can be used to augment the firmware with additional security-related functionality, such as exploits, benign malware and more advanced analysis tools. For example, this could be useful when there are no other ways to deliver an exploit (e.g., non-network local exploits such as kernel privilege escalation), or provide some (partial) form of introspection into the running device/firmware [163].

The `firmware-mod-kit` tool [262] is perhaps the most well-known (and possibly among the very few) firmware modification tools. Unfortunately, it supports a limited number of firmware formats, and while it can be extended to support more formats, to do so requires substantial manual effort. Further, for some formats it relies on external tools to perform some of the repacking. These tools are developed and maintained by different persons or entities in different shapes and forms, thus there is no uniform way to modify and repack firmware packages.

11.3 Static Firmware Analysis

Once the code is extracted further analysis can be performed. There are two main classes of analysis that can be preformed on a generic computing system—static analysis and dynamic analysis. In principle, the distinction between the two is easy: in static analysis the code is analyzed without executing it, but instead only reasoning about it, while in the dynamic setting the analysis is performed on the code while it is executed. With more advanced analysis techniques, however, this frontier is slightly blurred. For example, symbolic execution allows one to analyze software by considering some variables to have an unknown value (i.e., they are unconstrained). Symbolic execution is sometimes considered static analysis and at other times dynamic analysis. In this section, we will first describe simple static analysis which can be efficiently performed on firmware packages, then we will discuss more advanced static analysis approaches. Finally, we will cover the limitations of static analysis and in the next section focus on the dynamic analysis on firmware packages.

11.3.1 Simple Static Analysis on Firmware Packages

11.3.1.1 Configuration Analysis

For a large majority of complex embedded devices (i.e., those of Type-I as described in Sect. 11.1.5), while service configuration is stored within the file-system of the device, user-configurable information is often stored elsewhere—within a region of memory called Non-Volatile Random Access Memory (NVRAM) which retains its state between power cycles (similar to Flash memory in some ways). Many devices treat NVRAM as a key-value store and include utilities such as `nvram-get` and `nvram-set`, as well as dedicated libraries to get and set values stored there. On a router, for example, the current Wi-Fi passphrase and web-based configuration interface credentials, will often be stored within the NVRAM, which will be queried by software in order to facilitate the authentication of the device and its services.

All other device configuration, without performing a firmware upgrade, will be static. As a result of this, any, e.g., hard-coded passwords or certificates (as noted in [151]), can be leveraged by an adversary to compromise a device. To this end, Costin et al. [155] show many instances where devices are configured with user accounts and passwords that are weak, missing entirely, or stored in plain-text. Therefore, a first step in static analysis of firmware is to examine the configuration of its services: to check for improperly configured services, e.g., due to use of

```
mount -t proc proc /proc
mount -t ramfs ramfs /var
mkdir /var/tmp
mkdir /var/ppp/
mkdir /var/log
mkdir /var/run
mkdir /var/lock
mkdir /var/flash
#iwcontrol is required for RTL8185 Wireless driver
#iwcontrol auth &

#busybox insmod /lib/modules/2.4.26-uc0/kernel/drivers/usb/quickcam.o

/bin/webs -u root -d /www -i /var/run/thttpd.pid &
#ifconfig wlan0 up promisc
```

Fig. 11.1 Example of a boot script taken from an IP camera

unsafe defaults and hard-coded credentials. Configuration files are of further use in estimating the set of programs utilized and the initial global configuration of a device, in the absence of physical access to it. For example, by examination of its boot scripts, we are able to learn which services present in its firmware (among potentially hundreds) are actually utilized by the device, this can aid in reducing the amount of time taken by more complex analysis approaches described later.

Manual methods are often sufficient for analysis of a few firmware images and, with limited scope, analysis of things such as the device's configuration. For example, to estimate the set of processes started by a firmware one can inspect the contents of a boot script, e.g., `/etc/rcS`.

Figure 11.1 details such a boot script taken from the firmware of an IP camera. We are able to observe that the device's primary functionality is orchestrated by the `/bin/webs` binary, which we would then analyze further using the methods detailed in Sect. 11.3.2.

11.3.1.2 Software Version Analysis

Many devices are not designed to receive firmware updates. This prohibits patching against known security vulnerabilities and can often render a device useless to an end-user. This prevents abusing the firmware update as an attack vector. However, when a vulnerability is discovered, the only effective mitigation is to replace the device with a new one.

Many devices are designed to be updated and vendors provide firmware updates. However, the mechanisms for applying those updates are often not standardized and are largely ad-hoc. They also heavily rely on the end-user's diligence (to identify that an update is available) and action (to actually apply the updates). The end-result of this is that an overwhelming majority of devices are left unpatched against known vulnerabilities. Thus, a further step in the analysis of firmware is to identify the versions of software (both programs and libraries) it contains, and correlate those versions with known vulnerabilities (e.g., CVE database).

There are several possible approaches to perform this. For example, [155] use fuzzy hashing [340, 507] as a method to correlate files in firmware images. The effectiveness of the approach was demonstrated in several examples, in particular uncovering many IoT and embedded devices being so-called "white label" products.[2] Finally, machine learning can be used to identify firmware images [157] or to search for known vulnerabilities [585].

11.3.2 Static Code Analysis of Firmware Packages

Developing tools for performing automated static code analysis on embedded device firmware presents a number of complexities compared to performing analyses on software for commodity PC systems (i.e., Type-0 devices). The first challenge is the diversity of CPU architectures. This alone restricts the amount of existing tooling that can be used, and when attempting large scale analysis tools will inevitably have to deal with firmware from a number of distinct architectures. To facilitate the analysis in this case, the algorithms will either have to be reimplemented for each architecture being analyzed, or the architecture-specific disassembled firmware instructions will have to be lifted to a common, so-called Intermediate Language (IL) or Intermediate Representation (IR). A further difficulty for more simple devices (e.g., those of Type-III) is the often non-standard means by which different device firmware executes (e.g., it could be interrupt driven) and interacts with the memory and external peripherals. More complex firmware (e.g., that of Type-I devices) tends to more closely follow the execution behavior of more conventional devices (those of Type-0).

11.3.2.1 Code Analysis of Embedded Firmware

Despite the increased complexity of performing automated analysis of embedded device firmware, a number of techniques have been proposed for both targeted and large-scale static analysis. Eschweiler et al. [202] and Feng et al. [212] use numeric feature vectors to perform graph-based program comparisons [191] efficiently. They encode control-flow and instruction information in these feature vectors to identify known vulnerabilities in device firmware. Both methods provide a means of querying a data-set of binaries using a reference vulnerability as input and identifying the subset of binaries that contain constructs that are *similar* (but not necessarily the same) to those of the input vulnerability. The work in [585] improves the performance of these approaches by relying on Neural Networks.

[2]Generic products which are sold under a known brand.

11.3.2.2 Discovering Backdoors with Static Analysis

Aside from vulnerability discovery, a small body of work has attempted to automatically identify backdoor-like constructs in device firmware. Static analysis is most suited to detecting such constructs due to the fact it can achieve full program coverage. Dynamic analysis is less adequate in this case, as it relies solely on execution traces that can be captured and analyzed stemming from triggering standard program behaviors (which, by definition [551], a backdoor is not).

HumIDIFy[3] [552] uses a combination of Machine Learning (ML) and static analysis to identify anomalous and unexpected behavior in services commonly found in Linux-based firmware. ML is used first to identify the type of firmware binaries, e.g., a web-server, this then drives classification-specific static analysis on each binary. HumIDIFy attempts to validate that binaries do not perform any functionality outside of what is expected of the type of software they are identified as. For example, HumIDIFy is able to detect a backdoor within a web-server taken from Tenda router firmware[4] that contains an additional UDP listening thread which executes shell commands provided to it (without authentication) as the root user.

Stringer[5] [550] attempts to locate backdoor-like behavior in Linux-based firmware. It automatically discovers comparisons with static data that leads to execution of *unique* program functionality, which models the situation of a backdoor providing access to undocumented functionality via a hard-coded credential pair or undocumented command. Stringer provides an ordering of the functions within a binary based on how much their control-flow is influenced by static data comparisons that *guard* access to functionality not otherwise reachable. The authors demonstrate Stringer is able to detect both undocumented functionality and hard-coded credential backdoors in devices from a number of manufacturers.

Firmalice [528] is a tool for detecting authentication bypass vulnerabilities and backdoors within firmware by symbolic execution. It takes a so-called security policy as input, which specifies a condition a program (or firmware) will exhibit when it has reached an authenticated state. Using this security policy, it attempts to prove that it is possible to reach an authenticated state by discovering an input that when given to the program satisfies the conditions to reach that state. To discover such an input, Firmalice employs symbolic execution on a program slice taken from a program point acting as an input source to the point reached that signals the program is in an authenticated state. If it is able to satisfy all of the constraints such that a path exists between these two points, and an input variable can be concretised that satisfies those constraints, then it has discovered an authentication bypass backdoor (and a triggering input)—such an input will not be discoverable in a non-backdoored authentication routine. Unfortunately, Firmalice requires a degree of manual intervention to perform its analysis, such as identifying the security policy,

[3]Available as open-source: https://github.com/BaDSeED-SEC/HumIDIFy.

[4]http://www.devttys0.com/2013/10/from-china-with-love/.

[5]Available as open-source: https://github.com/BaDSeED-SEC/strngr.

input points and privileged program locations. It is therefore not easily adaptable for large-scale analysis.

11.3.2.3 Example Static Analysis to Discover Code Parsers

In order to interact with remote servers or connecting clients (e.g., for remote configuration), most firmware for networked embedded devices will contain client/server components, e.g., a web-server, or proprietary, domain-specific client/server software. In all cases, the firmware itself or software contained within it (for more complex devices) will implement parsers for handling the messages of the protocols required to communicate with corresponding client/server entities. Such parsers are a common source of bugs, whether their implementation incorrectly handles input in a way that causes a memory corruption, or permits an invalid state transition in a protocol's state machine logic. Thus, identifying these constructs in binary software is useful as a premise to performing targeted analyses. To this end, Cojocar et al. [150], propose PIE, a tool to automatically detect parsing routines in firmware binaries. PIE utilizes a supervised learning classifier trained on a number of simple features of the LLVM IL representation of firmware components known to contain parsing logic. Such features include: basic block count, number of incoming edges to blocks, and number of callers (for functions). PIE provides a means to identify specific functions responsible for performing parsing within an input firmware package, or software component. Stringer [550], described in Sect. 11.3.2.2, similarly provides a means of automatically identifying parser routines (for text-based input); in addition to identifying routines, it is also able to identify the individual (text-based) commands, processed by the parser.

11.4 Dynamic Firmware Analysis

Static analysis is indeed a robust technique that can help discover a wide range of vulnerability classes, such as misconfigurations or backdoors. However, it is not necessarily best suited for other types of vulnerabilities, especially when they depend on the complex runtime state of the program.

Similar to static analysis, powerful dynamic analysis techniques and tools have been developed for traditional systems and general purpose computers. However, the unique characteristics and challenges of the embedded systems make it difficult, if not impossible, to directly apply those proven methods. To this end, there are several distinct directions for dynamic analysis of embedded systems and we briefly discuss them below.

11.4.1 Device-Interactive Dynamic Analysis Without Emulation

When the device is present for analysis, the simplest form of device-interactive dynamic analysis is to test the devices in a “black-box” manner. The general idea of this approach is to setup and run the devices under analysis as in normal operation (e.g., connect to Ethernet LAN, WLAN, smartphone), and then test it with various tools and manual techniques (e.g., generic or specialized fuzzers, web penetration) and observe their behavior via externally observable side-effects such as device reboots, network daemon crashes, or XSS artifacts [439]. Similar approaches and results were reported by several independent and complementary works [259, 280, 292].

While being simple and easy to perform, this type of dynamic analysis has certain limitations, some of which are due to the “black-box” nature of the approach. For example, it is challenging to know what is happening with the entire system/device while the dynamic analysis is performed, i.e., the introspection of the system is missing or is hard to achieve. Also, in this approach it is not easy to control in detail what specifically is being executed and analyzed, the analysis being mostly driven by the data and actions fed to the device. In addition to this, some types of vulnerabilities might not have side-effects that are immediately [430] or externally visible (e.g., a crash of a daemon which does not necessarily expose a network port), therefore those bugs could be missed or misinterpreted during the analysis.

11.4.2 Device-Interactive Dynamic Analysis with Emulation

As an extension to the aforementioned approach, emulation can be coupled with device-interactive dynamic analysis to provide the required depth and breadth, therefore outperforming other static or dynamic analysis methods. The general idea of this approach is to split the execution of the embedded firmware between the analysis host and the actual running device. The analysis host is connected to the device via a debug (e.g., JTAG) or serial (e.g., UART) interface. Therefore one requirement is that the device under analysis must provide at least such an interface, whether documented or not. The analysis host then runs a dynamic analysis environment which is typically an emulator (e.g., QEMU-based) augmented or extended with additional layers and plugins such as symbolic execution and taint analysis. The analysis host has access to the execution and memory states both for the emulator and for the running device. The firmware is being analyzed first in the extended emulator environment. During the firmware emulation and analysis, certain parts of the analyzed firmware are transferred for execution by the analysis host from the emulator to the running device. This is sometimes required, for example, when the firmware needs to perform an I/O operation with a peripheral present on the devices but not in the emulator. The execution and state transfer to and from the device occur via the connected debug or serial interface. On the one hand, by using this approach it is possible to control exactly what is to be analyzed because the emulator is under the full supervision of the analysis host. On the other hand, this approach enables broader and deeper coverage of the execution because the device can complement the execution of firmware parts that are impossible to execute within the emulator.

This is the approach followed by Avatar [591] which aims at providing symbolic execution with S2E [140], while Avatar2 [429] focuses on better interoperability with more tools. Prospect [312] explores forwarding at the system calls level and Surrogates [341] provides a very fast debug interface. Inception [125] provides an analysis environment to use during testing when source code is available.

11.4.3 Device-Less Dynamic Analysis and Emulation

Performing dynamic analysis in a device-interactive manner certainly has its benefits, however such an approach has a number of limitations and is hard to fully automate. Firstly, it is not easy to scale the human operator's interventions and expertise required for many of the tasks related to the approach of device interaction with emulation. Secondly, it is challenging to automate and scale the logistics operations related to acquisition, tear-down, connection, configuration and reset of a large number of devices. Therefore, dynamic analysis techniques that are easier and more feasible to scale and automate are required. One such technique is the device-less analysis based on full or partial emulation.

Davidson et al. [175] presented the FIE tool that detects bugs in firmware of the MSP430 microcontroller family. FIE leverages KLEE [120] to perform symbolic execution of firmware in order to detect memory safety violations (e.g., buffer overflows and out-of-bounds memory accesses), and misuse of peripherals (e.g., attempted writes to read-only memory). FIE needs the availability of the source code, which is uncommon, and is able to handle a variety of the nuances and challenges faced during automated analysis of firmware, especially when dealing with firmware for Type-III devices. However, when reading I/O from a device, the values read are always assumed to return unconstrained (completely symbolic) values which leads to a state explosion problem. This limits the size of the programs which can be analyzed.

In [156], the authors perform device-less dynamic security analysis via automated and large-scale emulation of embedded firmware. Similarly, FIRMADYNE [137] presents an automated and scalable system for performing emulation and dynamic analysis of Linux-based embedded firmware.

The general idea of both works is to crawl and then unpack firmware packages into minimal root filesystems (i.e., `rootfs`) that can subsequently be virtualized and executed as a whole via "system emulation" (as opposed to "user emulation") using for example QEMU [69]. The emulator is first used to start an architecture-specific emulation host OS, such as Debian for ARM or MIPS depending on the architecture of the device whose firmware is being dynamically analyzed. Then the firmware root filesystem is uploaded to the emulation host OS, where its Linux boot sequence scripts are initiated, most likely in a `chroot` environment under the emulation host OS. Once the firmware's Linux boot sequence concludes, various services (e.g., a web server, SSH, telnet, and FTP) of the device/firmware under analysis should be running, and are ready for logging, tracing, instrumentation and debugging. The work in [137] extends this approach by running a custom operating system kernel which is able to emulate some of the missing drivers.

11.5 Conclusion

We have provided a short overview of the field: from our excursus, it is clear that analyzing the software of IoT/embedded devices and finding security vulnerabilities within them is still a challenging task. While multiple directions and techniques are being actively explored and developed within the field, more research, insights and tools are still required.

Unfortunately, the existing proven techniques (e.g., static, dynamic, hybrid analysis) cannot be applied in a straightforward manner to embedded devices and their software/firmware. One reason for this is the high heterogeneity and fragmentation of the technological space that supports embedded/IoT systems. Another reason is the "opaque" nature of embedded devices, which can be seen as akin to the "security by obscurity" principle. Such reasons make embedded systems harder to analyze compared to more traditional systems.

Indeed, the current embedded firmware "population" may still contain many latent backdoors and vulnerabilities, both known and unknown. However, as we detailed in this chapter, positive and promising avenues for the detection of embedded software bugs are becoming increasingly available. Such avenues include large-scale analysis and correlation techniques, hybrid/dynamic analysis of emulated firmware or running devices, and advanced techniques to specifically detect backdoors.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.