IoT Hacking – A Primer

Dorottya Papp, Kristóf Tamás, and Levente Buttyán

***Abstract*—The Internet of Things (IoT) enables many new and exciting applications, but it also creates a number of new risks related to information security. Several recent attacks on IoT devices and systems illustrate that they are notoriously insecure. It has also been shown that a major part of the attacks resulted in full adversarial control over IoT devices, and the reason for this is that IoT devices themselves are weakly protected and they often cannot resist even the most basic attacks. Penetration testing or ethical hacking of IoT devices can help discovering and fixing their vulnerabilities that, if exploited, can result in highly undesirable conditions, including damage of expensive physical equipment or even loss of human life. In this paper, we give a basic introduction into hacking IoT devices. We give an overview on the methods and tools for hardware hacking, firmware extraction and unpacking, and performing basic firmware analysis. We also provide a survey on recent research on more advanced firmware analysis methods, including static and dynamic analysis of binaries, taint analysis, fuzzing, and symbolic execution techniques. By giving an overview on both practical methods and readily available tools as well as current scientific research efforts, our work can be useful for both practitioners and academic researchers.**

***Index Terms*—IoT security, ethical hacking, penetration testing, embedded firmware analysis, binary program analysis.**

I. INTRODUCTION

THE Internet has grown beyond a network of laptops, PCs, and large servers: it also connects millions of small embedded devices. This new trend is called the Internet of Things, or IoT in short, and it enables many new and exciting applications. At the same time, however, it also creates a number of new risks related to information security.

On the one hand, embedding computers into everyday objects and connecting them to the Internet exposes our physical world to attacks originating from the cyber space. This means that cyber attacks may have physical consequences, including damage of physical equipment or even loss of human life. Probably, the most famous example for this is the Stuxnet worm [1], which was used in an attack targeting a uranium enrichment plant in Iran to compromise embedded industrial controllers and to physically damage the uranium centrifuges that they controlled [2]. Another famous example is the proof-of-concept attack on the Jeep Cherokee SUV [3], in which two security researchers remotely took control over a vehicle while it was running on the highway. Besides these famous cases, there are many other examples for cyber attacks on network connected embedded systems (essentially IoT applications), where the consequences were or could have been highly undesirable, including an attack on the Ukrainian power grid that resulted in an hour long black-out in the city of Kiev [4], an attack on a steel mill in Germany that resulted in "massive damage to the system" [5], and a potential attack that installed malware on pacemaker devices that could have resulted in a fatality [6].

The other side of the coin is that embedded devices with no or weak protection, when connected to the Internet, can put Internet based services and the Internet infrastructure itself at risk. Indeed, weakly protected WiFi routers, web cameras, and other "smart" devices connected to the Internet are low hanging fruits for attackers that they can use to build a massive attack infrastructure. An example for this is the Mirai botnet [7], which consists in millions of compromised IoT devices and which was used in the largest DDoS (Distributed Denial of Service) attack ever targeting the Domain Name System of the Internet and making popular Internet based services unavailable [8].

The general insecurity of the Internet of Things is a problem, and researchers have started to investigate what it stems from and how to address it. In a recent survey [9], the authors performed a comprehensive study on reported attacks and defenses in the IoT domain with the goal of understanding what goes wrong with existing IoT applications in terms of security. They identified 5 major problem areas: unconditional trust in the local network and in the physical environment an IoT device is operating in, over-privileging mobile applications used to control IoT devices, no or weak authentication, and implementation flaws. The study found that a major part of the attacks resulted in full adversarial control over IoT devices. The reason for this is that IoT devices themselves are weakly protected and they often cannot resist even the most basic attacks.

Whether IoT devices can be made more resistant to attacks in a cost efficient way is an open question and subject to intense research. However, even if future devices will be more secure, there are millions of devices already deployed, and it is also important to understand the level of security that they provide. This can usually be measured to some extent by penetration testing or ethical hacking methods. Hacking IoT devices can be fun, because it combines traditional hacking methods with some hands-on physical experience, but more importantly, it is also a very useful activity that can help discovering and fixing vulnerabilities in IoT devices that, if exploited, can result in highly undesirable conditions, as we saw above.

In this paper, we give a basic introduction into hacking IoT devices. We begin with giving an overview on hardware hacking, as IoT hacking is often started by disassembling the IoT device under study. The vulnerabilities that can be exploited to gain full adversarial control over a device can often be found in the device's firmware. Therefore, we con-

The authors are affiliated with the CrySyS Lab at the Department of Networked Systems and Services of the Budapest University of Technology and Economics, e-mail: (see http://www.crysys.hu/).

Kristóf Tamás is currently with Ukatemi Technologies.

Manuscript received: February 2019; revised: May 2019.

DOI: 10.36244/ICJ.2019.2.1

tinue our introduction by explaining how the firmware can be extracted from the devices and unpacked. Then, we briefly summarize some basic firmware analysis methods and tools that aim at identifying hard-coded secrets, misconfigurations of the device, and simple bugs in scripts. Most of these tools are open source and freely available on the Internet, and we provide references to them. Finally, we complete our primer on IoT hacking by providing a survey on more advanced analysis methods, including static and dynamic analysis of binaries, taint analysis, fuzzing, and symbolic execution. Advanced binary analysis of embedded firmware is still an active area of research, hence, instead of tools readily available on the Internet as in the case of basic firmware analysis, advanced methods are mainly described in scientific publications. Accordingly, we provide references to the most relevant papers in this exciting research domain. We hope that this duality (i.e., giving an overview both on practical methods and readily available tools, as well as on current scientific research efforts) makes our work useful for both practitioners and academic researchers.

II. HARDWARE HACKING

In the IoT context, the IoT device being analyzed is often physically accessible to the hacker, which allows him/her to inspect the hardware components of the device, including chips and connectors soldered on the motherboard, and peripherals attached to it. Inspection of the hardware can be carried out in three phases:

1) **Hardware reconnaissance without opening the device:** In this phase, the main objective is to collect publicly available information about the hardware at hand, mainly from the Internet, as whatever information is discovered in this phase can be used later in the analysis. For instance, the serial or model number printed on the device may allow for the identification of data sheets or manuals on the Internet, which might include important information about the device. Wireless devices produced or used in the USA have an FCC ID (Federal Communication Commission Identifier) printed on them, which one can use to look up information on different web sites[1]. These web pages usually contain more information about the device than its data sheet, including the labelled motherboard, I/O (Input/Output) pins, test reports, and external and internal photos about the device. For the later phases, it is vital to identify the power requirements of the device and the needed adapters. The most important information include the level of amperage, the level of voltage, and the polarity. From the data sheets and photos, or by visually examining the device, it is also important to identify whether it has any kind of tamper protection, because opening a tamper protected device can lead to irreversible damage of the hardware. Finally, it might be possible to obtain public information about some known vulnerabilities of the device, which may be exploited without opening the housing of the device.
2) **Opening the housing of the device and inspecting the motherboard:** This phase usually requires more electrical engineering knowledge. Most importantly, it might be impossible to re-assemble the device into its original state after dismantling. Therefore, photos and notes have to be made and taken during the dismantling process. Once the device is open, the chips, pins, and interfaces on it can be inspected. With the chip identifiers found, a search on different web databases[2] can determine the purpose of the chip (e.g. processor, flash, RAM) and the function of its pins. In addition, the external communication interfaces, such as UART (Universal Asynchronous Receiver-Transmitter) or JTAG (Joint Test Action Group), are identified in this phase, as well as signs of use of communication protocols, such as SPI (Serial Peripheral Interface) or I2C (Inter-Integrated Circuit).
3) **Desoldering the chips form the motherboard (if necessary):** Sometimes, the pins of a chip cannot be accessed without desoldering the chip from the motherboard. For instance, to dump the content of a flash chip, the chip might need to be desoldered from the motherboard in order to solder it to an external adapter with connectable pins.

At the end of this phase, profound knowledge is gained about how the analyzed IoT device works at the hardware level. The next stage could be dumping the firmware from the device via SPI, gaining root access to the device via UART, or looking for vulnerabilities using JTAG. We discuss these techniques in the following sections.

A. The UART interface and protocol

UART (Universal Asynchronous Receiver-Transmitter) is an asynchronous serial communication protocol. Being asynchronous, no external clock is required for synchronization, but communicating parties must agree on the speed of the communication, the so called baud rate. The most common baud rate values are 9600, 19200, 38400, 57600 and 115200 bps.

A hardware UART port has at least four pins: voltage (Vcc), Ground (Gnd), Transmit (Tx), and Receive (Rx). The Tx pin is used to transmit data from the device to another connected device, while the Rx pin is used to receive data from the other device. The communication is usually full duplex, meaning that both parties can transmit bits at the same time.

In IoT devices, the UART protocol is used to display debug information, or to configure or repair the device. For instance, if the device has a software malfunction and its web interface is unavailable, one approach to fix it is to make a wired connection to the device through its UART port. From the hacking point of view, UART can be used to collect information about the device's bootloader, operating system, and configuration. The steps to connect to an IoT device are the following:

1e.g., fccid.gov or fccid.io

2e.g., datasheets.com, arrow.com, datasheetcatalog.com, alldatasheet.com, microchip.com

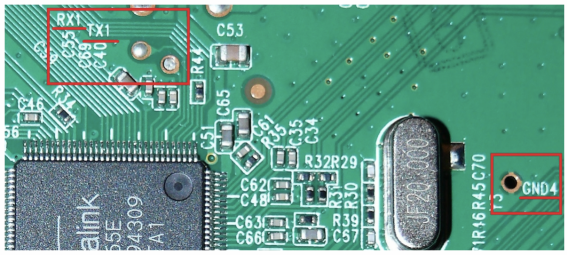

Fig. 1. UART ports on the TP-Link W8951ND router

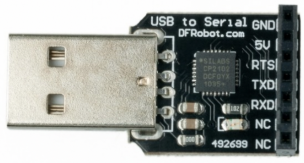

Fig. 2. An UART-to-USB converter device

- **Identifying UART ports and pinouts:** After removing the cover from the device, potential UART ports must be identified. The pins might be explicitly labeled on the motherboard or the four UART signals can be matched to the pins. In order to identify the pins, the board can be analyzed visually, or by using a multimeter or a logic analyzer. Figure 1 shows part of the motherboard of the TP-Link TD-W8951ND router where the UART pins are visible.
- **Connecting the UART pins to a computer:** After having identified the UART pins, the device has to be connected to a computer. For this step, special hardware is needed which can translate between USB and UART. These devices are usually called USB-to-TTL or UART-to-USB devices. An example is shown in Figure 2.
- **Identifying the baud rate:** In order to communicate with the device, the correct baud rate has to be identified.This can be done by trying the most common values manually. Also, there are open source scripts available for this purpose, such as `baudrate.py`[3].
- **Interacting with the device:** Besides the baud rate, the data frame configuration of the IoT device is also needed for proper communication. That can be determined in three ways: the vendor may have described it in the product manual, it might have been posted on a forum on the web, or it can be determined by trying the common frame configurations exhaustively. Once everything has been set, one can communicate with the device via UART by using off-the-shelf programs such as: `minicom`[4], `screen`[5], `dterm`[6], `picocom`[7], or `serialclient`[8]. To interact with a serial port, root privileges are required.

[3] https://github.com/devttys0/baudrate
[4] https://help.ubuntu.com/community/Minicom
[5] https://www.gnu.org/software/screen/manual/screen.html
[6] http://www.knossos.net.nz/resources/free-software/dterm/
[7] https://github.com/npat-efault/picocom
[8] https://github.com/flagos/serialclient

After a successful connection, some devices may require login credentials. Common username/password combinations can be tried to gain access to the device nevertheless. In other cases, UART connection to the device gives access to the boot-loader, a command line interface (CLI) or a shell. However, the received shell may be non-interactive, nevertheless, useful information can be gathered about the device.

B. The SPI protocol

SPI (Serial Peripheral Interface) is a synchronous serial communication bus protocol for short distance communication. SPI operates in a one-master-many-slaves setting, where one master (usually the CPU) controls a Slave Select (SS) wire for each slave. The master initiates communication with a slave by pulling down its SS wire. Also, the master is responsible for generating the clock signal. Like UART, SPI is also a full duplex protocol. Even when one party has no output to send, dummy data is sent on the affected line.

The communication takes place on four lines:

- Serial Clock (SCLK): The clock signal coming from the master. The clock speed must not exceed the maximum guaranteed clock speed of the selected slave.
- Master-Out-Slave-In (MOSI), sometimes Data In (DI): Communication line for sending data from the master to the selected slaves.
- Master-In-Slave-Out (MISO), sometimes Data Out (DO): Communication line for sending data from the selected slave to the master.
- Slave Select (SS): Signals to the slave that the master has initiated communication with it.

From the hacker's point of view, the SPI protocol is usually used to dump the content of an EEPROM or a Flash Memory, which typically implement the SPI protocol and store programs or persistent data.

Exploitation of SPI has similar steps to those of UART exploitation: Firstly, the chip of interest has to be identified and its pins must be matched to the lines described above. Then, the chip has to be connected to a computer, which can be done either with or without desoldering it. Communicating with the chip without desoldering is made possible by special clips such as the one shown in Figure 3. One challenge is that communication with the chip requires it to be powered up. Powering up the entire device is an option, but there can be interference on the chip's legs whenever the CPU tries to communicate with it. If the data sheet specifies the exact voltage level on which the chip should be used, the specified power can be directly applied to the chip from a DC power supply. If the legs are unreachable, then desoldering the chip and soldering it to an SPI Flash or EEPROM adapter is the only option left.

Communicating with the chip using the SPI protocol needs an SPI-to-USB adapter or bridge, such as Bus Pirate[9], a multifunction tool capable of UART, SPI, I2C and JTAG com-

[9] https://www.sparkfun.com/products/12942

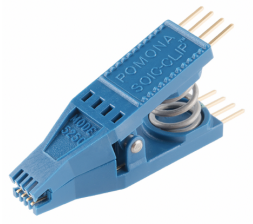

Fig. 3. SPI test clip

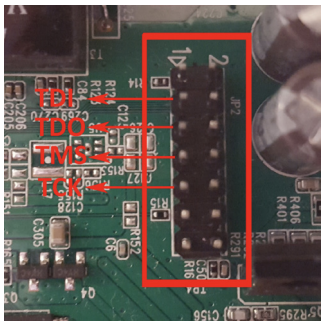

Fig. 4. JTAG interface

munications. Special software is also needed on the connected computer, such as `SPIFlash`[10], or `flashrom`[11].

C. The JTAG interface

JTAG (named after the Joint Test Action Group which specified it) is an industry standard for verifying designs and testing printed circuit boards after manufacture. Essentially, JTAG specifies the use of a dedicated port that implements a serial communications interface for accessing different signals on the board without requiring direct access to the system address and data buses.

JTAG has other higher level usage, namely debugging, which makes it possible to set breakpoints, view register and memory content, and dump the firmware. The beginning of the workflow to exploit a device through JTAG is quite similar to that of UART and SPI: identifying the JTAG pins (see Figure 4 for an example), connecting the device to a computer through JTAG and an adapter device like Bus Pirate mentioned above, and interacting with the device. Interaction is handled on the connected computer by an appropriate tool, such as OpenOCD[12].

OpenOCD uses special configuration files to communicate with the devices. There are many build-in configuration files, but new configurations can be created as well. However, this requires special knowledge about the device, such as its CPU architecture, endianness, TAP (Test Access Port) controller configuration, clock speed, *etc*. After finding or creating the configuration files, and connecting the device and the computer, OpenOCD accepts `telnet` connections at port 4444 and `gdb` connections at port 3333, which can be used to interact with the device.

[10] https://github.com/LowPowerLab/SPIFlash
[11] https://www.flashrom.org/Flashrom
[12] http://openocd.org

III. FIRMWARE EXTRACTION

The firmware is the low level code running on the IoT device that handles access to its hardware components and peripherals, and provides general services to higher level programs, such as an application. In this paper, we consider the operating system (if there is any) of the device as part of its firmware, which is a quite common approach in the domain of embedded systems.

The firmware usually consists of three main parts:

- **Bootloader:** A piece of low level code that initializes the hardware and loads the main operating system. Basically, it is the first program that is executed after switching a device on or after a reset. The bootloader might execute in two stages: in the first stage, only very basic code runs which loads code for the second stage, loading the operating system. This allows the second stage to be updated, while the first stage remains static. Common bootloaders used on embedded devices include Das U-Boot[13], MCU Boot[14], RedBoot[15], iBoot[16], BareBox[17], Bootbase and CFE[18]. Bootloaders may have vulnerabilities, which might be found by tools such as BootStomp[19], a bootloader bug finder for ARM architectures. Vulnerabilities in a bootloader may be exploited by malware, such as UbootKit [10], with the aim of loading a modified operating system and applications, i.e., to compromise the entire device.
- **Operating system (OS):** The operating system provides an execution environment for applications. The OS kernel is the core component of the operating system, which is loaded and started by the bootloader. There is a wide range of operating systems used in embedded devices, ranging from more complex ones like Linux to less complex ones like eCos. The most common operating systems used by IoT devices are Linux[20], VxWorks[21], eCos[22], OpenWRT[23], Junos OS[24] and uCOS[25]. Like the bootloader, the operating system might also contain security holes, but finding these are not trivial either. We discuss some of the approaches later in Section V.
- **File system:** The file system contains configuration files, libraries, development environments, and application programs run by the device. Many IoT devices ship with web servers on them, allowing for web based remote configuration of the device. Such applications are of particular

[13] https://www.denx.de/wiki/U-Boot
[14] https://github.com/runtimeco/mcuboot
[15] https://sourceware.org/redboot/
[16] https://www.theiphonewiki.com/wiki/IBoot_(Bootloader)
[17] https://www.barebox.org/
[18] https://en.wikipedia.org/wiki/Common_Firmware_Environment
[19] https://github.com/ucsb-seclab/BootStomp
[20] https://www.elinux.org/Main_Page
[21] https://www.windriver.com/products/vxworks/
[22] https://www.ecoscentric.com/ecos/index.shtml
[23] https://openwrt.org/
[24] https://www.juniper.net/us/en/products-services/nos/junos/
[25] https://www.micrium.com/rtos/

interest to hackers, because finding vulnerabilities in them does not require special embedded systems background. There are many different file systems for embedded devices including SquashFS[26], UBIFS[27], YAFFS2[28], and JFFS2[29].

A. Obtaining the Firmware

The firmware image of the IoT device can sometimes be found on the vendor's support page, although the image is often only partial. The complete firmware contains the entire file system, whereas a partial firmware image only contains some part of it (typically updated binaries or configuration files). However, even a partial firmware can reveal potential security flaws in older devices, because it usually contains updates that fix security holes. By comparing the updated files with their old versions, the vulnerability fixed in the update can be identified.

Even if the firmware image cannot be obtained from the vendor's support page, it may have already been made available on the Internet by other parties. However, firmware images obtained in this manner should be handled cautiously; they might be modified or their version might be different from the one on the device. The process is also time consuming, but in case of success, the exact binary that is present on the device can be obtained, potentially including the bootloader and the OS kernel. If the firmware image cannot be found on the Internet, it can be dumped from the device using the serial communication protocols presented in Section II.

Some devices have over-the-air (OTA) firmware update functionality, which can be initiated manually or automatically. During the update, a new (partial) firmware image is downloaded from the Internet, and hence, it can be captured with sniffing or man-in-the-middle techniques.

Finally, the simplest IoT devices like smart plugs, smart light bulbs, and smart locks usually come with mobile application used to manage them. Often such a mobile application contains a URL where the original firmware or firmware update can be downloaded from, but which is not indexed by search engines. Reverse engineering the mobile application can provide the hacker with that URL.

B. Unpacking the firmware

The complete firmware image is usually packed into a single compressed or archived file with the file system and the OS kernel. The file also contains a license file or user manual and a binary file. This binary file contains the firmware image, and is sometimes encrypted. The files packed within the binary may be further compressed or archived individually. In addition, the file system component can be stored in a special format. All in all, unpacking the firmware image usually requires to deal with encryption, compression and archive formats, and file system formats. The *de facto* standard tool used for unpacking is called `binwalk`[30], an advanced pattern matching tool capable of analyzing and extracting the content of a firmware image for a large number of different formats and encodings.

1) Dealing with encryption: Dealing with encryption is a challenge. The encryption algorithm and the entire encryption process might not be well-documented, and use proprietary methods. Even if a standard encryption algorithm, such as AES, is used, the keys are usually not readily available. The keys may be stored in tamper resistant hardware on the device, in which case, decrypting the firmware is near impossible. However, if the keys are stored in regular persistent memory which is not tamper resistant, then they can be extracted and the firmware can be decrypted.

To figure out whether the firmware is encrypted or not, entropy based analysis can be used, which is supported by `binwalk`. For an encrypted image, the entropy is flat across the entire binary and its value is close to 1. For a non-encrypted image, the entropy is not flat, its value is usually lower than 1, and it contains fluctuations across the entire file (i.e., there are sections with very low entropy values).

2) Dealing with compression: The different parts of the firmware are usually compressed or archived. Compression is used to save storage space, while archiving creates one single file from several files and directories. Compression and archives can be dealt with in almost the same way. There are many compression methods and archive formats, but `binwalk` can identify many of these methods and formats by searching for their *magic numbers* in the binary.

`binwalk` can find out if the file is compressed or archived even if the algorithm or format is unknown to the program, however, it cannot extract the content. In this case, one can try to find the decompression or extraction code in the non-volatile memory of the device. This, however, is difficult and requires deep technical skills.

3) Interpreting the file system: The file system becomes available after identifying, extracting, and decrypting the firmware image. It defines how files and directories are stored, accessed, and retrieved. A file system is just a binary blob in the firmware image, and its type can be identified based on signatures, just like in case of compressions and archives, but this method is typically more complex and less reliable for file systems. Extracting the file system content requires interpreting the structure, extracting the files, and placing them in the host file system. `binwalk` can identify and unpack many popular file systems, including those discussed at the beginning of this section.

4) What if `binwalk` fails?: It might seem for the reader that `binwalk` can unpack any firmware images. This is indeed true for common Linux-based firmware images in most of the cases. However, in case of special, proprietary firmware formats, `binwalk` may fail, as such formats may not use magic numbers or their extraction methods may be unknown to `binwalk`. However, even in such cases, `binwalk` may output useful information that can give clues regarding where to look for special tools that might work.

[26] http://squashfs.sourceforge.net
[27] http://www.linux-mtd.infradead.org/doc/ubifs.html
[28] https://yaffs.net
[29] http://www.linux-mtd.infradead.org/doc/jffs2.html
[30] http://binwalk.org

IV. BASIC FIRMWARE ANALYSIS

This section covers some basic analysis methods and tools, which can be used mainly on the non-binary parts of the firmware (e.g. text based config files and scripts in the file system). Analyzing the binary content requires advanced methods and tools, which we will discuss in details in Section V.

The main goal of basic firmware analysis is to find hard-coded secrets (e.g., passwords or keys) contained in non-binary files, such as configuration files, password files, and scripts (e.g., shell, Python, JavaScript, and Perl scripts, or alike). More specifically, the potentially collectable information include hard-coded credentials (e.g., username/password), private keys, encryption keys, API (Application Programming Interface) keys, access tokens, authentication cookies, and sensitive URLs or IP addresses. The file system may also store in readable format configuration files, lightweight database files, and password files that may contain useful information. In addition, it is also possible to figure out from the configuration files and scripts what services the device runs (e.g., `telnet`, `ssh`, `ftp`, `http`). Sometimes, basic firmware analysis may also include identifying common configuration errors in configuration files and exploitable programming bugs in scripts.

Useful tools for analyzing non-binary files include the following:

- **grep/egrep:** the *de facto* pattern matching tool on Linux. It can be used to search for strings like 'passwd', 'password', 'telnet', 'ssh', 'secret', *etc.* within all files in the file system.
- **find:** a tool that can find files by their attributes (content, name, permissions, type) with regular expressions.
- **firmwalker**[31]**:** a bash script that searches through the file systems for all the above mentioned keywords (passwords, keys, URLs, *etc.*) using `grep` and `find`, and saves the result in a text file.
- **firmflaws**[32]**:** a standalone Django web server, which uses other basic analysis tools to extract and analyze the contents of a firmware file. It expects a single packed firmware image as input, and it tries to extract its content (with `binwalk`) and analyze it.

A. Example: Basic analysis of the firmware of the D-Link DWR-932 WiFi router (version 4.00b05 Revision D)

For illustration purposes, we present here the result of our basic analysis of the firmware of the D-Link DWR-932 WiFi router.

We ran `firmwalker` on the firmware, which found nearly 1500 files. Some of those files contained interesting strings. We excluded the standard Linux binaries, and the HTML and JavaScript files, and manually analyzed the remaining files.

The file system contained the `/etc/passwd`, the `/etc/shadow` and the `/etc/group` files, which hold information about the users, their passwords, and the groups, respectively, on the system. Checking these files revealed that the default root password was empty:

```
$ cat /etc/shadow
root::17121:0:99999:7:::
...
```

Furthermore, the `/etc/securetty` file, which lists the terminals on which root is allowed to login, contained the serial console `ttyS0`, the USB dongle terminal `ttyUSB0`, and the standard consoles from `tty1` to `tty63`.

The file `/etc/miniupnpd/miniupnpd.conf` contains the default UPnP (Universal Plug and Play) configuration. UPnP was enabled on port 8201, with secure mode off and possible connections from any port and any host:

```
$ cat /etc/miniupnpd/miniupnpd.conf
...
port=8201
...
enable_upnp=yes
...
secure_mode=no
...
allow 0-65535 0.0.0.0/0 0-65535
```

We also found the possible WPA (WiFi Protected Access) passphrase 1234567890 and the possible WPS (WiFi Protected Setup) pin code 12345670 in multiple configuration files.

We identified that Dropbear (a lightweight SSH service) was present in the firmware, however, its automatic start was commented out:

```
$ cat /etc/init.d/dropbear
...
#start-stop-daemon -S \
#  -x "$DAEMON" -- $KEY_ARGS \
#    -p "$DROPBEAR_PORT" $DROPBEAR_EXTRA_ARGS
```

Dnsmasq 2.55 (a lightweight DNS and DHCP server) was also present and started automatically. However, versions lower than 2.78 have serious known vulnerabilities[33], although they can only be exploited when Dnsmasq is configured as a DHCPv6 server, which was not the case on this router.

V. ADVANCED FIRMWARE ANALYSIS

In this section, we give an overview on some advanced techniques used for uncovering vulnerabilities in firmware. As the presented techniques can focus on either the firmware image or the binary executables stored on the filesystem, we will refer to the analyzed piece of code simply as binary code.

Traditionally, analysis techniques can be categorized as either static or dynamic analysis techniques. *Static* techniques interpret instructions of the binary code and perform analysis in an abstract domain. These techniques scale well and can handle large code bases which makes them particularly useful for analyzing whole firmware images. Additionally, they require no test bed or platform. Coupled with the previous advantage, static analysis is a natural choice for performing large-scale analysis of firmware images [11]. However, without runtime information, such techniques often produce false positives, e.g. report vulnerable segments of code which cannot be executed in real life.

[31] https://github.com/craigz28/firmwalker
[32] https://github.com/Ganapati/firmflaws
[33] https://github.com/google/security-research-pocs/tree/master/vulnerabilities/dnsmasq

On the other hand, *dynamic* techniques analyze code as it runs on its intended platform. As a result, these techniques have access to runtime information, which allows for more precise results. However, dynamic techniques cannot provide information on behavior which has not been observed. As a result, these techniques are prone to false negatives, e.g. not all vulnerabilities may be reported. Additionally, analysis requires a test environment, which poses several challenges for IoT devices.

The advantages and disadvantages of both categories are complementary to each other and are often combined to achieve better results. Static analysis techniques are usually performed first, in order to focus dynamic analysis techniques to potentially vulnerable parts of the analyzed piece of code. In return, dynamic techniques can verify the results of static analysis and reduce false positives. As a result, the most advanced analysis techniques in literature cannot be categorized as either static or dynamic analysis, but instead inherit techniques from both categories.

The remainder of this section is structured as follows. We discuss the challenges of analyzing binary instructions in Section V-A and those of test environments for dynamic analysis techniques in Section V-B. Then, we discuss approaches to quickly find potentially vulnerable components in Section V-C and present three advanced analysis techniques: taint analysis in Section V-D, fuzzing in Section V-E and symbolic execution in Section V-F.

A. Challenges of analyzing binary instructions

In order to start analysis, the entry point of the binary has to be determined. This is easy for applications in known file formats (e.g. ELF for Linux-based systems), but challenging for proprietary formats and the firmware image itself. [12] overcame this challenge by analyzing jump tables in the image and starting analysis from multiple potential addresses.

In addition, precise analysis requires context sensitivity, i.e., all call and return sites have to be recovered accurately. While certain architectures have specific instructions for calling and returning from functions, other architectures can achieve the same semantics with indirect jumps. As an example, let us consider the ARM platform, in which the program counter (`pc`) is a general purpose register and the return address is stored in the link register (`lr`). The following (non-exhaustive) list of instructions all result in returns from functions:

```
; Push-pop pair
push lr
pop pc

; Unconditional jump
bx lr ; Used in functions where
      ; lr is not stored on the stack

; Direct program counter manipulation
mov pc, lr

; Bitwise operations
orr r15, r14, r14    ; pc (r15) = lr (r14)
                     ; bitwise-OR lr (r14)
```

While dynamic analysis tools have runtime information available and can accurately compute the call and return addresses, static analysis techniques are hampered in such scenarios. Additionally, there are proprietary architectures in the IoT ecosystem with unknown calling conventions, which makes streamlining tools a challenge [13].

The IoT ecosystem is a heterogeneous ecosystem with many architectures, platforms and firmware. This setting presents several challenges for interpreting binary code and performing static analysis. Firstly, compiler optimization heavily affects the resulting binary code and as a result, the same source code can be compiled into syntactically different, but semantically equivalent binary instructions. Secondly, the different architectures and calling conventions present in the IoT ecosystem make it hard to detect that two sets of instructions compute the same semantic result. Thirdly, depending of the toolchain used to compile a piece of code, the resulting binaries may differ as well.

To overcome these challenges and provide platform indepdence, static analysis techniques are typically not performed on the binary instructions but rather on an *intermediate representation* (IR). The instructions of an IR are often at a higher level than the binary instructions, however, they still lack the same semantic information found in source code. Popular intermediate representations include VEX of valgrind [14], TCG of QEMU [15] and the LLVM bitcode [16].

B. Setting up a test environment

Dynamic analysis techniques require an analysis environment in which the analyzed code can be run. If analysis has access to the underlying hardware or device, those could be used as the environment. However, most platforms do not ship with the tools required to turn the device into a test bed. As a result, significant engineering work is required before analysis can take place. If analysis has no access to the underlying hardware, there are multiple approaches to emulate the platform or certain parts.

Hardware emulation emulates all hardware elements of the underlying platform, including all its peripherals and interrupt handling system. This approach works well for well-understood platforms (e.g. QEMU [15]). However, the platform may be customized without accessible documentation which makes adapting existing emulators near infeasible. Vendors may develop accurate system emulators as part of the development lifecycle to enable firmware developers to work parallel to hardware developers. However, such emulators are usually unavailable to the public and often lack support for code instrumentation necessary for many security analysis techniques.

Even if the hardware is unavailable, the kernel could still be recovered from the firmware image. Emulating the recovered kernel can give more accurate analysis results. However, the kernel may be customized to the platform, hampering generic emulators. [17] overcame this limitation by leveraging the real device to handle I/O operations, signals and interrupts.

If the kernel cannot be recovered, the file system can still be booted with a generic kernel [18], assuming that the original

kernel is based on a generic, available kernel. Applications on the device can be analyzed, but analysis will not yield precise results if they rely on a customized kernel.

If analysis concerns only a single binary application from the file system and the kernel is not customized, then a generic environment can be emulated for the analyzed application, constraining its access to objects present on the original file system. In case of Linux-based IoT platforms, this approach relies on the Linux kernel's ability to call an interpreter to execute an ELF (Executable and Linkable Format) executable for a foreign architecture.

C. Finding potentially vulnerable components

Many vendors in the IoT ecosystem reuse open source components, e.g. the Linux kernel for firmware images, which are customized during the development process [19]. Security vulnerabilities in final products may come from the original code, as was the case with the Heartbleed vulnerability[34], or introduced to the product during development. Either way, the IoT ecosystem is left with potentially tens of thousands devices with similar vulnerabilities. Significant effort has been put into finding similar components based on known vulnerable functions. The main challenge in this area is the sheer number of devices and firmware images to cover.

In order to perform efficient searches, similarity metrics and bug patterns are required. Similarity metrics often include structural features [20], [21], [22], [23] such as the number of instructions, string and numeric constants or the structure of the control flow graph (CFG). However, such metrics face challenges when vulnerable components must be matched in a cross-platform manner. As discussed before, different platforms and toolchains can produce vastly different binary code, even if the original source code is the same.

To handle the heterogeneity of the ecosystem, similarity metrics capturing semantic information are required. Existing approaches include checking input-output pairs computed over higher-level representations, e.g. blocks of IR instructions [24] and conditional formulas [25]. Recently, machine learning algorithms have also been leveraged in order to quickly find code similar to a known vulnerable component [26], [27], [28].

D. Taint analysis

Taint analysis is a technique to detect vulnerabilities resulting from improper data sanitization, i.e. data derived from untrusted input is used in a security sensitive operation. The starting points of the analysis are called *sources* and denote program points where untrusted, user-controlled data can enter the analyzed piece of code, e.g. by reading environmental variables or reading from the standard input. The end points of the analysis are called *sinks* and denote security sensitive operations which can be utilized by attackers to carry out attacks, e.g. jump instructions for circumventing intended control flow. During analysis, the untrustworthiness of data is signaled by *tainting* it and then propagating the taint throughout the code according to a *taint propagation policy*. Vulnerabilities are detected, if a sink performs operations on tainted data. Note, however, that program integrity may have been violated before detection. Taint analysis has been successfully used to identify security-related crashes [29], [30] and recognizing protocol parser code in firmware images [31].

[34]https://www.wired.com/2014/04/heartbleed-embedded/ Last visited: 04.02.2019

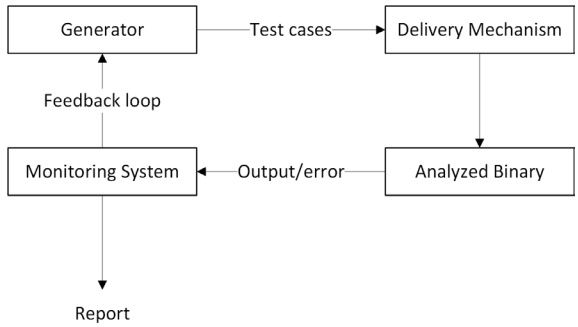

Fig. 5. Main Components of Fuzzing Tools

The technique can be performed in either static or dynamic ways. Static taint analysis [32] considers all possible execution paths starting from sources to sinks but faces the challenge of accurately identifying and analyzing data flows. Challenges arise from indirect memory accesses, indirect calls and pointer aliasing, when the same memory chunck is pointed to by different names. However, if the device cannot be emulated accurately, taint analysis can only be performed in a static manner.

Dynamic taint analysis [33], on the other hand, analyzes a single execution path and as a result, is able to handle scenarios challenging for static variants. However, certain challenges still remain. *Undertainting* is the error arising from the improper handling of certain information flows. Since taint analysis inherently deals with data paths, adding data dependencies to the taint propagation policy is obvious. However, information flow may occur through control dependencies as well, which cannot be computed in pure dynamic analysis as it requires considering multiple execution paths.

Overtainting (also known as *taint spread*), on the other hand, is the error of marking values tainted when they are derived from a taint source. For example, the ARM instruction `eor r0,r1,r1` computes the bitwise exclusive OR on `r1` and itself and then stores the result in `r0`. No matter the value of `r1`, the result will always be `0`. However, if the value of `r1` is tainted and the taint propagation policy does not exclude the example scenario, analysis will incorrectly consider the result tainted as well.

E. Fuzzing

The main idea behind fuzzing [34] is to supply randomly generated input values to the analyzed piece of code and then observe how it reacts. Since the input value is random, there is a high chance that it does not conform with the specification and will trigger anomalies in the code [35], [36], [37].

Figure 5 shows the high-level overview of the main components of fuzzing tools. The *generator* is tasked with generating

the random inputs used during analysis. There are three main types of strategies for input generation:

- **Mutation-based strategy:** Inputs are generated as a mutation of valid initial inputs. Initial inputs have to be specified at the beginning of the fuzzing process. This strategy is easy to set up even without a priori knowledge about the analyzed code, but has a low chance to pass validation checks.
- **Generation-based strategy:** Requires knowledge of program input, usually in the form of a configuration file. Generated random inputs confirm with the configuration file and are able to pass validation checks in programs, reaching deeper code.
- **Evolutionary strategy:** A feedback loop is used to supply the generator with information regarding execution behavior as well as results of other program analysis techniques. This allows for more fine-grained input generation and can greatly increase code coverage.

State-of-the-art fuzzing tools, like VUzzer [38], afl [39], or PULSAR [40], usually deploy evolutionary strategies.

The *delivery mechanism* receives the generated random inputs and supplies it to the analyzed binary. Depending on the input, different types of delivery mechanisms are needed, e.g. messages received over the network have to be delivered to the analyzed binary in a way that is different from the user behavior-based inputs on the embedded web server's graphical interface.

The *monitoring system* plays a crucial role in observing the output of the analyzed binary and detecting faulty behavior. In case of IoT devices, implementing a monitoring system is especially challenging because many traditional signals of faulty behavior are not present on these devices. What is more, the effects of memory corruption are often less visible because the analyzed piece of code may become unresponsive or produce late crashes [41]. There are two main approaches to implementing the monitoring system. Active probing requires special inputs to the code to check liveness. This approach was demonstrated in [42], where heartbeat messages were sent to the analyzed device over UDP. Passive probing, on the other hand, retrieves information about the execution state without alteration.

Fuzzing IoT devices presents unique challenges. In traditional IT settings, many instances of the same software can be started and fuzzed in parallel. For embedded devices, a large number of the same physical device is needed as many of them do not have the necessary memory and computational power, resulting in increased costs. Emulating the device could be a solution, however, there could be infrastructural limits to the number of devices that can be emulated in parallel. What is more, after a bug is triggered in the device, a clean state has to be restored, which often means a full reboot, slowing the process down.

Additionally, many tools require source code instrumentation to implement the feedback loop or the monitoring system. In the IoT ecosystem, the source code is often not available. Even when it is, a comprehensive toolchain would be required to recompile it into binary code. As a solution, dynamic binary instrumentation was proposed and is implemented in many tools, e.g. valgrind [14], Pin [43] or DynamoRIO [44].

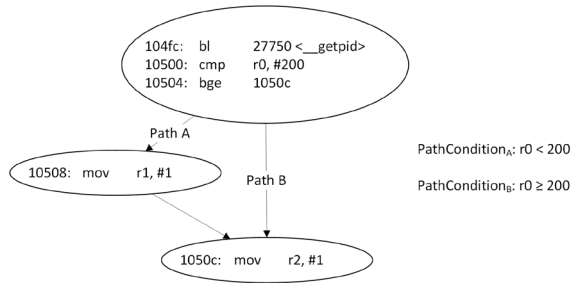

Fig. 6. Example ARM Instructions For Demonstrating Symbolic Execution

F. Symbolic execution

Symbolic execution is an emerging technique for finding vulnerabilities in IoT firmware images and applications [45]. This technique uses special symbols, *symbolic variables*, as values instead of concrete values to explore execution paths. Throughout this section, we demonstrate symbolic execution through the example ARM instructions in Figure 6. The code snippet calls `getpid()` and executes the instruction at 0x10508 only if the process ID is less than 200.

The first step of the analysis is the introduction of symbolic variables. Initially, symbolic variables are unconstrained representing the fact that a certain register or memory location may contain anything. In our example, consider the return value of `getpid()` an unconstrained symbolic variable.

The potential values of symbolic variables are refined at instruction which results in a control flow transitions. If multiple addresses can be followed, execution splits into multiple instances (*forks*). Each instance follows a potential control flow transition and places constraints upon the symbolic variables. The constraints represent the fact that the actual value held in the register or memory location had to satisfy the condition encoded into the branch. In our example, two execution paths are possible. On Path A, the constraint added to the path condition tells that the symbolic variable held in `r0` has to be less than 200. On Path B, the added constraint is for the symbolic variable to be greater or equal to 200. The constraints collected on an execution path are collectively referred to as the *path condition*. Note, that the path condition is taken into consideration at forks as previously added constraints may limit the available execution paths.

When an execution path terminates, the path condition can be solved by a Satisfiability Modulo Theory solver to acquire concrete values for the symbolic variables. The concrete values can then be used as test cases: assuming deterministic code, real-life execution of the analyzed piece of code will follow the same execution path as symbolic analysis did. In our example, for Path A, the solver could return any number below 200, e.g. 100. For Path B, it will return a value greater or equal to 200, e.g. 200. These concrete values can then be used to construct concrete test cases for both paths, maximizing code coverage automatically.

The concepts of symbolic analysis present multiple challenges for its real-life applications. As the subject has been discussed in multiple surveys [33], [46], we only give short descriptions of certain challenges as they are encountered during binary analysis.

Firstly, as analysis spawns two instances at each branch, the number of execution paths available for analysis grows exponentially, presenting serious *scalability issues*. There are many program constructs frequently used which result in exponential growth in the number of paths: symbolic loop guards, symbolic indices, etc. For binary code, symbolic offsets in memory and symbolic jump addresses can further complicate analysis. There two existing approaches to mitigate the issue:

- **Mixed concrete and symbolic execution**: The analyzed code is segmented into two parts: interesting instructions are analyzed over the symbolic domain, while uninteresting instructions are analyzed as if they were executed by the CPU. Segmentation can be determined by the tool: before an instruction is analyzed, the engine can check whether any of the operands is symbolic. If there is such an operand, the instruction is analyzed over the symbolic domain, otherwise it is analyzed over the concrete domain. However, given the complexity of some firmware images, the search space still remains too large for the mixed approach. In such cases, program slices can be computed over the firmware image to limit the scope of the analysis [12].
- **Path selection**: Unless the number of symbolic variables is kept at a minimum, the symbolic domain of the mixed approach may still remain too large to cover. As tools cannot hope to explore all execution paths, certain execution paths must be prioritized or abandoned according to some criterion. There have been numerous proposed criteria [47] for different application domains, but no universally effective method has been proposed yet.

Secondly, symbolic analysis engines have to model the execution environment of the analyzed code. In case of binary code, the engine has to posses knowledge about the potential registers a given platform can use, it has to model the memory and it must also be able to model the side effects certain instructions have (e.g. by setting flags) as well as interrupts [47] and hardware interactions [48]. In order to achieve platform independence, tools can leverage the intermediate representations discussed in Section V-A.

Finally, symbolic execution can only reason about code it analyzes, it cannot reason about unseen code. This challenge arises when specific binary applications are analyzed separated from the firmware image's filesystem and/or kernel and it is known as the *environment problem*. Unseen code (e.g. library functions, system calls) can have significant side effects on the analyzed program, which must be taken into consideration for precise analysis. One widely used solution is to create summary functions for such code to model its side effects. Several symbolic analysis tools (e.g. KLEE [49], EXE [50], angr [51]) implement this approach.

VI. CONCLUSION

In this paper, we gave a basic introduction into hacking IoT devices. We first introduced some details on the interfaces and the protocols at the hardware level that can be useful in a penetration testing context, and we explained how these interfaces can be identified in the device and how the protocols can be used for interacting with the device. Next, we summarized the methods and tools for extracting the firmware of the device and unpacking it for further analysis. We also gave an overview on some basic firmware analysis methods and tools that can be used to find hard-coded passwords and keys, to identify erroneous configuration settings, and to find simple bugs in scripts. Finally, we dealt with some more advanced analysis methods that can be used to discover vulnerabilities in the binary programs that belong to the firmware. Binary program analysis is still an active area of research, so we surveyed the most relevant scientific publications in the domain, including papers on static and dynamic analysis of binaries, taint analysis techniques, fuzzing, and symbolic execution of programs.

We deliberately restricted ourselves to hardware hacking and the analysis of the device's firmware, as vulnerabilities in the firmware can lead to full adversarial control over the device. We note, however, that penetration testing can be extended to the wireless interfaces of and protocols used by the device, to the applications running on the device, including web servers and remote access tools, to the mobile application that may be provided to remotely configure and control the device, and to the cloud end-points that the device may connect and send data to.

Ethical hacking of IoT devices is fun and useful at the same time. However, it is a relatively new area of research that still needs to mature. We expect that similarly to the best practice guides and standards for penetration testing of networks and web based applications, best practices and standards for IoT hacking will emerge and evolve in the near future. In particular, standards for security testing industrial IoT systems as well as autonomous and connected vehicles are needed in order to integrate the security testing activity into the development life cycle of those systems, and hence, to increase trust in them. In the home IoT area, best practices for security testing can encourage vendors to pay more attention to security, and ultimately, raise the bar for attackers to a level that is acceptable by home users.

ACKNOWLEDGEMENT

The work of Dorottya Papp and Levente Buttyán has been supported by the SETIT Project[35] (Security Enhancing Technologies for the Internet of Things).

REFERENCES

[1] N. Falliere, L. O'Murchu, and E. Chien,"W32.Stuxnet dossier,"Symantec Technical Report version 1.4., 2011.
[2] R. Langner, "To kill a centrifuge – a technical analysis of what Stuxnet creators tried to achieve," online: https://www.langner.com/wp-content/uploads/2017/03/to-kill-a-centrifuge.pdf, 2013.

[35]Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

[3] A. Greenberg, "Hackers remotely kill a Jeep on the highway – with me in it," Wired, July 2015.
[4] ——, "Crash Override: the malware that took down a power grid," *Wired*, June 2017.
[5] K. Zetter, "A cyberattack has caused confirmed physical damage for the second time ever," *Wired*, January 2015.
[6] L. H. Newman, "A new pacemaker hack puts malware directly on the device," *Wired*, August 2018.
[7] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai botnet," in *Proceedings of the Usenix Security Symposium*, 2017.
[8] G. M. Graff, "How a dorm room minecraft scam brought down the internet," *Wired*, December 2017.
[9] N. Zhang, S. Demetriou, X. Mi, W. Diao, K. Yuan, P. Zong, F. Qian, X. Wang, K. Chen, Y. Tian, C. A. Gunter, K. Zhang, P. Tague, and Y.-H. Lin, "Understanding IoT security through the data crystal ball: Where we are now and where we are going to be," online: https://arxiv.org/pdf/1703.09809.pdf, March 2017.
[10] J. Yang, C. Geng, B. Wnag, Z. Liu, C. Li, J. Gao, G. Liu, and W. Yang, "UbootKit: a worm attack for the bootloader of IoT devices," in *BlackHat Asia Conference*, 2018.
[11] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis, "A large-scale analysis of the security of embedded firmwares." in *USENIX Security Symposium*, 2014, pp. 95–110.
[12] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware." in *Network and Distributed Systems Security Symposium (NDSS)*, 2015.
[13] M. C. Ang Cui and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *Network and Distributed System Security Symposium (NDSS)*, 2013.
[14] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746
[15] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401
[16] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the llvm intermediate representation for verified program transformations," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 427–440. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103709
[17] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti et al., "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares." in *Network and Distributed Systems Security Symposium (NDSS)*, 2014.
[18] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for linux-based embedded firmware." in *Network and Distributed Systems Security Symposium (NDSS)*, 2016.
[19] M. Liu, Y. Zhang, J. Li, J. Shu, and D. Gu, "Security analysis of vendor customized code in firmware of embedded device," in *Security and Privacy in Communication Networks*, R. Deng, J. Weng, K. Ren, and V. Yegneswaran, Eds. Cham: Springer International Publishing, 2017, pp. 722–739.
[20] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *Network and Distributed Systems Security Symposium (NDSS)*, 2016.
[21] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 480–491. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978370
[22] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Giuffrida, S. Bardin, and G. Blanc, Eds. Cham: Springer International Publishing, 2018, pp. 114–138.
[23] H. Lin, D. Zhao, L. Ran, M. Han, J. Tian, J. Xiang, X. Ma, and Y. Zhong, "Cvssa: Cross-architecture vulnerability search in firmware based on support vector machine and attributed control flow graph," in *2017 International Conference on Dependable Systems and Their Applications (DSA)*, Oct 2017, pp. 35–41.
[24] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 709–724.
[25] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin, "Extracting conditional formulas for cross-platform bug search," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017, pp. 346–359. [Online]. Available: http://doi.acm.org/10.1145/3052973.3052995
[26] Y. Li, W. Xu, Y. Tang, X. Mi, and B. Wang, "Semhunt: Identifying vulnerability type with double validation in binary code." in *29th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2017, pp. 491–494.
[27] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 896–899. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240480
[28] J. Gao, X. Yang, Y. Fu, Y. Jiang, H. Shi, and J. Sun, "Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 803–808. [Online]. Available: http://doi.acm.org/10.1145/3236024.3275524
[29] K. Eom, J. Paik, S. Mok, H. Jeon, E. Cho, D. Kim, and J. Ryu, "Automated crash filtering for arm binary programs," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, July 2015, pp. 478–483.
[30] H. Jeon, S. Mok, and E. Cho, "Automated crash filtering using inter-procedural static analysis for binary codes," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, July 2017, pp. 614–623.
[31] Y. Zheng, K. Cheng, Z. Li, S. Pan, H. Zhu, and L. Sun, "A lightweight method for accelerating discovery of taint-style vulnerabilities in embedded systems," in *Information and Communications Security*, K.-Y. Lam, C.-H. Chi, and S. Qing, Eds. Cham: Springer International Publishing, 2016, pp. 27–36.
[32] K. Cheng, Q. Li, L. Wang, Q. Chen, Y. Zheng, L. Sun, and Z. Liang, "Dtaint: Detecting the taint-style vulnerability in embedded device firmware," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2018, pp. 430–441.
[33] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.
[34] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, no. 1, p. 6, Jun 2018. [Online]. Available: https://doi.org/10.1186/s42400-018-0002-y
[35] Z. Wang, Y. Zhang, and Q. Liu, "Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing," *KSII Transactions on Internet and Information Systems*, vol. 7, no. 8, pp. 1989–2009, 2013.
[36] W. Frisby, B. Moench, B. Recht, and T. Ristenpart, "Security analysis of smartphone point-of-sale systems," in P*roceedings of the 6th USENIX Conference on Offensive Technologies*, ser. WOOT'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2372399.2372403
[37] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 447–462.
[38] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Network and Dis- tributed System Security Symposium (NDSS)*, 2017.
[39] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, last visited: Feb 7, 2019.
[40] H. Gascon, C. Wressnegger, F. Yamaguchi, D. Arp, and K. Rieck, "Pulsar: Stateful black-box fuzzing of proprietary network protocols," in *Security and Privacy in Communication Networks*, B. Thuraisingham, X. Wang, and V. Yegneswaran, Eds. Cham: Springer International Publishing, 2015, pp. 330–347.

[41] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[42] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[43] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[44] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 265–275. [Online]. Available: http://dl.acm.org/citation.cfm?id=776261.776290

[45] I. Pustogarov, T. Ristenpart, and V. Shmatikov, "Using program analysis to synthesize sensor spoofing attacks," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017, pp. 757–770. [Online]. Available: http://doi.acm.org/10.1145/3052973.3053038

[46] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 317–331.

[47] D. Davidson, B. Moench, S. Jha, and T. Ristenpart, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in P*roceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 463–478. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534806

[48] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: system-wide security testing of real-world embedded systems software," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 309–326.

[49] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[50] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1–10:38, Dec. 2008. [Online]. Available: http://doi.acm.org/10.1145/1455518.1455522

[51] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 138–157.

Dorottya Papp received the MSc degree in Computer Science in 2016 from the Budapest University of Technology and Economics (BME) and she is currently pursuing the Ph.D. degree under the supervision of Dr. Levente Buttyán. She joined the Laboratory of Cryptography and System Security (CrySyS Lab) during her undergraduate studies at BME. Since 2015, she has also taken part in research projects at the Austrian Institute of Technology (AIT). Her research interests include security assurance, program analysis and security testing of binary programs. She is involved in research projects in the domain of embedded systems and the Internet of Things.

Kristóf Tamás received the B.Sc. and M.Sc. degrees (Diploma with Honours) in Computer Science from the Budapest University of Technology and Economics (BME) in 2017 and 2019, respectively. As a student, he has done research on identifying malicious web pages with static analysis and on penetration testing the Internet of Things under the supervision of Dr. Levente Buttyán. After graduation, he remained an associate member of the Laboratory of Cryptography and Systems Security (CrySyS Lab) and a member of the Lab's talent management program. He also has a leading role in the university's Capture-The-Flag (CTF) team, called c0r3dump. Kristóf Tamás currently works at Ukatemi Technologies Kft. - a CrySyS Lab spin-off company - as a junior security engineer and penetration tester.

Levente Buttyán received the M.Sc. degree in Computer Science from the Budapest University of Technology and Economics (BME) in 1995, and earned the Ph.D. degree from the Swiss Federal Institute of Technology - Lausanne (EPFL) in 2002. In 2003, he joined the Department of Networked Systems and Services at BME, where he currently holds a position as an Associate Professor and leads the Laboratory of Cryptography and Systems Security (CrySyS Lab). He has done research on the design and analysis of secure protocols and privacy enhancing mechanisms for wireless networked embedded systems (including wireless sensor networks, mesh networks, vehicular communications, and RFID systems). He was also involved in the analysis of some high profile targeted malware, such as Duqu, Flame, MiniDuke, and TeamSpy. His current research interest is in security of cyber-physical systems (including industrial automation and conrtrol systems, modern vehicles, cooperative intelligent transport systems, and the Internet of Things in general). Levente Buttyán played instrumental roles in various national and international research projects, published 150+ refereed journal articles and conference/workshop papers, and co-authored multiple books and patents. Besides research, he teaches courses on applied cryptography and IT security at BME and at the Aquincum Institute of Technology (AIT Budapest), and he leads a talent management program in IT security in the CrySyS Lab. He also co-founded multiple spin-off companies, notably Tresorit, Ukatemi Technologies, and Avatao.