JOHANNES KEPLER
UNIVERSITY LINZ

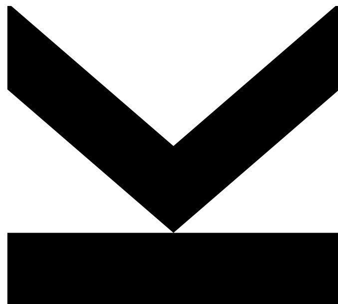
Submitted by
Tobias Höller

Submitted at
Institute of Networks and Security

Supervisor
Univ.-Prof. Priv.-Doz. DI Dr. Rene Mayrhofer

August 2017

Automatic Updates for IoT devices exemplified by OpenWRT



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

The era of the Internet of Things is on the horizon and the security of these devices is a problem of ever growing importance. A key piece to solving this issue is an update process that allows vendors to automatically update their IoT devices. Different update implementations were analysed to find the most promising approach to take towards that goal. The suggested solution is then also implemented as a proof-of-concept on a router running OpenWRT. This thesis is designed as a roadmap to help developers, who are trying to bring automatic updates to their device, to find the best update procedure for their situation and implement it on their specific platform.

Kurzfassung

Durch den aktuellen Siegeszug des ”Internet of Things”, wird die Sicherheit von entsprechenden Geräten zu einem immer größeren Problem. Die Frage wie man solche Geräte zuverlässig mit Aktualisierungen versorgen kann, ist eines der wesentlichen Probleme, das in diesem Zusammenhang gelöst werden muss. Unterschiedliche Implementierungen von Updates wurden untersucht, um einen möglichst vielversprechenden Ansatz für IoT Geräte zu finden. Die vorgeschlagene Lösung wird darüber hinaus testweise auf einem OpenWRT Router implementiert. Diese Arbeit soll IoT Entwicklern, die ihre Geräte mit automatischen Updates versorgen wollen, als Hilfestellung dienen, den für sie besten Lösungsansatz zu finden und umzusetzen.

Contents

Abstract iii

1. Introduction 1
- 1.1. Why autoupdates ? . . . 1
  - 1.1.1. Case Study: The Mirai Botnet . . . 2
  - 1.1.2. Case Study: The attack on Telekom routers . . . 2
- 1.2. Why are there no automatic updates yet ? . . . 4

2. Technical Background 7
- 2.1. Introduction to OpenWRT . . . 7
  - 2.1.1. Memory Layout of OpenWRT . . . 8
- 2.2. Introduction to U-Boot . . . 9
  - 2.2.1. Boot process . . . 10
  - 2.2.2. Other U-Boot features . . . 11
- 2.3. Automatic Updates . . . 12
  - 2.3.1. Update trigger . . . 13
  - 2.3.2. Update mechanism . . . 13
  - 2.3.3. Recovery scheme . . . 14
  - 2.3.4. Requirements . . . 16

3. State of the Art 19
- 3.1. Common update processes . . . 19
  - 3.1.1. Windows Update . . . 19
  - 3.1.2. Linux package managers . . . 20
  - 3.1.3. Android Updates . . . 23
  - 3.1.4. Android App Store . . . 26
- 3.2. Update processes for OpenWRT . . . 27
  - 3.2.1. Manual Update . . . 27
  - 3.2.2. Gluon Updater . . . 29

3.2.3. Turris Omnia Updater . . . 31
3.2.4. Linksys Autoupdates . . . 34

4. Hardware **37**
4.1. TP Link Archer C7 . . . 37
4.1.1. Hardware Overview . . . 37
4.1.2. Observations . . . 38
4.2. Zyxel NBG6716 . . . 39
4.2.1. Hardware Overview . . . 39
4.2.2. Observations . . . 40
4.2.3. Partition Layout . . . 41
4.3. Linksys WRT1200AC . . . 42
4.3.1. Hardware Overview . . . 42
4.3.2. Observations . . . 43

5. Strategy selection **45**
5.1. Desired Update process . . . 45
5.1.1. Update trigger . . . 45
5.1.2. Update mechanism . . . 46
5.1.3. Recovery scheme . . . 49

6. PoC on Zyxel NBG6716 **51**
6.1. Sysupgrade Script . . . 51
6.2. Unlock U-Boot . . . 56
6.2.1. Gain Write Permissions on U-Boot Environment . . . 56
6.2.2. Work around zloader . . . 58
6.3. Enable Dual-Boot . . . 60
6.3.1. Investigate Linksys WRT1200AC Setup . . . 60
6.3.2. Implementing Dual Boot on the Zyxel NBG6716 . . . 61
6.4. Enable Recovery . . . 63
6.4.1. Switch to alternative firmware at next boot . . . 64
6.4.2. Only switch to alternative firmware if there is an error . . . 65
6.4.3. Restore U-Boot environment . . . 67
6.5. Securing the update process . . . 68
6.6. Summary . . . 68

7. Future work **71**
7.1. Prepare build environment 71
7.2. Test produced images 72
7.3. Prevent user disruption 72
7.4. Create minimal recovery partition 73
7.5. Track necessary recoveries 74
7.6. Bring update procedure to more devices 74

8. Conclusion **77**

Bibliography **79**

A. U-Boot Environments **81**

B. Relevant OpenWRT Scripts **87**

Specification

The goal of this thesis is to find a way to automatically update OpenWRT routers in a way that is robust and can be applied in many scenarios. The solution should not require major modifications of the OpenWRT source code and remain compatible with the development principles of the project.

1. Introduction

1.1. Why autoupdates ?

Usually software is updated to either fix problems in the current version or provide additional features. Embedded devices generally have a very limited software to support their function and almost never need additional features. So would it not be easier and cheaper to test the software thoroughly to detect and resolve errors before they inconvenience users ?

One of the most important things about software development is, that there is nothing like a perfect program. While a software can in theory mathematically be proven to be free of bugs, in practice such a proof is almost impossible to obtain, unless the program is simplistic or the invested resources are enormous. Fortunately the tasks for software in embedded devices are often limited to simple applications. That along with the fact that it is easy to predict how users will use the device makes it easier to test all scenarios relevant to potential customers. Therefore one could argue that most embedded devices do not need automatic updates, because they do not need updates at all.

But the current trend to add networking capabilities to these devices changes the game completely. Devices with a full network stack require more complex operating systems and have to rely on a multitude of libraries. All of these components are possible error sources and the networking interface makes it easier for attackers to detect and exploit existing vulnerabilities. Devices using an internet connection are often referred to as IoT (Internet of Things) devices and are getting more and more popular. Unfortunately may manufacturers and vendors of such devices do not believe that they are responsible for the security of their devices, which is turning the vision of an internet of things more and more into the security nightmare of an internet of insecure things.

1.1.1. Case Study: The Mirai Botnet

The distributed denial-of-service (DDOS) attacks performed by botnets consisting mainly of IoT devices in late 2016 (while this thesis was being worked on) show what happens when devices are connected to the internet without any regard for security. The Mirai malware responsible for accumulating hundreds of thousands of IoT devices into botnets does not even require any complex exploits, it works by simply searching the Internet for possible targets (mainly IP surveillance cameras, routers and DVRs) and tries a list of default passwords[3].

The developers of these devices made a simple mistake by forgetting that users do not change default passwords unless they have to. And from their perspective this does make sense, because they will rarely need the password so keeping the one already printed on the device is much safer than changing it to a new one that is very likely to be forgotten. This simple oversight enabled Mirai to create a botnet which launched the most powerful DDOS attack in history in November of 2016 [12]. The effects may have been local and concentrated to the US, but rendering sites like Amazon, PayPal, Netflix or Twitter unavailable is no small feat for a single attack. Considering the fact that the number of IoT devices connected to the internet has been climbing continuously for the last few years, their lacking security may very well become a threat to the infrastructure of the Internet as we know it today.

1.1.2. Case Study: The attack on Telekom routers

Another just as interesting security incident took place in November 2016, which temporarily disabled more than 900 thousand routers operated by the "Deutsche Telekom" in Germany. The attack was noticed when numerous customers complained because their internet connection was not working. It turned out that an attack was going on that was using parts of the source code of the Mirai malware but instead of guessing passwords, it used a known vulnerability for a remote management protocol known as CPE WAN Management Protocol (CWMP). The Telekom managed to stop this attack by modifying their infrastructure to no longer forward any packages destined for port 7547, which is the port the CWMP protocol was listening on. [15] [16]

Thanks to the fact that the person responsible for this attack has been caught we do know a lot about the intentions of the attacker as well as what happened in detail. First of all the attack was not supposed to damage the Deutsche Telekom, as a matter of fact it was not intended to impact users at all, it just tried to add their routers to a botnet without them noticing. It turns out that the vulnerability used for this attack only affected devices based on Linux and the Telekom routers in question were using a custom operating system by the Arcadyan Technology Corporation, so the exploit was not supposed to work on them. But the malware was not configured to check if the target device was running Linux, it simply tried the exploit on every device listening on port 7547 and that caused the problems. Apparently the affected routers do have another error in their software that causes them to slow down and crash if multiple connections to the CWMP port are established. And because a lot of other routers had been attacked successfully and were trying to add new devices to the botnet, there were a lot of active connections to that port on every device. [15] [16]

The first and possibly most frightening lesson to learn from this attack: If it had not disrupted so many Telekom customers by accident, this attack may have gone unnoticed for much longer affecting many more devices. In addition to that we learn that some routers, devices that are definitely supposed to be connected to the internet, are extremely vulnerable to attacks, if scanning the same port multiple times is enough to effectively disable them. And of course we see that even though the vulnerability in the protocol was known and a fix available most vendors were either incapable or unwilling to patch their vulnerable devices. If they had, this attack would have never have had such a massive impact.

Lastly it seems important to point out that this attack was executed by a single individual, who had been contracted to launch a DDOS attack on a company. In order to execute that attack he needed a botnet, so he modified a publicly available version of Mirai to use a recently reported vulnerability in routers based on Linux and let it loose. The effort for this attack was low, the promised payment only 10.000€. A world where every hacker can create his own botnet of IoT devices in a matter of days for any purpose should be enough to raise concern for this topic.

But this case revealed another even more frightening perspective. What would

happen if a group of hackers invested more time and effort into finding ways to permanently disable routers instead of taking control over them. Just imagine if millions of routers worldwide are destroyed within days. Especially in times where more and more devices need to be connected to the internet in order to work properly, the damage and disruption caused by such an attack could be enormous.

1.2. Why are there no automatic updates yet ?

The previous section should have made a convincing argument why an update strategy for routers and IoT devices is important. But devices like routers do already have an update strategy, many vendors regularly release new firmware versions for their devices, which increase the stability of the software and fix known vulnerabilities like the one used in the attack on the Telekom. So why is there still a problem ?

The simple answer is that these updates are only published on the vendors website and nobody checks a website regularly for a new firmware version. And even if users would know about it, the majority of them does not care about security updates, they want their device to work and if it does not, they throw it out and buy another one. So like many other things in security, updates have to happen transparently to the user, otherwise they will not happen at all and the updates published by vendors are merely placebos so they can say they have done everything in their power by publishing a new firmware, even if they know that 99% of their devices will never receive it.

So if updates for routers are already available, why are they not downloaded and installed automatically ? In theory it sounds easy to regularly run a program that checks for new firmware versions and installs them, but in practice there are quite a few pitfalls that need to be considered.

The most likely reason why this does not happen is a simple economic consideration: What can a vendor gain by automatically updating his devices and what does he risk ? If the only danger consists of the device being added to a botnet without interrupting normal operation, most customers will never notice the lack of security. The victims would be the targets attacked by the

botnet, but these are usually companies and no relevant stakeholder group for a router vendor. So providing automatic updates would not provide relevant economic gains, which reasonable explains why companies do not want to put any effort into the topic.

In the wake of the recent attacks through Mirai botnets, some politicians are considering policies to make vendors of IoT devices liable for the damage their devices cause as parts of botnets, if they were not properly secured. Until either such regulations are passed or attackers start disturbing routers normal operation, the motivation to improve the security of these devices will remain low.

On the other hand automatic updates can be the source for a multitude of problems. Most updates require a reboot, which temporarily prevents users from using the device. In addition to that every update carries the risk that something unexpected goes wrong, like a new library no longer supporting an old method, hardware being no longer supported by an updated driver, new bugs introduced through updates, power outages during the update or write errors while writing the update to the flash memory.

This long list of potential problems, which can all translate into economic disadvantages, along with the short list of benefits provided by automatic updates makes it easier to understand why most devices do not receive updates automatically yet. And exactly this is the point this paper is trying to change by decreasing the risks associated with automatic updates.

The goal of this thesis is to come up with an update procedure that includes a recovery scheme, which automatically reverts to a working firmware if the update has caused any kind of trouble. This should severely decrease the risks associated with automatic updates and thereby increase the willingness of vendors to implement them on their devices.

The suggested update procedure will not only be discussed in theory but also implemented on a router as a proof of concept. OpenWRT was chosen as a platform because it is an open source and well established software for routers and routers again are the most established embedded device with networking capabilities. They were around before the term IoT device was invented and are usually not counted as such, but the technical similarities cannot be denied,

especially since OpenWRT has also been used on several other IoT systems like IP cameras.

Hopefully the work done for this thesis will support any developer trying to bring automatic updates to their IoT device and ideally be the basis for a community effort to bring automatic updates to devices running OpenWRT.

2. Technical Background

This chapter will give a quick introduction into the software discussed in this thesis as well as a definition of how "Automatic Updates" and associated terms are defined in the context of this thesis.

2.1. Introduction to OpenWRT

OpenWRT was originally based on a firmware released by Linksys for their WRT54G routers. This firmware contained software licensed under the GPL, which forced Linksys to release the source code for their router. Due to this several developers were able to add and modify features of the original firmware. [11]

About a year later the OpenWRT project was officially founded, with the goal of providing a free Linux distribution for embedded devices in general, but there is a clear focus on single owner/home office devices like the original WRT54G. The development continued over several years bringing multiple new versions of OpenWRT with many new features. By requesting the source code from other vendors, which also used GPL licensed source code in their products, the project was able to improve and extend the number of available features and supported devices. The current table of hardware [1] lists more than 600 supported devices and at the time of writing this paper (March 2017) the included package manager(opkg) served 4438 different packages. Furthermore there are several other projects that either follow the development of OpenWRT or forked form it at some point in time. [13]

These facts along with the fact that several vendors are selling devices which are shipped with more or less customized versions of OpenWRT demonstrate

[1] https://wiki.openwrt.org/toh/start

how successful this open source project has been over the last years. The most important thing to understand about OpenWRT is that it does not try to be the same system for every user. It is much easier to describe it as a skeleton system, which initially only includes the bare minimum necessary for a working operating system. Everything else needed by the user, must be added. This approach allows for maximum versatility and makes it easy for users to optimize a system according to their needs. As a downside such an approach deters beginners and inexperienced users. This is the main reason why several projects are based on OpenWRT but try to make installation and usage easier by equipping their systems with reasonable defaults that meet the requirements of the majority of users as well as improving the graphical user interfaces.

It should be mentioned at this point that during the work on this thesis, a new fork named LEDE (Linux Embedded Development Environment) was created, after the OpenWRT project had seen very little progress for a significant amount of time. This fork gathered a majority of active OpenWRT developers and progressed significantly compared to the original project. In the first months of 2017 however, it was announced that the two projects would merge again, using the LEDE source code for their merged project but keeping the OpenWRT name. This paper will operate under the assumption that the merge is successful under the announced terms and make no further distinctions between LEDE and OpenWRT. [5]

2.1.1. Memory Layout of OpenWRT

To gain a more detailed understanding of the challenge, that updating OpenWRT presents, it is necessary to understand the usual layout of OpenWRT[2]. This layout is designed towards stability, recoverability and memory efficiency:

By taking a closer look at layer 2 of figure 2.1 it becomes clear that OpenWRT depends on a bootloader to be initialized as well as several optional partitions specific to the individual system it is running on. For the scope of this paper these hardware specific partitions will be ignored.

[2] https://wiki.openwrt.org/doc/techref/flash.layout

Figure 2.1.: Default Flash Layout for OpenWrt

<table>
<tr><td>Layer 0</td><td colspan="6">raw flash</td></tr>
<tr><td>Layer 1</td><td rowspan="3">bootloader partition(s)</td><td rowspan="3">optional SoC specific partitions</td><td colspan="3">OpenWrt firmware partition</td><td rowspan="3">optional SoC specific partitions</td></tr>
<tr><td>Layer 2</td><td rowspan="2">Linux Kernel</td><td colspan="2">rootfs
Mounted on “/”, OverlayFS with /overlay</td></tr>
<tr><td>Layer 3</td><td>SquashFS: /dev/root
Mounted on /rom</td><td>JFFS2: rootfs_data
Mounted on /overlay</td></tr>
</table>

The main information to be taken from here is that the firmware for OpenWRT generally consists of three distinct parts: The Linux Kernel, a SquashFS root partition and and a JFFS2 rootfs_data partition. Note that SquashFS is a compressed read-only filesystem designed to make use of every single available bit. Thanks to this highly efficient storage mechanism the entire Firmware can be fit into as little as 4 Megabytes. The Linux Kernel is sometimes included in the SquashFS partition or otherwise written directly to the raw flash in a compressed form. The OverlayFS (which does not necessarily have to be JFFS2, there are also others like UbiFS for example) is used only to store changes made by the user, which are often limited to a few configuration files.

2.2. Introduction to U-Boot

U-Boot is a project to develop a universal bootloader running on multiple different architectures and is commonly used on embedded systems. Its development is closely related to Linux. Not only are some parts of the code taken from the Linux Project, but it is a declared priority of the project to support booting the Linux kernel from U-Boot. The project itself was started in 1999 with the first release of 8xxROM by Magnus Damm and called itself PPCBOOT for the first years. In the years 2000 until 2002 there was extensive development until in 2002 the last version of PPCBoot (2.0.0) was released. At this point the project was renamed into ”Das U-Boot” and has been developed under this name since than. [7]

Just like OpenWRT U-Boot is highly customizable both during compile time and runtime and can therefore easily be adapted to a new hardware platform. This fact in combination with its tight connection to the Linux kernel explains

why many routers make use of U-Boot to start their operating system. So while OpenWRT does not run exclusively on routers using U-Boot, it is a very common combination.

The main reason for the runtime configurability lies in the fact that U-Boot by default splits the system and the configuration into two different partitions. The system partition is usually referred to as ”u-boot” partition, while the configuration partition is called ”u-boot-env” or ”env” being short for U-Boot environment. This environment works like a key-value store, where users can customize the bootloader by adding new key-value pairs, changing existing values or removing a key entirely.

2.2.1. Boot process

The main task of any bootloader is to initialize the hardware, do some preparation work and then trigger the start of the operating system. The initialisation part will be ignored for this work, but the commands needed to boot the Linux kernel as well as the arguments that are passed, will be important. First U-Boot only executes code from RAM so the kernel of the operating system must first be loaded into RAM, before it can be executed. This can be done in many different ways. It is possible to load it from the flash directly using either ”cp” if its NOR flash, or ”nand read” if its NAND flash. There are also options available to load an image from a memory card or via USB, if they use a file format supported by U-Boot. Often supported file systems are FAT32 and EXT2, but specific vendors may choose to include support for alternative file systems. If the filesystem is supported by U-Boot it is also possible to load a file from a file system on the flash chip, instead of a direct access. And lastly for recovery reasons it is also possible to load files from a remote location using TFTP, which is especially handy for recovery purposes. [6]

No matter how the kernel is loaded, once it has been loaded it is necessary to start running that kernel. The best way to do this is the ”bootm” command, which actually triggers a series of commands that verifies and uncompresses the kernel and relocates initrd and the flat devices tree. Optionally U-Boot can be compiled to also execute certain OS specific tasks before continuing execution at the uncompressed kernel using the ”go” command. [6]

Every U-Boot environment contains a key called ”bootcmd” (could of course be customized to have a different name), which holds the command to be executed when trying to boot. During a normal startup U-boot waits for few seconds to give the user a chance to interrupt the boot process, before the startup is continued by executing the contents of the ”bootcmd” value.

2.2.2. Other U-Boot features

U-Boot does provide a few other features relevant for this thesis. Those that are used or mentioned in later chapters shall be introduced at this point.

U-Boot command line

As just mentioned in the previous chapter, U-Boot provides a way to interrupt the boot process, exactly before the contents of the ”bootcmd” key are executed. This interrupt can either be an arbitrary signal via the serial interface of a device, or a specific password. If the interrupt is successful, U-Boot provides a command-line interface to the user (the U-Boot command line).

The description in section 2.2.1 mentioned several commands, all of those can be executed manually on the command line. In addition there are many other commands available mainly to debug errors by obtaining information about the system, recover from a broken operating system or change the U-Boot environment. More detailed information on the available commands can be obtained from the U-Boot documentation.[6]

Pass kernel command line arguments

As already mentioned U-Boot was mainly designed to boot Linux kernels, so it does not come as a surprise that this bootloader also supports passing additional information about the system to the kernel. By default the contents of the ”bootargs” key in the U-Boot Environment are passed on as kernel arguments. While this feature is not always used in current implementations, there are several scenarios where this feature is essential in order to successfully boot Linux on a device.

U-Boot TFTP recovery

As already mentioned there are commands available to use the U-Boot CLI in order to recover from a broken operating system, failing to complete the boot process. But as this requires a serial interface and a trained user and is a generally cumbersome procedure, there was need for a more trivial recovery option.

The recovery is usually triggered by pressing or holding a hardware button available on the device (details are platform specific) during the boot process. If this button is pressed, U-Boot connects to a TFTP server, whose address is specified in the U-Boot environment and downloads a new image file. The correct file is identified by criteria also specified in the U-Boot environment and then written to the flash, starting at the address where the kernel would normally be loaded. If the image was written correctly, a subsequent boot process should be successful again.

This solution does not require a serial interface or manually entering commands, it is sufficient to download the correct image, prepare the TFTP server and push the button.

2.3. Automatic Updates

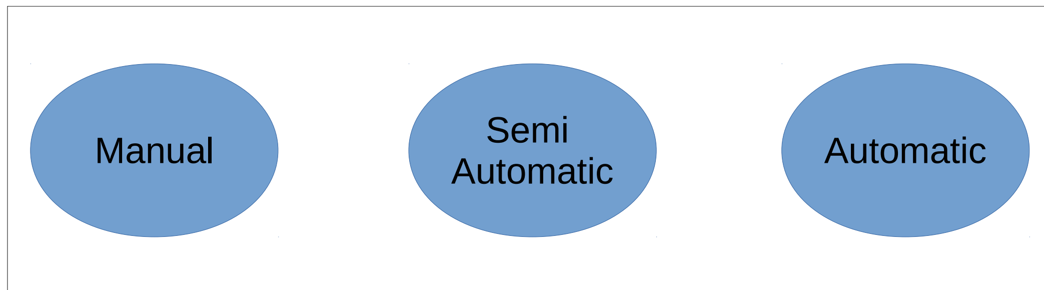
Generally the concept of automatic updates (and system updates in general) is well known on many software platforms. It is common on most operating systems for servers, personal computers (Microsoft Windows, MacOS, Linux, BSD, ...) or mobile devices and the necessity of these updates is widely acknowledged. Depending on the individual requirements and available hardware the details of the update processes do differ in multiple ways. In order to give an easier overview over existing solutions, their requirements, strengths and weaknesses, they will be grouped according to three criteria.

2.3.1. Update trigger

The first criterion refers to the way updates are triggered. This can happen manually, requiring a user to manually check for, download and install updates. This constitutes the most basic but also least favourable approach, as it puts all responsibility on the user of a device and also demands a constant maintenance effort.

An improvement over this approach can be reached by choosing a semi automatic update strategy, that automatically checks for updates and only requires user permission/interaction in order to install the update. This reduces the effort and responsibility on the user side significantly, but it still requires manual effort. This works best for devices where users want to have more control over the running software, like personal computers or mobile phones, not so much for IoT devices, where users mostly do not care about the software as long as it works.

Figure 2.2.: Update Triggers



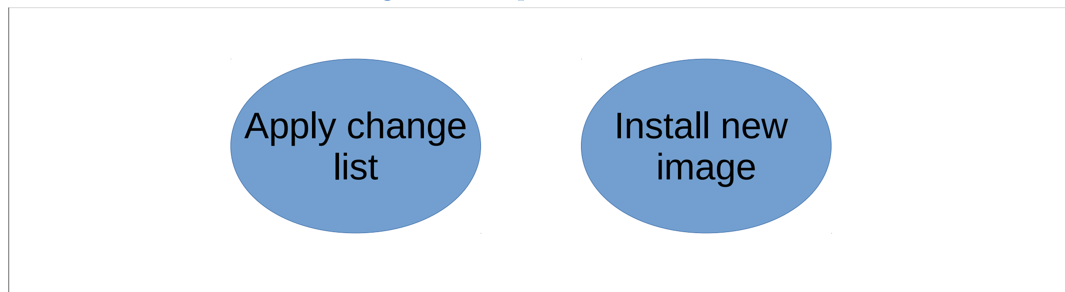
The preferable way of updating is fully automatic and transparent to the user. Generally it is safe to say that most major software systems are striving towards offering an automatic update trigger.

2.3.2. Update mechanism

For the case of this paper only two different update approaches will be distinguished. The first approach imitates the installation process. So an update is basically just deleting the old operating system and replacing it with the new

one. This does not require a special update routine, but instead relies on the installation mechanism, which is also required to initially install an operation system on a device. Generally such an approach is best used when system and user data are strictly separated. This prevents data while upgrading to a new firmware.

Figure 2.3.: Update mechanism



The other commonly used approach to updating a system works at a much smaller scale. Rather than updating the entire system at once, small pieces of the system are updated, whenever necessary. This reduces the update overhead, allows rolling out updates faster and makes it easier to update applications and operating system with the same update scheme. Another key benefit of this way is that it avoids restarting the system during the update process. Instead of being forced to restart the system for every update, restarts can be limited to those updates, where a reboot is necessary.

2.3.3. Recovery scheme

For the scope of this paper recovery schemes are limited to resolving errors caused by the updates rolled out through an automatic update system or through problems during the update process. Recovery from any other errors like hardware failures or user interaction are explicitly ignored, as they would introduce an additional layer of complexity.

A working recovery scheme requires error detection as well as a way to invert the changes made by the update mechanism. Given this close link it is quite

obvious that the choice of an update mechanism strongly impacts the choice of the corresponding recovery scheme. Nonetheless a way was found to group recovery schemes in three different ways, depending on how they track and revert the changes made by the update mechanism. How they detect errors and how the recovery is triggered, was not considered for this categorization.

The easiest solution retains a copy of the original files, avoiding the need to track changes at all. This can either be achieved by making a snapshot of the current system before an update or by copying the relevant files to another location before the update. An update can simply be reverted by replacing the updated files with the original ones. While this approach requires a lot of memory it allows restoring the system even if it was damaged to a point where any internal recovery mechanism fails.

Figure 2.4.: Recovery Schemes

A more memory efficient alternative is to keep a change log describing which changes have been applied to the system. This usually only works with a change list based update mechanism that enables a system to revert individual updates, if an error is encountered. The saved memory however means that the original files are not stored locally and must be downloaded again, which causes problems if the components needed for networking have been broken by an update. As the individual changes on a change list often have dependencies and requirements, correct reversion of such changes is a far more complex task, requiring dedicated logic that too must not be affected by a faulty update and is prone to bugs added by implementation.

Opposite to the update trigger and mechanism, there is no actual need for an update process to incorporate a recovery strategy. Some update processes chose to not implement one and instead rely on users to manually resolve issues

introduced through updates. This choice is usually made when the user base is experienced or the update process is so stable that errors through updates happen rarely.

Ultimately it should be pointed out that a recovery scheme is further complicated by the fact, that it is often very hard to identify the cause of an error. So recovery mechanisms should not only be evaluated by how they revert changes, but also by how they determine that an error has occurred during the update process.

2.3.4. Requirements

Unfortunately the established update processes cannot simply be applied to routers and most other IoT (Internet of Things) products. For one the user interface for most devices is minimal or hidden, because common users have no reason to access it. They simply want their device to work without needing any effort on their part. This dismisses all manual and semi automatic update triggers, because they require user interactions for selecting and installing updates.

As updates usually interrupt the functionality of a system for a brief time span, possible user dissatisfaction through unexpected down times is another potential problem. If the user should remain unaware of both the software inside the embedded device as well as its update procedure, the automatic update trigger should also validate if a user is using the device right now or likely to use it in the near future.

For the choice of the update mechanism it is important to keep in mind that the hardware of IoT devices is often a tight fit to the requirements of the intended job. This cost optimizing strategy leaves no room for extra storage or memory capabilities, which would be needed in order to implement a complex recovery scheme.

For the recovery strategy the most critical issue is the lacking capability to output error messages, which makes is impossible for most users to distinguish between hardware and software failures. This might lead to automatic updates accidentally breaking a lot of devices at once, possibly causing their owners to conceive them as broken.

To prevent this a recovery scheme has to be able to automatically recover from any kind of error possibly introduced by an automatic update. Assuming that updates modify all parts of the operating system including the core components the scheme has to work even if the system fails to boot or cannot connect to a network any longer. Furthermore the recovery process must also be transparent to the user, so both triggering and recovering must happen without user intervention.

Therefore this thesis will search for ways to update OpenWRT systems, which meet the following criteria:

- The updates trigger should be fully automatic
- There should be a non-manual recovery scheme in place
- The user's work should never be interrupted by the update process.
- The found solution should work on a significant percentage of the devices running with OpenWRT
- The update process should be properly secured

3. State of the Art

The chapter will take a look at different automatic update procedures used by established operating systems. These will be analysed in regard to their strengths and weaknesses and the environment they are running in. The goal is to gain an overview over the components of successful update procedures in order to inspire possible solutions for OpenWRT. As the goal of this paper is a fully automatic update process, the presented implementations have been limited to a selected few, which were found to be most interesting for the given topic.

After taking a look at update processes in general, a more detailed look will be taken at currently available update mechanisms for OpenWRT.

3.1. Common update processes

3.1.1. Windows Update

The most common operating system for personal computers naturally offers one of the most used update systems. Unfortunately there is not much documentation available on how exactly Microsoft has implemented their update process, so information could only be obtained by observing the system. Therefore the description of the update process will seem rather vague, compared to others.

Windows Updates can trigger automatically, even giving users permission to select a preferred time, when updates should be installed, to avoid disturbing users while they are working. Furthermore if a user is working while updates are installed and the installation process requires actions, that would disrupt any other running process, the user is given the chance to delay the update to a more suitable time. The combination of these two features minimizes

user dissatisfaction through badly timed updates and is therefore a very useful implementation of an update trigger.

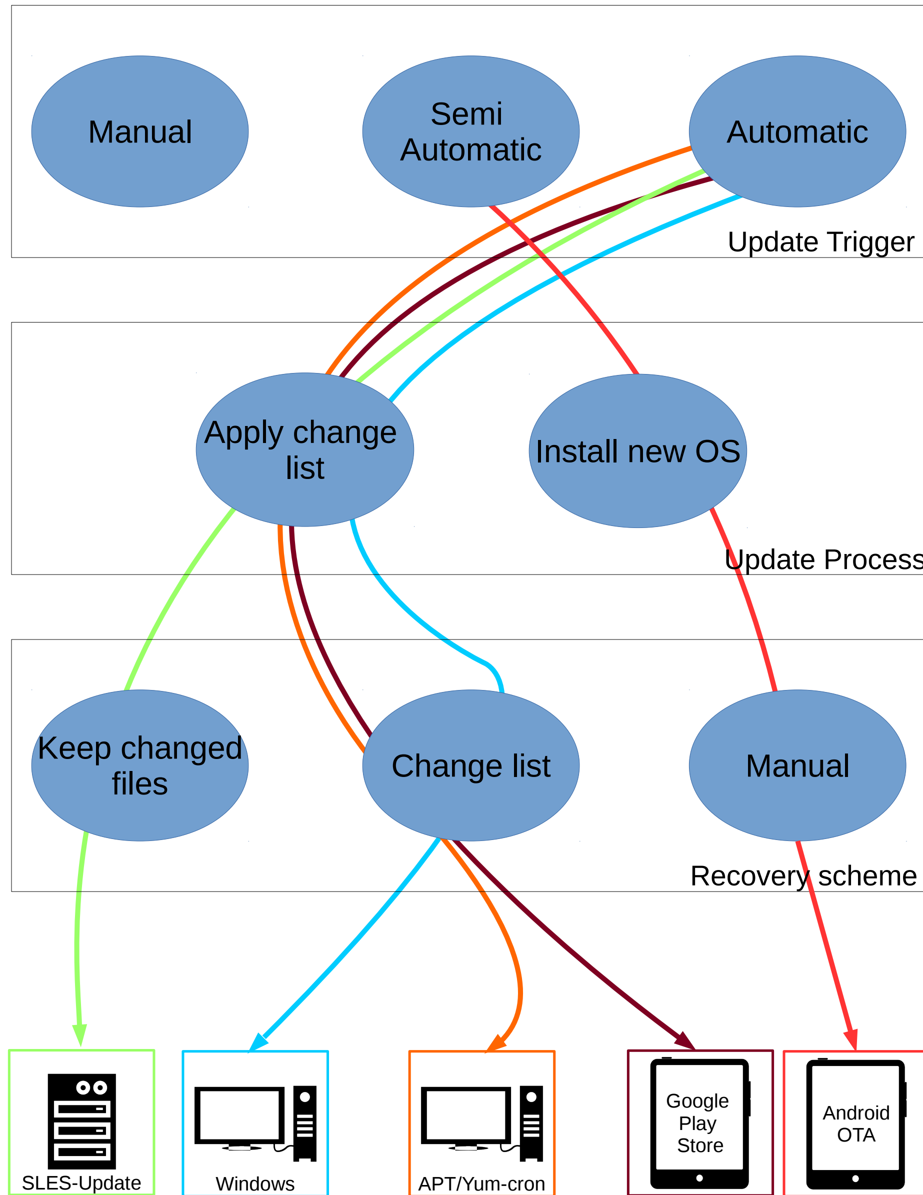
As Microsoft Windows is a relatively large operating system, it would not be feasible to reinstall the entire system for every update. Therefore it makes sense that Microsoft has chosen a change list based update mechanism, which updates a system through many small changes. An interesting observation that should be mentioned in this context, is the development of the Microsoft update process over the last years. In the past it was quite common to install multiple small updates at once, each of them solving a single issue. This of course enabled users to very selectively decide, which of the many available updates they would like to install. In practice this has lead to an increasingly fragmented code base, with an almost unlimited amount of possible update levels, depending on which updates were installed and which were not. In order to reduce this fragmentation and make managing updates easier, Microsoft has chosen to bundle most updates into one monthly update. This lesson should be considered by anyone implementing an update process with a change list based mechanism.

Errors while installing updates are to be expected, especially if an operating system is running on so many different hardware configurations as Microsoft Windows. In the past there have been cases, when Windows Updates have caused the problems they were supposed to prevent. Microsoft does keep a list of which changes have been applied by an update. This allows either automatic reversion of an update, if the error is detected by the system during update configuration or manual recovery through de-installing an update if the error is not detected automatically. If the system fails to boot and cannot fix itself through automatic reversion of the update, the operating system does install a recovery partition along with the main system, that can be booted instead. It can either be used to try and recover automatically by restoring core system components or provide an interface for manual recovery for advanced users.

3.1.2. Linux package managers

This section will take a look at yum-cron and cron-apt, two applications designed to automatically trigger updates for common linux distributions. Simply spoken they only call the manual update trigger regularly and positively

Figure 3.1.: Common Update Processes



confirm all requested user inputs. Initially this sounds like a very risky solution, but if the software repositories used by the package manager are carefully maintained, the danger is minimal compared to the dangers posed by outdated software. Especially common distributions do put a lot of effort into making sure that their packages are compatible and their package managers support many features that prevent unexpected side effects.

Before continuing with the used update scheme it should be pointed out, that some Linux distributions use a rolling release, while others rely on a fixed-release model. Rolling releases get all their updates via package managers, while fixed releases make one major version jump every now and then and only provide security and small feature updates through package managers. This section will only take a look at how package managers work, ignoring the solution used for switching between fixed releases.

The used update mechanism is already given away in the title and another good example for the Unix philosophy to do one thing and do it well. Since there is no central developer responsible for all parts of the system, every part receives updates from its own developer team. Once the update is published, anyone can take the newest version of the software and create a package from it. A software repository provides a searchable list of packages which can be downloaded. As there can be complex dependencies between different packages, some effort is needed to ensure that all packages provided by a repository remain compatible to each other. The reliability of a package based update process mainly depends on the quality of the chosen package repository.

The error recovery scheme in this case relies on the versioning of the provided packages. If the upgrade of a package causes errors the user can simply downgrade to an older version to avoid the issue. Unfortunately there is no mechanism in place to automatically do this, in case an error is encountered. This is again due to the fact that there are complex dependencies between packages and a single update may cause a multitude of changes to other packages. This along with the fact that it is difficult to detect a faulty package is the main reason why the recovery scheme requires a user to take action.

SLES with yast

An individual look will be dedicated to the update process implemented for SUSE's Linux Enterprise Server. The yast tool allows for easy configuration of automatic updates through a GUI. Additionally it provides options for the update interval, licence agreements, how to handle interactive updates (requiring user input) and filters determining which packages should be updated. This grants more fine grained control over which updates should be installed, than provided by the other solutions.

The other even more interesting feature about the SLES update procedure is their recovery option via LVM snapshots. Instead of keeping a change list, every update is preceded by taking a snapshot of every possibly changed volume. After a successful update the snapshot is deleted. If the update encounters any kind of error, the previous system can easily be restored. The advantage of this scheme is that the system can be restored to it's original state, even if the failed update prevents the system from booting.

3.1.3. Android Updates

The Android operating system is possibly the most interesting case study for the topic of this thesis. Just like OpenWRT Android is an open source operating system, which must be customized for every device. Additionally the system itself is free to use and always sold alongside the corresponding hardware. So this software is generally distributed by hardware vendors, which rely on and use the operating system, but do not have their core business focus on the software development.

Finally it should be pointed out that there are multiple similarities in the partition layout of Android and OpenWRT. In general Android uses the following partitions:

1. **boot:** Contains the linux kernel and a minimal root filesystem. It is responsible for mounting partitions and initializing the system located on the system partition.
2. **system:** Contains Android system applications and is generally mounted read-only. Should only be changed through an over-the-air (OTA) update.

3. **vendor:** Contains all aplications that are not generally shipped with Android but have been added by the vendor responsible for customizing Android to the individual device. Should only be changed through an OTA update.
4. **userdata:** Stores all applications installed by the user as well as all the data created by those applications. The content of this partition should not be updated through an OTA update, but by an App Store, most commonly Google Play.
5. **cache:** Memory area used for holding temporary data. Also used for storing OTA update packages.
6. **recovery:** Contains an alternative linux system with another kernel and the binaries needed to modify and update the other partitions.
7. **misc:** Stores process information about actions taken by the recovery system, in case the device experiences a shutdown during an update process.

After this it should be quite obvious that there are strong similarities between OpenWRT's SquashFS root file system and Android's system and vendor partition. Both of them are mounted read-only and only modified during a system update, therefore they both only hold data and applications that are an integral part of the system and cannot be removed easily. An equal degree of similarity can be found between OpenWRT's rootfs-data and Android's userdata partition. Both of them are used for storing all user triggered changes to the operating system and cannot be updated in the same way as the rest of the system. [1]

The following two sections will take a closer look, how Android updates both the system and the userdata partitions and most importantly what experiences can be drawn from another operating system that has many similarities with OpenWRT but is far more widespread.

Android OTA

An Android over-the-air (OTA) update is used to update the system and vendor partitions of a device. It can either deliver security fixes, new features or move the device to a new major version of the operating system. The search and download for such an update usually happens automatically, afterwards however

the user is prompted to install the update. The update is then moved to the cache partition before the device reboots into recovery mode in order to be able to write the system and vendor partitions. Once the recovery system has been booted, it finds the update located in the cache partition, verifies the update cryptographically and then updates the boot, vendor and system partitions according to the update instructions. During this entire process, information is written to the misc partition in order to be able to continue/revert the update process in case of an unexpected shutdown. If the recovery partition should also be updated by the OTA, the new contents of the recovery partitions are stored on the system partition. Once the update has been completed, the device reboots again. If the system detects during the startup process, that the contents of the recovery partition do not match the expected contents, stored on the system partition (in case recovery was updated by the update), the new recovery is flashed to the recovery partition. Once the device has successfully rebooted, the update process is complete. [1]

At this point is seems interesting to compare the described update process against the requirements laid out in section 2.3.4. The update process is properly secured, because the update is cryptographically verified, before it is installed. Usually it is even verified twice, once by the system before prompting the user to update and once again by the recovery before the update is installed. The update process does work on all customizations of Android, unless the customization broke the update process. The user is never interrupted, because he is prompted for permission, before starting the update process. Thanks to the misc partition the recovery is able to continue if an update is interrupted, but if the update itself is faulty there is no mechanism in place to restore a working operating system. The update trigger is only semi automatic, but the user prompt prevents user disruption and for a device that is directly used by its owner, has an extensive and commonly used UI and is generally expected to accept incoming calls and messages, it can be argued that a semi automatic update trigger is more user friendly.

So except for some weaknesses in the automatic recovery area, it seems that Android does have a very stable update mechanism. Its only weakness lies in the fact that an update that prevents the system from being booted, cannot be reverted automatically. This puts some responsibility on the update providers, but that is of course also true for the updates provided by Microsoft or Linux package repositories. An important difference however lies in the fact that every

update has to be customized to every hardware platform, which makes it easier to test an update but also multiplies the workload by the number of devices.

And this leads to the most problematic thing about Android OTA updates. They are hardly provided in practice. By taking a look at the current distribution of Android Versions it becomes clear that most vendors simply do not take the effort to update their devices for their entire lifetime. At the time this paper was written, Android 7.0 had been released for more than 10 months, yet less than 10% of devices had been upgraded to the newest version and more than 20% of running Android devices are still running versions that are more than 3 year old. [2]

So even a working update procedure does not ensure that updates will be provided, especially if these updates have to be customized to every hardware platform individually. While this paper will not strive to provide an answer to this dilemma, it certainly in an important lesson one can learn from the Android ecosystem, because IoT devices based on OpenWRT will also need their updates customized to their specific platform.

3.1.4. Android App Store

As already mentioned in the Android partition table, the userdata partition is not updated via Android OTA updates. Instead an app store, most commonly Google Play takes over that job. In essence it works very similarly to a Linux package repository, providing users with an easy interface to download, install and update applications suited for their operating system. However due to the lacking update support from hardware vendors selling devices running Google's Android, an interesting transition took place.

Over time more and more parts of the Android system, especially security critcal parts in bad need of regular updates, were extracted into applications, that could be updated via Google Play. Considering the fact that Google Play is installed at a vast majority of Android devices, this seems like a reasonable step and allows Google to provide some security updates centrally, without needing access to the system partition. On the other hand this decision cripples the separation of system and user data.

This step demonstrates that the company behind the development of Android has a strong interest to keep their system secure and up to date, but the hardware vendors that depend on that very system to sell their devices, do not share that interest. The reasons for these diverging interests would be interesting to study, but are not within the scope of this thesis. Relevant is only the lesson that updates will only be delivered if the party responsible for providing them, does have an interest in the security and quality of the software in question. Especially with the internet of things on the horizon, this lesson should be kept in mind.

3.2. Update processes for OpenWRT

After taking a close look at many established update processes, the focus will now be shifted towards already existing OpenWRT update processes. As already explained OpenWRT is a highly configurable system that is used in a multitude of different contexts and environments by users with different proficiencies.

This results in different groups having different requirements towards an update solution. This section will quickly cover existing solutions, identify the additional/alternative design goals they aim for, examine their advantages and disadvantages and maybe redirect some readers to other projects which are more tailored to their needs.

3.2.1. Manual Update

Background information

The default update mechanism for OpenWRT is a simple manual update, which can be triggered via the command line or the web interface. However even at this trivial stage there are two different distinctions in regard to updates that should be kept apart, depending on which parts of the file system they concern.

OPKG-How it works

The already existing package manager is capable of both finding and installing updates for already installed packages. So at least this part of the system could easily be updated with a cron job similar to the Linux package managers already discussed in section 3.1.2. In theory such a package approach (like it is common for Linux Desktop Environments) could be extended to provide a reasonable automatic update solution. Unfortunately there are reasons why this update mechanism is not taken generally.

As already pointed out in chapter 2.1.1 most implementations of OpenWRT do use a dual layered root file system, with a read-only base and an overlay used to store modifications made by the user. This means that updating anything stored on the read-only filesystem in the conventional way would cause the updated version to be written to the overlay. While this would work as expected, it would also quickly use up the available space in the overlay partition. An update process based on this idea is investigated in section 3.2.3.

Sysupgrade-How it works

Of course OpenWRT does ship with a way to upgrade the SquashFS filesystem. This method is referred to as ”sysupgrade”, which basically is a complete reinstall of the entire firmware. Every time an update is shipped to the customer, the provider of the update builds a new firmware image, complete with both a kernel and a new SquashFS root partition. The sysupgrade utility is capable of transferring files from the old overlay partition to the new one.

The sysupgrade process first copies the parts of the overlay intended to be carried over into its volatile memory. Afterwards it creates new a ram-based filesystem, copies the currently running firmware there, kills all running processes and switches to that new temporary filesystem. Once the switch has been completed the flash memory is overwritten with the new image. If any files have been selected to be carried over, they are copied to the new overlay. Once the final step is completed an automatic reboot is triggered and the system reboots into the new firmware.

Review

Both OPKG and sysupgrade work on almost every device running with OpenWRT. They rely on a user to manually trigger an update and use different update mechanisms to update different partitions. For opkg there is some kind of a recovery scheme available, because OpenWRT does have a failsafe mode, where the overlay partition is simply not mounted, allowing a user to reboot a device and fix a faulty package.

For the sysupgrade part however, there is no such thing. Generally a sysupgrade will not touch the boot loader partition, so advanced users might be able to make use of U-Boot's TFTP recovery to restore a device after a failed update, but this is usually beyond an average user's capabilities. This means that this update process also relies on users to make sure that no unexpected interruption occurs while the changes are written to the flash.

Summing up it could be argued that opkg and sysupgrade are tools that offer decent update mechanisms, but the manual update trigger and recovery scheme makes them a poor choice for an update procedure.

3.2.2. Gluon Updater

Background Information

Gluon is a framework which is used to build specialized OpenWRT firmwares for wireless mesh nodes. It is developed and maintained by the Freifunk community, which tries to provide an open wireless internet access to everyone. Different Freifunk communities are generally limited to a single city or district and mostly consist of technically advanced users, which allowed them to include an automatic update function that would not be suited for many other cases, but works nicely for them. [4]

How it works

In simple terms the Gluon autoupdater consists of a few lua and shell scripts run by a cronjob. Through these scripts a configurable server is queried in

a configurable interval for available updates. If an update is available, it is downloaded, verified and installed via the integrated sysupgrade command. In order to prevent some of the issues introduced through automatic updates, a few other configuration options are available. The first one allows the user to specify a priority, which defines the maximum number of days that may pass between the release of an update and its installation. The individual nodes will decide at random at which point in the time between the release and the specified priority the update will occur. This ensures that a faulty update can only break a limited amount of nodes instead of the entire mesh. Secondly there is a parameter to decide at which time the update should be run, which can be specified to the hour. The minute to download and install the new image is randomized to distribute the load on the download server. [4]

Review

The major disadvantage of this approach is that the individual Freifunk communities all use their own modified versions of OpenWRT with Gluon, meaning that every community has to run its own server to build and distribute the updates, which is only feasibly if you are running a significant number of nodes.

Apart from this however, this approach meets many of the requirements specified in section 2.3.4: It is applied automatically once an update has been released to the update server, it updates at a time where it is reasonable to assume that no users are currently needing the device, the downloaded image is signed with a signature that can be verified and since this approach is using the default sysupgrade logic in the end it can be extended to almost every device running OpenWRT.

The only requirement completely ignored by this solution is the automatic recovery and that makes sense when you think about the context the Gluon autoupdater is used in. These updates are for nodes in a mesh managed by experienced users, that are able to distinguish between a software failure caused by a faulty update and a hardware breakdown. Additionally these advanced users would have no troubles to restore a working state on a device, even if the firmware was completely broken. The time delay in rolling out the update ensures that one faulty firmware version is not rolled out to the entire mesh at

once but at a much slower rate allowing for timely intervention by the users monitoring the nodes of the mesh. Therefore the developers of Gluon had no reason to worry about an automatic recovery mechanism, they can handle the recovery manually if necessary.

3.2.3. Turris Omnia Updater

Background Information

The Turris Omnia is a single owner/home office router developed by the CZ.NIC. It provides a device with open hardware and open software based on OpenWRT. [8]

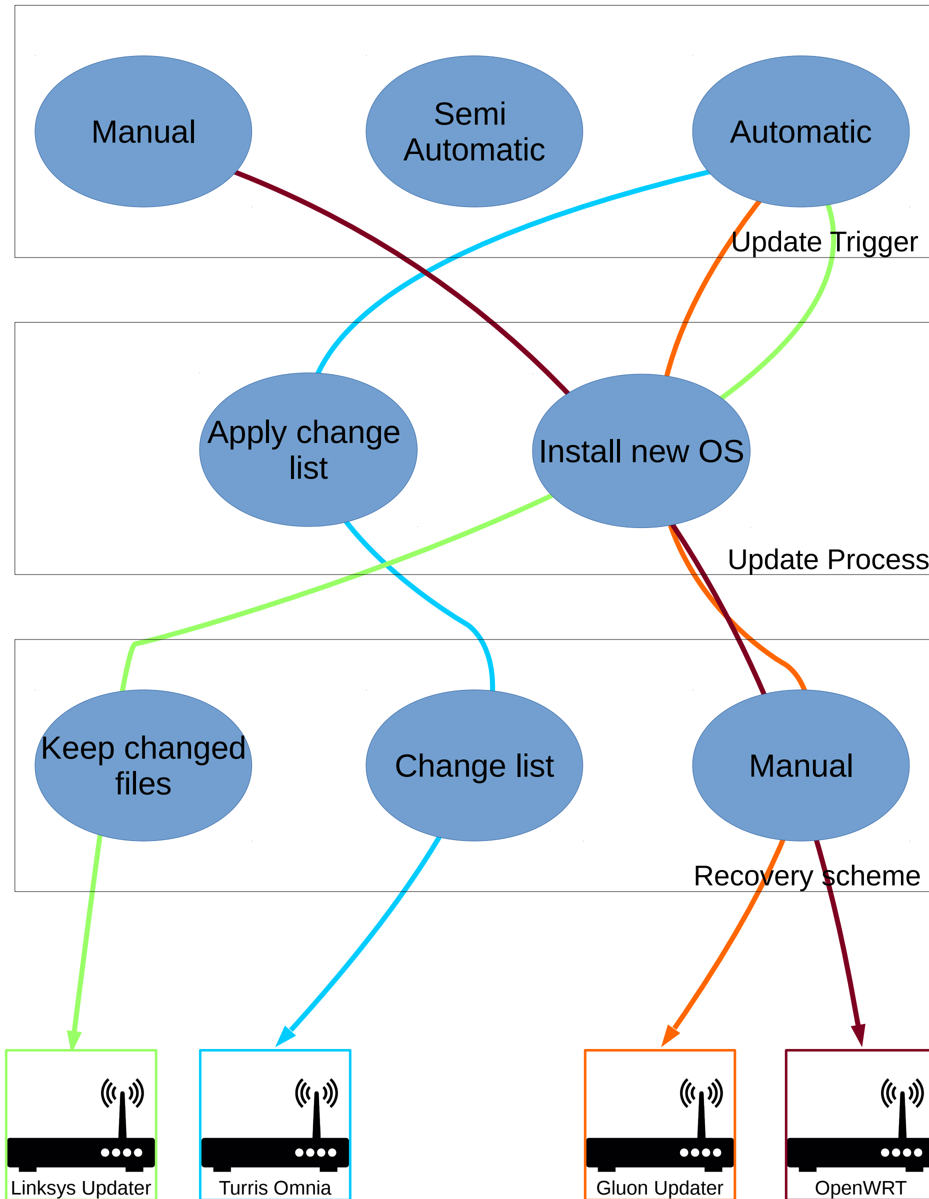
One of the selling points for this device is the fact that it receives automatic updates. This project is especially interesting because their open source approach does not only reveal how they finally implemented their update procedure, but also documents what other attempts of theirs met failure. Before looking into the update mechanism in more detail however, it is important to glance at the hardware specifications of the device in question. The most important bullet points are 8GB of flash memory, up to two GB of RAM and a 1.6GHz dual-core ARM CPU. This hardware is many times more potent than the one usually found in this class of devices. Keep in mind, OpenWRT is designed to run on devices with as little as 4MB of flash memory and has already been run successfully with only 2MB. So it is only natural that a firmware compiled for hardware as potent as this, does not need the default memory layout.

In addition to this the Turris Omnia project defined slightly different criteria for their update solution.

- Upgrade/downgrade of packages
- Addition and removal of packages
- Major version updates
- Preservation of configuration
- Recovery from power loss during upgrades

The need to not only upgrade but also downgrade packages seems reasonable in order to resolve partially broken updates. Notable is the requirement to add and remove software on the client device through automatic updates, which

Figure 3.2.: OpenWRT Update Processes



means that no consent of the user is required. This is certainly interesting to push new features to the devices at a later point in time or remove functions that have proven to be a security vulnerability, but many users may not be comfortable with another person having so much power over their devices. Of course the open nature of the system allows every user to restrict these updates, if he does not like them. Lastly there is no goal about not interrupting working users, but similar to the Gluon updater, the time of the update can be configured so that the odds of an interruption are minimal. [17]

How it works

Originally the Turris Omnia team tried to implement a solution similar to the one envisioned in chapter 3.2.1 by implementing a series of scripts around the package manager opkg, in order to update the device. Thanks to the relatively huge amount of flash memory available there was never any need for a read-only root file system, so every part of the system can be updated individually. Furthermore there is enough storage available to duplicate all critical parts of the system on the flash memory before updating them, so that the system is still able to recover itself, even if a power loss occurs during an update.

As already stated in chapter 3.2.1 however, opkg is currently only used to install additional packages or update selectd packages. When the Turris Omnia team tried to use it as the foundation of their update strategy they encountered a series of problems with the dependency resolution and installation order of packages. Some of these are due to the fact that opkg like most software written for OpenWRT has a reduced feature set in order to reduce the application's size and resource consumption. Others however are simply bugs that were not encountered before, because opkg is generally not used in this way. [17]

Ultimately it was decided to abandon opkg for this project and it was replaced with a new updater solution, they call "Updater-ng". This new solution consists of a mixture of lua scripts and c programs, which access the same repositories and databases as opkg, but provide many new features not offered by opkg. [17]

Review

While updater-ng is a solution developed for one specific hardware platform, all components are open source and available to the public and there is no argument against porting this updater solution to other platforms. There is even a good chance that the team from Turris Omnia would support such an action, if only to increase the user base of their update solution. Of course this is only feasibly for devices that have enough flash memory integrated to be able to afford dismissing the default memory layout of OpenWRT. In practice this might be interesting for expensive embedded devices that can afford the increased hardware costs in order to gain a flexible and reliable update mechanism. For all common devices, which have very limited flash space available, this approach cannot be used directly.

3.2.4. Linksys Autoupdates

Background Information

The last update process analysed in this chapter is a vendor solution developed by Linksys. It is not offered for all devices, but it is designed to be used by private home users, so it is no surprise that it meets most of the requirements laid out in section 2.3.4. Unfortunately it is not open source and not very well documented, so many technical details were not available to the author.

How it works

From a user perspective, it is very convenient to set everything up. First one has to create a Linksys cloud account, while being connected to the wireless network created by the router. This automatically associates the device with the account. Then it is sufficient to simply select a checkbox in the web interface of the cloud service in order to have the device updated automatically. [10]

While there are no technical details about the system provided, an investigation of the files and partitions does give an insight into how the system most likely functions. There is not much use in investigating the update trigger, since the

device has to be connected to the cloud account and that cloud account can manage the device, it is obvious that the update can also be pushed by the vendor to the devices, instead of making the devices check for an update. This provides a fully automatic update trigger, but little can be said about how it works in detail. In order to find out about the chosen update mechanism and recovery scheme, it is most enlightening to take a look at the flash partition layout:

```
mtdparts=mtdparts=armada-nand:2048K(uboot)ro, 256K(u_env),
    256K(s_env), 1m@9m(devinfo), 40m@10m(kernel), 34m@16m(rootfs),
    40m@50m(alt_kernel), 34m@56m(alt_rootfs), 80m@10m(ubifs),
    @90m(syscfg)
```

There are a couple of things that attract attention. The first thing of course is the fact, that there are two kernel and two rootfs partitions, indicating that there are always two versions of the firmware installed. This would allow the system to update one firmware and fall back to the alternative one, in case an error is encountered during the boot process. This also hints at the fact that Linksys has chosen an update mechanism that installs an entirely new firmware, possibly based on OpenWRT's sysupgrade mechanism.

Next it is worth noting that there is only one overlay file system using ubifs (an enhanced version of JFFS2), meaning that while the entire system data is rewritten by updates, all user changes remain untouched.

The syscfg partition is also using the ubifs file system and appears to be used to store configurations and some update-related information. This partition is mounted inside the /tmp directory (which is unusual considering that /tmp is usually a volatile ramdisk in OpenWRT). Taking a look at the folder shows files like files-to-keep.conf or detected_hosts_history along with a sysconfig.dat file, which appears to contain all settings made through the web interface. Why these changes are stored there and not on the overlay file system or if these are just backups in case errors occur with the original overlay is unknown.

So it seems a reasonable assumption that the update process is triggered by the cloud server, causing the download of a new firmware. Afterwards this firmware is flashed to the currently inactive system and then the system reboots into the alternate now up-to-date firmware. Unfortunately it is not clear if

and under which circumstances the system is capable of recovering. Ordinarily one would expect that, if the system encounters an error during the startup process, the alternative firmware is booted on the next try. But the author was unable to find any mechanism inside the bootloader that would enable such a behaviour.

Review

As already stated Linksys has provided the most end user friendly update solution for OpenWRT. It triggers automatically as soon as the account has been set up and linked with the device and the corresponding checkbox has been selected. It would be preferable if enabled updates were the default, rather than requiring users to make that setting on their own. In addition the requirement to pair a device with an account does bring about a multitude of privacy concerns, especially when the information flowing between device and cloud seems to contain more than just updates and technical information (remember the detected_hosts_history).

For the update mechanism it is possible that Linksys uses a modified sysupgrade routine, but it is just as likely that the update is written to the flash in some other way (For example by using mtd or dd directly).

The recovery scheme appears to be implemented by keeping the entire running system untouched and just changing the system that will be booted next time. Surprisingly no mechanism could be detected that would ensure that the system reboots into the old firmware if starting the new firmware fails in any way. This may be because this function is hidden inside binaries that were not analysed or because there simply is no such mechanism in place.

4. Hardware

For the research done in this paper, three different devices were analysed and evaluated. All three devices could be found on the table of supported hardware of the OpenWRT project and were deemed interesting devices for different reasons. This chapter will give a quick overview of the hardware capabilities and the software configuration of these devices. This part is relevant to the overall research mainly for two reasons. First it was a clearly stated goal, that the chosen update mechanism should work on a majority of OpenWRT systems, so it is necessary to take a look at some commonly used ones. Secondly it gives a nice impression of the quirks and inconsistencies one encounters when working with the hard- and software of embedded devices. When trying to find a generic update process understanding and working around those inconsistencies consumes a lot of time and leads to unexpected difficulties.

4.1. TP Link Archer C7

The first device is a quite commonly chosen device for people using OpenWRT. The TP Link Archer C7 comes with capable hardware for a reasonable price.

4.1.1. Hardware Overview

There are two different hardware revisions available for this device, that come with slightly different hardware capabilities. For this paper we are going to focus only on the second revision, as it is the one available for testing.

Table 4.1 shows that the memory of this device is limited to 16MB, but it should be pointed out, that the first revision was limited to only 8MB, so the

Table 4.1.: Hardware Overview

Attribute	**TP-Link Archer C7 v2**
Architecture	MIPS32 (Mips74Kc)
System on a chip (Soc)	QCA9558-AT4A
Flash Chip	Winbond W25Q128FV
Flash capacity	16MB
RAM	128MB
Bootloader	U-Boot
Price Estimate	~80€

device was actually designed to also work with half of its available flash memory.

4.1.2. Observations

Bootloader

The first interesting fact about this device is the way its bootloader is used, because it deviates from the suggested U-Boot setup. Usually it is recommended to have one partition holding the compiled bootloader (usually called u-boot) and a second partition to hold all configurable parameters of the bootloader (usually called u-boot-env). This prevents configuration changes from damaging the bootloader itself, because even in case the configuration of the U-Boot is broken, there is still a minimal backup configuration compiled into the bootloader to enable recovery. For this device however, TP-Link has chosen to remove the u-boot-env partition entirely and instead included all their configuration changes into the backup configuration. This may save some flash memory, but it also is a major obstacle for any recovery scheme requiring access to the bootloader, because every configuration change would require rewriting the entire bootloader, bricking the device in case of a failure.

Another problematic issue that should be mentioned at this point is the fact, that the version of U-Boot used on this device is not very up to date. It is a modified version of U-Boot 1.1.4, which was released in 2005 and according to

Table 4.2.: Hardware Overview

Attribute	**Zyxel NGB6716**
Architecture	MIPS32 (Mips74Kc)
System on a chip (Soc)	QCA9558-AT4A Rev 2
Flash Chip	Macronix MX25L128 + Hynix H27U2G8F2CTR-BC
Flash capacity	16MB + 128MB
RAM	256MB
Bootloader	U-Boot
Price Estimate	∼100€

the buildinfo last build in 2013. A closer look at the source code reveals that TP-Link has added some custom modifications to the bootloader and even applied some patches included in later versions of U-Boot, but they never released their own changes, so they could be included in future versions. Therefore it is not feasible to upgrade the bootloader to a newer version, so whatever update process is chosen, it must be compatible with the oldest versions of U-Boot still in use.

4.2. Zyxel NBG6716

The second and most used test device was Zyxel's NBG6716 router. It is more expensive than the first device and comes with quite a few hardly understandable inconsistencies, but these inconsistencies along with its hardware capabilities made this device an ideal candidate for implementing an update prototype.

4.2.1. Hardware Overview

The highlight of this system is without a doubt the fact, that it contains two different flash chips, which even use different technologies, as one is based on NOR gates and the other on NAND. That in combination with the high storage capacity of more than 128MB made it easy to run tests without risking bricking the device.

4.2.2. Observations

Bootloader

The first important observation to make is that Zyxel has made more intense modifications to the U-Boot, than it was the case for the other vendors in question. Instead of the normal bootloader, a so called zloader (which seems to be a Zyxel specific implementation) is called by the bootloader and overrides the default console. The only observed advantage of this zloader is that it prevents users from entering the U-Boot command line interface without the correct debugging password. That along with the fact that the u-boot-env partition is mounted read-only, prevents users from changing the bootloaders configuration. Again this increases the difficulty for any update solution needing access to the bootloader.

Another thing worth noticing is that the kernel is not stored on raw flash as a separate partition, as suggested by the default memory layout described in section 2.1.1. Instead the kernel is stored in a compressed form inside the already compressed JFFS2 file system. Instead of loading the kernel directly from flash, the following command is used to make U-Boot search through the entire flash and all recognized file systems for the specified file. If the file is found, it is copied the given address in RAM.

```
fsload 80400000 /boot/vmlinux.lzma.uImage
```

When trying to install two different firmware version along each other, the kernel was always loaded from the first of the two possible file systems. When trying to work around this issue by giving different names to the different kernels, an interesting issue was encountered. The only file path from which the bootloader was able to load a file was the default kernel location. For every other path, the command spent a long time searching through the entire flash before reporting a failure. The suspected reason for this lies in the fact that the bootloader was modified to be able to locate the kernel more quickly than it usually would, but in return the generic functionality of the used U-Boot command was limited to a special use case. This issue might have been one of the reasons for not using the second rootfs partition.

4.2.3. Partition Layout

The reasoning behind the partition layout of this device is still a miracle at the time of writing this thesis. The first flash chip contains four partitions: *u-boot*, *env*, *RFdata* and *nbu*. The purpose of the first two partitions is clear, the third partition probably contains some SoC specific information, but the fourth is completely empty. This means that 3712KB of 4096KB are not used by design, which is an unusual waste of memory for a router.

But things get even more confusing, when the focus is shifted towards the second flash chip, which offers 128MB of memory. This second flash is divided into 8 partitions: *rootfs_data*, *romd*, *header*, *rootfs*, *header1*, *rootfs1*, *bu1* and *bu2*. Of those 8 partitions, only two were found to be actively used during the analysis of the vendor firmware. These two were *rootfs_data*, which acts as a JFFS2 overlay file system and *rootfs*, which also uses JFFS2 format, but is mounted read-only, like a SquashFS filesystem usually would. The header partition contains a few bytes of data detailing the currently installed firmware version, but it can be deleted without causing any disruptions, so it appears that this partition is not needed. All other partitions are completely empty and not mounted into the system making sure that they cannot be used by any applications. All functions and applications shipped along with the firmware were tested to see, if any of them would make use of the extra partitions, but none of them did, so they truly remain unused.

One weird design decision that should be pointed out at this point is the fact, that Zyxel choose a ridiculously small size for the *rootfs_data* partition. It is limited to only 2MB, which means that users running an unmodified stock firmware run into out of memory issues after using up just 2MB of flash storage, even if there are about 100MB of memory available and unused.

Another point worth noticing at this point is that the rootfs uses a JFFS2 filesystem, rather than a SquashFS. This is because SquashFS requires an additional layer like mtdblock to make the flash device look like a block device to the operating system. Since this layer does not offer bad block correction, it is necessary to either wrap the SquashFS in another file system like JFFS2 or UbiFS or as done in this case, use the other file system directly.

The unused partition gives the impression, that the firmware was originally

Table 4.3.: Hardware Overview

Attribute	**Linksys WRT1200AC**
Architecture	ARM (ARMv7)
System on a chip (Soc)	Marvell Armada 385 88F6820
Flash Chip	Spansion S70FL01GS
Flash capacity	128MB
RAM	512MB
Bootloader	U-Boot
Price Estimate	~120€

designed to support an update process similar to the one implemented by Linksys, considering the doubled header and firmware partitions. For some reason however this feature was cut from the released firmware without reverting the changes in the partition layout. This discovery makes this device the most interesting one for implementing a new update process similar to the one pursued by Linksys, because the partition layout is already prepared for it.

4.3. Linksys WRT1200AC

The final device used for testing in this paper will be a Linksys device that is already shipped with a working automatic update process discussed in section 3.2.4. Generally it can be said that this system is the most expensive one, but it was also found to have the most consistent setup.

4.3.1. Hardware Overview

At first glance this system differs from the others in two things: It uses only a NAND flash and relies on a different CPU architecture. Historically ARM based routers are not the standard, but they are likely to get more common, as ARM platforms are generally used more frequently thanks to other device types like mobile phones.

4.3.2. Observations

Many observations about this device were already discussed in section 3.2.4 and as already stated, this device makes fewer odd design choices, than other systems. Worth pointing out seems the fact, that UBI is used here as a layer between the operating system and the flash, which allows using SquashFS for the root file system, even on a nand partition.

Confusing is the chosen overlapping partition setup, which contradicts the normal OpenWRT partition layout. Instead of having distinct partitions for kernel and filesystem, as one would expect after seeing the default layout (Fig: 2.1), the *rootfs* partitions lie inside their corresponding *kernel* partitions and both kernel partitions lie inside a *ubifs* partition.

Another confusing observation lies in the fact, that the bootloader passes JFFS2 as file system type to the Linux kernel, even though the root file system is then mounted as SquashFS. This may be because the SquashFS is inside an UbiFS and UbiFS is technologically similar to JFFS2, but it does not make it easier to understand the system setup.

5. Strategy selection

After investigating other update processes and the available hardware, it is time to think about how an update strategy with the required features (see section 2.3.4) could look like in theory, before trying to implement this solution on the available devices.

5.1. Desired Update process

5.1.1. Update trigger

This is the most obvious point, as the goal for this thesis was explicitly to search for an update process with an automatic update trigger. Nevertheless it seems reasonable at this point to consider a semi automatic update trigger as an alternative.

This would require a proper user interface and notification system to interact with the user in order to obtain permission to update. It would reduce the risk of user dissatisfaction through unexpected updates, reduce the risk of power outages or other unexpected shutdowns during the update process and make it easier for users to notice, if an update has caused a problem. In theory the interface for a semi automatic solution could be implemented with a modified captive portal, that relays users website calls towards a static page, that informs them, that a new update for their router is available. This would have to be modified to prevent IoT devices or web applications other than browsers from being blocked by this update portal. And even than users without access to the router would still be notified about an available update, effectively introducing a new source of user dissatisfaction. Summing up a semi-automatic update

trigger might be an interesting alternative for routers in certain scenarios, but a more general IoT update process will definitely require a fully automatic one.

That the update process is started without any user interaction does not mean however, that an update may take place at any given time. Independent of the chosen update mechanism, updates will require a reboot of either the entire device or at least certain services, interrupting current work. In order to prevent this the update time should by default lie during night hours, where the chances of user activity are lower. Even better would be a solution where the device itself analyses if it is being used and only triggers the update, if it is not. A low effort solution would be a script that uses tools like bmon or iftop to check the currently used bandwidth on all network interfaces and if it is low enough (indicating that no device is actively using the network right now) the update happens.

5.1.2. Update mechanism

Selecting the best update mechanism was possibly the hardest decision to be made while working on this thesis. For a long time the question was left unanswered instead researching both options trying to find a distinctive reason to prefer one solution over the other. Therefore this section will detail how both solutions could work in theory and afterwards explain, which one was ultimately chosen over the other.

Apply change list

The first strategy is inspired by the the updater-ng discussed in section 3.2.3. As already mentioned their update mechanism cannot be easily transferred to other devices, unless they offer similarly capable hardware, especially in regards to flash capacity. One of the key requirements towards an update solution for this paper, is the need to be able to support a majority of OpenWRT devices and the majority of these devices has only between 4MB and 16MB of flash storage available.

So the first question to ask is if there is a reasonable way to adopt the updater-ng to a system with as little as 8MB of memory. The first idea that comes into

mind, would be to deviate from the default flash layout by using a read-write root filesystem, rather than storing all updates on the overlay partition. In principle a default OpenWRT installation takes up between 2 and 4 MB of flash memory excluding the overlay filesystem. However switching to a read-write file system (jffs2 or ubifs to name commonly chosen options) increases the space requirements by 20-30%[14]. Considering the possible benefits gained by automatic updates, this would seem like a reasonable trade-off. The only other disadvantage of a read-write file system lies in the problem that OpenWRT usually allows users to boot into a failsafe mode, in which the overlay partition is not mounted. This feature would be lost, but as this mode can only be triggered manually, it is not useful for the desired recovery scheme and most users lack the skill to use such a feature anyway.

Another potential issue that should be considered lies in the fact that the normally used filesystems supporting write operations like JFFS2 and UbiFS are compressed. So even if the size of a file is known, package managers like OPKG or updater-ng would not know how much space the file will finally occupy on the flash chip. A non compressed filesystem would roughly double the flash size requirements leaving to few possible target devices. This means that the exact memory requirements of an update are only known after applying it. This can of course be mitigated by having space reserved for updates and installing updates on a test device first, measuring their memory requirements, but in environments with very tight memory, this lack of certainty will be a source of trouble.

Install new OS

The alternative strategy makes use of the already established procedure to manually upgrade firmware and is mainly inspired by the Gluon and Linksys solutions. This way the maintainer only has to release a single file, if he wants to release a new firmware and the update can easily be installed manually, if automatic updates have been disabled. One important disadvantage to keep in mind at this point lies in the fact that some users will modify the configuration and even install additional packages. These packages can be persisted through updates, but their continued functionality is not guaranteed and even if it were, they would never be updated through this mechanism. These issues could

maybe be resolved by using a package based updater additionally, similarly to how it is done by Android. Generally it can be said that this mechanism needs much less discussion than the other one, as it is already established and both Linksys and the Gluon project have proven that it can also be used in combination with an automatic update trigger.

Chosen mechanism

After reviewing both available options and looking at their advantages and disadvantages there was one perspective that lead to a decision. The possible recovery schemes depend greatly on how the update mechanism works, so the decision made at this points also effects the decisions made at the next section, where the goal will be to find a non-manual recovery scheme.

It must be expected at all times that an update is either faulty or not installed correctly for unknown reasons, like manual shutdowns or power outages. To guarantee automatic continued operation even under such circumstances, another operating system that is known to work must remain untouched by the update process. In chapter 3 several systems were mentioned, that rely on update mechanisms with change lists. And while they are often capable of recovering from certain errors, the only feasible way to guarantee recovery is through a recovery partition, as used by Microsoft Windows. This removes one key advantage of the change list mechanism, namely the fact that it would not require any support from the boot loader but could entirely be handled and configured inside OpenWRT.

Additionally it should be kept in mind that the number of read-write cycles for flash chips is limited compared to other memory devices. Therefore it does make sense to bundle changes into a few major updates instead of continuous small updates, as it reduces the amount of required erase operations on the chip.

These two points along with the fact that the established manual update procedure is already well established has lead to the conclusion that a mechanism that updates by installing a new OS will be easier to implement and work more reliable in most scenarios.

5.1.3. Recovery scheme

This part of the update process can be considered optional in general but as already argued it is vital for routers and IoT devices. One key aspect has already been mentioned in the previous section: The only way to guarantee recovery even after key components of the updated system have been damaged, is to keep an alternative system in place that is not touched by the update and can be used to recover the main system (will be referred to as recovery partition in the future).

Doing this requires some degree of communication between the OS and the bootloader. If the bootloader can notice that its default boot command fails to successfully boot an image, it can switch to an alternative boot command that targets the recovery partition. This recovery partition can automatically download the last version of the OS known to work on the device, install it and trigger a reboot again.

Generally if an embedded device has stopped working, the only interaction one can expect from almost every user is to turn it off and on again, which would (if the failed boot was detected by the bootloader) start the recovery partition. But the user, lacking a proper user interface, would be unable to distinguish between a second boot failure and a working recovery needing time to download a new image. Even worse a broken internet connection could prevent the recovery from working at all. To prevent all this, there is one preferable alternative to a recovery partition, if the flash capacity allows it. Use a copy of the operating system instead. This means that there are always two working versions of the operating system installed on the device. Every update the running system deletes the other one and replaces it with the newest version, before triggering a reboot into the other firmware. If it is successful, the new version continues to run until the next update, if not the old system is booted again and the system returns to its working state without needing time or internet access.

6. PoC on Zyxel NBG6716

After it has been settled how the update process should work, the only thing left is to implement it in practice. Amongst the three available hardware platforms, the Zyxel NGB6716 was chosen to serve as platform for the implementation of a proof-of-concept. It was chosen over the Linksys WRT1200 because that device already has a working autoupdate solution, that has strong similarities with the suggested update procedure and the goal is to demonstrate how automatic updates can be brought to a new platform. The TP Link Archer C7 was dismissed because it comes with a very old boot loader version and even though the sources are released under the GPL, reassembling the firmware from these sources was much harder than with Zyxel's sources.

While this chapter mostly documents the work done on only one device, many of the findings can be applied to other devices as well and the chapter is structured in a way that readers can use it as a blueprint to try and bring this update process to any other device as well.

6.1. Sysupgrade Script

Before diving into the details of implementing the update process, the available functionality of the already established manual upgrade process must be revisited. It has already been discussed on a superficial level in chapter 3, but now a more technical analysis of this piece of software is required.

Generally spoken the sysupgrade program is a shell script, that takes many different arguments as input. These inputs can be split into two different categories, one for creating backups and one for doing upgrades.

Listing 6.1: Help text of Sysupgrade script

```
Usage: $0 [<upgrade-option>...] <image file or URL>
       $0 [-q] [-i] <backup-command> <file>

upgrade-option:
   -d <delay>  add a delay before rebooting
   -f <config> restore configuration from .tar.gz (file or url)
   -i          interactive mode
   -c          attempt to preserve all changed files in /etc/
   -n          do not save configuration over reflash
   -T | --test
               Verify image and config .tar.gz but do not actually
                  flash.
   -F | --force
               Flash image even if image checks fail, this is
                  dangerous!
   -q          less verbose
   -v          more verbose
   -h | --help display this help

backup-command:
   -b | --create-backup <file>
               create .tar.gz of files specified in sysupgrade.conf
               then exit. Does not flash an image. If file is '-',
               i.e. stdout, verbosity is set to 0 (i.e. quiet).
   -r | --restore-backup <file>
               restore a .tar.gz created with sysupgrade -b
               then exit. Does not flash an image. If file is '-',
               the archive is read from stdin.
   -l | --list-backup
               list the files that would be backed up when calling
               sysupgrade -b. Does not create a backup file.
```

All functions connected to backups are more relevant to the recovery scheme, than to the actual update mechanism and the other options only allow deciding how much of the configuration should be carried over and how much output and user interaction the upgrade process should provide. Far more interesting at this point is the control flow of this utility when carrying out its tasks

along with the technical details how it is able to provide a common upgrade mechanism for all hardware platforms supported by OpenWRT.

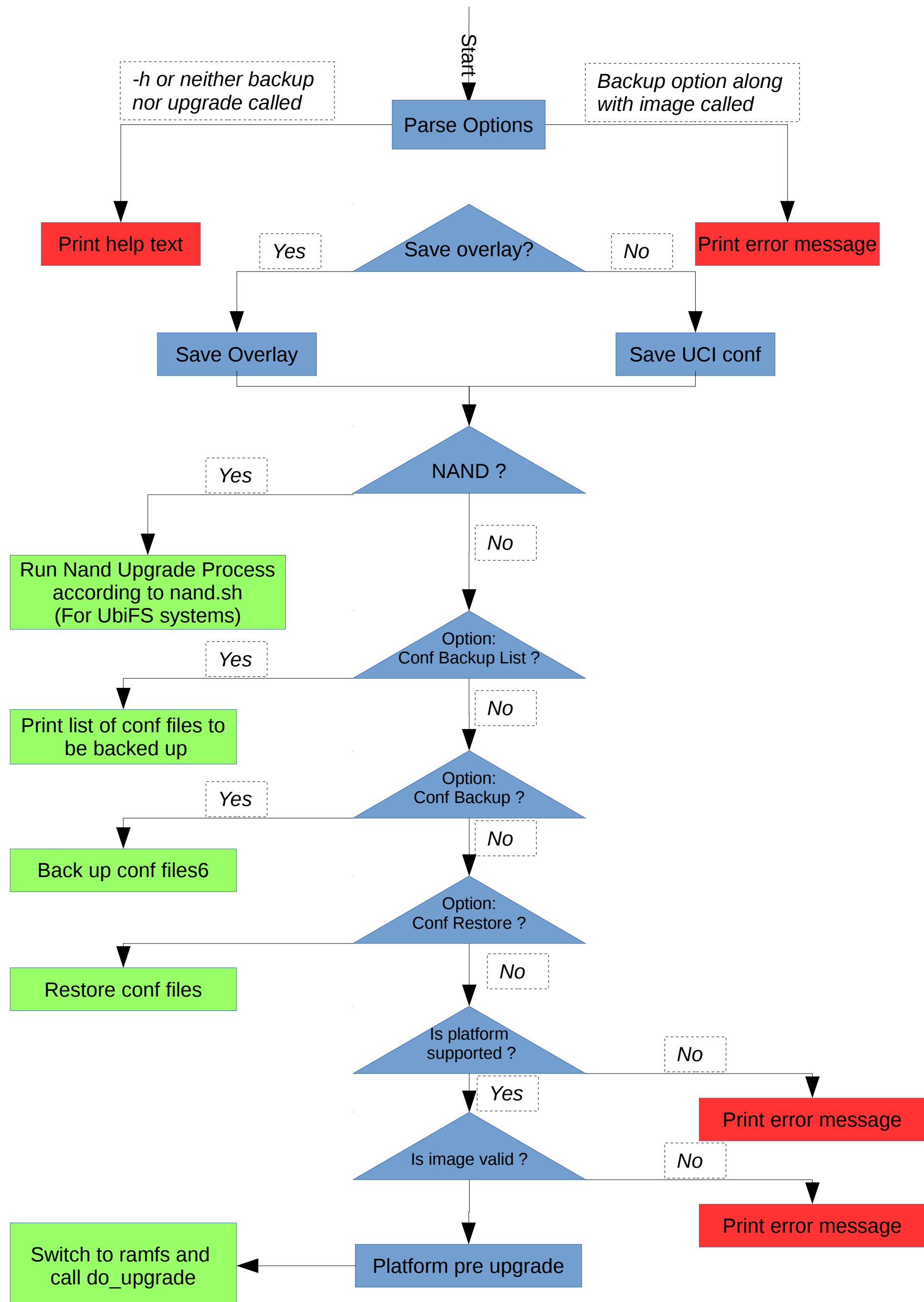
This is not achieved by customizing the sysupgrade script for every platform, as one might expect, but instead by implementing a generic upgrade function, that uses functions implemented in platform.sh, to handle platform specific tasks and settings. Figure 6.1 shows that a method called "platform_pre_upgrade" is executed before the "do_upgrade" method. Even before that a check is made if the current platform is supported at all, but as listed above, this check can be ignored by calling sysupgrade with the "–force" option.

The upgrade method itself is worth taking a look at, mainly because it is simpler than one would expect:

Listing 6.2: do_upgrade methods

```
1  default_do_upgrade() {
2     sync
3     if [ "$SAVE_CONFIG" -eq 1 ]; then
4        get_image "$1" "$2" | mtd $MTD_CONFIG_ARGS -j "$CONF_TAR"
            write - "${PART_NAME:-image}"
5     else
6        get_image "$1" "$2" | mtd write - "${PART_NAME:-image}" #
7     fi
8  }
9
10 do_upgrade() {
11    if type 'platform_do_upgrade' >/dev/null 2>/dev/null; then
12       platform_do_upgrade "$ARGV"
13    else
14       default_do_upgrade "$ARGV"
15    fi
16
17    if [ "$SAVE_CONFIG" -eq 1 ] && type 'platform_copy_config'
         >/dev/null 2>/dev/null; then
18       platform_copy_config
19    fi
20
21    v "Upgrade completed"
22    [ -n "$DELAY" ] && sleep "$DELAY"
```

Figure 6.1.: Sysupgrade flow diagram



```
23      ask_bool 1 "Reboot" && {
24         v "Rebooting system..."
25         reboot -f
26         sleep 5
27         echo b 2>/dev/null >/proc/sysrq-trigger
28      }
29   }
```

In the default case the image is simply loaded into RAM, then written directly to the flash using the Linux mtd utility. If a platform specific implementation exists, this implementation is called instead. Considering the many different platforms supported by OpenWRT, it is quite surprising that the majority of boards does not require any platform specific modifications. To give a rough estimate, "platform.sh" lists more than 180 supported boards and only about 30 of them do not use the default_do_upgrade method. So it should come as no surprise that none of the devices analysed for this paper, despite their differences, requires a custom upgrade procedure.

Careful readers may have noticed that there is one alternative upgrade path, that has not been discussed so far, namely the one for nand devices, that relies on it's own set of scripts found in "nand.sh". A quick investigation reveals that the name is slightly missleading, as these functions are not designed to update any device using a NAND flash chip, but only devices running UBI on it. As already mentioned UBI is a relatively new technology and apparently there have been some issues integrating it into the ordinary sysupgrade procedure. That is at least indicated by the fact, that the path for updating such devices diverges from the main update path before even verifying that the current platform is supported.

The main lesson learned from this investigation is, that the sysupgrade utility provides support for a multitude of platforms, thanks to hooks and optional platform specific methods it can easily be modified to handle tasks differently and should be able to meet all requirements towards an upgrade mechanism. On the other hand however for the majority of devices it would suffice to support only the default upgrade implementation, so for the requirements of this paper it would be sufficient to find a method compatible with all devices using the "default_do_upgrade" method.

6.2. Unlock U-Boot

As discussed in section 5.1.3 the chosen recovery scheme requires communication between the bootloader and OpenWRT. Usually such communication is handled via a U-Boot environment that can be modified by both the bootloader and the operating system. Unfortunately Zyxel has chosen to write-protect this area, a design choice made by many other vendors as well. For some devices the environment partition has even been removed entirely. Therefore a precondition for implementing the suggested upgrade process is gaining write permissions to the u-boot environment.

6.2.1. Gain Write Permissions on U-Boot Environment

Usually write protection is implemented with a single command on the kernel command line, defining the partition layout on the flash chip. This parameter is passed by U-Boot to the linux kernel and to no surprise can be configured within the U-Boot environment. This snippet from the environment of the NBG6716 shows how boot the ”u-boot”, ”env” and ”RFdata” partitions are protected by the ”readonly” parameter.

```
readonly=ro
setmtdparts=setenv mtdparts
    mtdparts=spi0.0:${ldr_psize}(u-boot)${readonly},
    ${env_psize}(env)${readonly}, ${rfdat_psize}(RFdata)${readonly},
    -(nbu)\; ath79-nand:${rfsdat_psize}(rootfs_data),
    ${romd_psize}(romd), ${hdr_psize}(header), ${rfs_psize}(rootfs),
    ${hdr1_psize}(header1), ${rfs1_psize}(rootfs1),
    ${bu1_psize}(bu1),-(bu2)
```

This of course means that this write protection is only active after the operating system has been booted, so it is possible to use a serial connection in order to gain access to the U-Boot command line and remove the readonly flag. While this solution works nicely for development, it is no practicable approach for ordinary users, creating a dilemma: The only way to gain write access from OpenWRT to the U-Boot environment is by changing a value within that very U-Boot environment from OpenWRT.

Fortunately there is a way to work around this problem. In section 4.2.2 a whole subsection was dedicated to the fact that the chosen partition layout for this device is quite questionable and it is not the only platform where the bootloader passes a useless partition layout. The OpenWRT community could have tried to gain access to bootloaders of all these devices and change the memory layout passed to the kernel, but this would have required a major effort and made the installation of an alternative firmware much harder. So instead they chose to put a hardcoded partition layout into the kernel, ignoring the information provided by the bootloader.

This means that it is possible to prepare an alternative firmware, which grants OpenWRT write access to the locked partitions. And because this is already done for many platforms, it is supported by the OpenWRT toolchain.

Listing 6.3: From /target/linux/ar71xx/Makefile

```
zcn1523h_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 6208k(rootfs), 1472k(kernel),
    64k(configure)ro, 64k(mfg)ro, 64k(art)ro, 7680k@0x50000(firmware)
mynet_n600_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 64k(devdata)ro, 64k(devconf)ro,
    15872k(firmware), 64k(radiocfg)ro
mynet_rext_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    7808k(firmware), 64k(nvram)ro, 64k(ART)ro
zyx_nbg6716_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(env)ro,
    64k(RFdata)ro, -(nbu);ar934x-nfc:2048k(zyxel_rfsd), 2048k(romd),
    1024k(header), 2048k(kernel), -(ubi)
qihoo_c301_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env), 64k(devdata), 64k(devconf), 15744k(firmware),
    64k(warm_start), 64k(action_image_config),
    64k(radiocfg)ro;spi0.1:15360k(upgrade2), 1024k(privatedata)
```

Listing 6.3 shows a snippet taken from an OpenWRT Makefile, which contains the hardcoded flash layout, that is usually used for this device along with a few others. Simply deleting the letters ”ro” twice is enough to build an image with write access to the previously locked partitions.

Once the modified firmware has been installed, the environment can be easily changed and once the readonly flag has been removed from the *env* partition,

even the old firmware using the flash layout information provided by the bootloader has write permissions to the U-Boot environment.

6.2.2. Work around zloader

As already mentioned in chapter 4.2.2 Zyxel has gone beyong ordinary U-Boot modifications by implementing a custom binary blob referred to as zloader. While the source code for the modifications of U-Boot is public under the GPL, the sources of the zloader are not published, so its features and purposes can only be guessed based on observation. On the surface it only replaces the ordinary U-Boot command line with a custom one that provides a limited set of features, allowing only to continue the boot process, recover via tftp, show status information or enter a debug mode (which is password protected).

All of these feature are also provided by U-Boot itself, so this zloader does not provide any new functions. One could argue that it provides more convenient commands for mainstream users, but given the fact that mainstream users are very unlikely to solder pins to the serial connection in order to use it, that argument does not satisfy. Another theory that came to mind was that the zloader is used to validate the installed firmware and make sure that only firmwares released by Zyxel are booted. This theory was supported by the fact that the zloaders for different devices were not equal, but it turned out they can be switched between devices without any errors, so that theory seems wrong. Whatever the purpose of this zloader is, U-Boot still runs the ordinary U-Boot boot command specified in the environment, so it does not limit or change any functionality.

For the development process however, the missing access to the U-Boot command line turns out to be a massive limitation, as it renders testing of U-Boot commands much more tedious than it has to be. Fortunately it turned out that the sources provided by Zyxel include an option to build the bootloader without the zloader, granting direct access to the command line. Afterwards it was necessary to build a modified firmware again, that removed write protection for the "u-boot" partition. Then it was easy to replace the original bootloader with the new one without the zloader. In order to simplify the development process this alternative bootloader was installed, but as this is a risky operation

(a failure while switching the bootloader may render the device unusable) the suggested update process must work with the stock bootloader.

U-Boot chainloading

Given the fact that multiple issues were encountered with U-Boot, starting from outdated versions over vendor specific patches until custom injections like the zloader, the idea of chainloading the bootloader was investigated. Ideally this would leave the original bootlader and environment untouched and place a second bootloader instead of the old firmware kernel. This way there would be no risk of rendering the device unusable by damaging the bootloader and the version, features and bugs of the original bootloader would be irrelevant, as well as the write protection or lack of the original environment.

Unfortunately however U-Boot does not make any statements on chain loading, in practice there are some people doing it successfully and others reporting errors. The overall take away seems to be that it is possible to chainload the same U-Boot multiple times without errors, for different versions however the chances decrease the more different the versions are. In addition to that the command to load another U-Boot is different from the command to load a linux kernel, so even if it worked, it would require modifications to the original U-Boot's environment, which limits its usability.

Is is possible to do U-Boot chain loading on the NBG6716 if the same bootloader is used twice, it was even possible to chainload a version without zloader, but the attempt to load a newer U-Boot version failed (Note: It should be pointed out at this point, that no excessive effort was made to get it to work. Given the versatility of U-Boot it is probably possible to build a version, which can be chain loaded).

The idea was abandoned at this point, as it requires access to the U-Boot environment and does not allow an easy version upgrade. As long as U-Boot does not implement some features to extend their chain loading support, it is not recommended to use this approach to gain access to the bootloader.

6.3. Enable Dual-Boot

In order to implement the solution suggested in chapter 5.1.3 it is necessary to configure U-Boot to run two alternative systems. This allows booting a recovery partition in case the default system is broken or running two parallel systems entirely in order to allow easy switching between them. Given the fact that the partition layout already seems to support the second case, it was chosen for implementation.

6.3.1. Investigate Linksys WRT1200AC Setup

Before trying to support dual booting on the Zyxel device is makes sense to take a look at the hardware platform that already supports dual booting, the Linksys WRT1200AC.

Listing 6.4: U-Boot Environment parameters for Dual Boot

```
bootcmd=run altnandboot
nandboot=setenv bootargs console=ttyS0,115200 root=/dev/mtdblock5 ro
    rootdelay=1 rootfstype=jffs2 earlyprintk $mtdparts;nand read
    $defaultLoadAddr $priKernAddr $priKernSize; bootm
    $defaultLoadAddr
altnandboot=setenv bootargs console=ttyS0,115200 root=/dev/mtdblock7
    ro rootdelay=1 rootfstype=jffs2 earlyprintk $mtdparts;nand read
    $defaultLoadAddr $altKernAddr $altKernSize; bootm
    $defaultLoadAddr
mtdparts=mtdparts=armada-nand:2048K(uboot)ro, 256K(u_env),
    256K(s_env), 1m@9m(devinfo), 40m@10m(kernel), 34m@16m(rootfs),
    40m@50m(alt_kernel), 34m@56m(alt_rootfs), 80m@10m(ubifs),
    -@90m(syscfg)
defaultLoadAddr=0x2000000
priKernAddr=0x0a00000
priKernSize=0x0600000
altKernAddr=0x3200000
altKernSize=0x0600000
```

Listing 6.4 shows the parts of the U-Boot environment responsible for the dual boot support. This setup allows switching between operating systems by toggling the bootcmd value between "run nandboot" and "run altnandboot". Most interesting for the work on the Zyxel router are the differences between those two boot commands.

The bootargs they pass to the Linux kernel are almost identical, only the partition of the rootfs changes, which makes sense, since the two systems have to use different root partitions. Then they both load the Linux kernel from different flash locations to the same memory segment in RAM before continuing with the bootm command to start the boot process.

This shows that U-Boot can be modified to support booting two different versions of OpenWRT without need for any special U-Boot features or customizations.

6.3.2. Implementing Dual Boot on the Zyxel NBG6716

Considering the lessons learned in the previous section and the partition layout of the Zyxel device it seems rather easy to install a second system and boot in in parallel. At first a second version of the firmware was installed on the partition mtd9, which had been completely empty in the original firmware.

Listing 6.5: Zyxel U-Boot Environment parameters

```
loadaddr=80400000
readonly=ro
setmtdparts=setenv mtdparts
    mtdparts=spi0.0:${ldr_psize}(u-boot)${readonly},
    ${env_psize}(env)${readonly}, ${rfdat_psize}(RFdata)${readonly},
    -(nbu)\; ath79-nand:${rfsdat_psize}(rootfs_data),
    ${romd_psize}(romd), ${hdr_psize}(header), ${rfs_psize}(rootfs),
    ${hdr1_psize}(header1), ${rfs1_psize}(rootfs1),
    ${bu1_psize}(bu1),-(bu2)
flashargs=setenv bootargs board=NBG6716 root=/dev/mtdblock7
    rootfstype=jffs2 noinitrd ${bootmode} ${zld_ver}
addtty=setenv bootargs ${bootargs} console=ttyS0,${baudrate}
addmtd=setenv bootargs ${bootargs} ${mtdparts}
```

```
boot_flash=run setmtdparts flashargs addtty addmtd;fsload
    ${loadaddr} /boot/vmlinux.lzma.uImage;bootm ${loadaddr}
bootcmd=run boot_flash
```

The first attempt to boot this firmware was made by altering the flashargs value to use a different root partition. The kernel load parameters were not changed because the fsload command does not accept any input about where to start searching. So the first attempt failed because fsload still loaded the kernel from the original file system instead of the alternative one.

In order to fix this problem it was necessary to change to fsload command to make it load the kernel from the correct file system. But the only argument supported by this command is the location where the loaded file should be stored in RAM and the name of the file. So the next attempt was to slightly modify the second firmware by renaming its kernel file to "vmlinux_alt.lzma.uImage". In theory this should have enabled U-Boot to find the alternative kernel stored on the alternative file system. Unfortunately however this did not work as expected, because the fsload command failed to load the alternative kernel.

Some experimentation showed that the fsload command fails whenever it is called with any other arguments than the ones specified in the original U-Boot environment. The reason for this can probably be found in one of the patches written by Zyxel for their device, called:
"008-zyxel_improve_performace_for_fsload_command.patch".
The exact functionality of this patch was not investigated, but nevertheless a reasonable assumption can be made, why it would be causing problems: Usually loading files from a JFFS2 partition would pose a performance problem, because JFFS2 requires loading and scanning the entire file system before it can be searched. The patch speeds up that process but at the cost of not really searching the entire file system. Instead the patch implements a custom method that loads the kernel by analysing JFFS2 nodes and file system specific summaries. Even without a closer investigation this implementation seems like a hack that apparently fails as soon as a single input changes.

Since fixing the fsload command would require modifying the U-Boot, which is not practicable for the majority of users, alternative ways to load the kernel from memory were needed. Fortunately the U-Boot provided by Zyxel does support an alternative method by directly reading from the flash memory. A

second helpful fact was that Zyxel defined header partitions for each rootfs partitions. Both header partitions are unused by the operating system and conveniently sized to store a kernel file. So for the next attempt the kernel image was written to that header partition using the dd command. Now finally the alternative systems kernel could be loaded and the boot process completed without issue.

Listing 6.6: Zyxel U-Boot Environment modifications

```
flashargs2=setenv bootargs board=NBG6716 root=/dev/mtdblock9
    rootfstype=jffs2 noinitrd
loadKernel2=nand read 0x80400000 0x2300000 0x100000
boot_flash2=run setmtdparts flashargs2 addtty addmtd
    loadKernel2;bootm ${loadaddr}
bootcmd=run boot_flash2
```

Listing 6.6 shows all modifications necessary to the U-Boot environment in order to support the dual boot process. At this point only a single variable must be changed in order to switch between booting the original firmware or the alternative one.

6.4. Enable Recovery

Now that it is possible to boot two different systems on the device, it is necessary to define an update procedure that enables the device to recover from any error caused by the update process. This starts with errors during the download resulting in a damaged image file, continues with write errors while installing the new image and ends with external disruption like power outages during the update. Furthermore every update carries the risk of introducing new unexpected errors. The verification of the downloaded image will be postponed to section 6.5 because from a recovery perspective it does not matter why the content of the flash is faulty.

Assuming that our update process only touches the U-Boot environment and either the rootfs or the rootfs1 partition, the possible errors caused through an update can be grouped in these scenarios:

1. The system encounters an error during the boot process. System does not boot any longer.
2. The system boots successfully but key functions like networking or remote access do not work.
3. The system boots successfully but side packages like ad-blockers do no longer work.
4. The U-Boot Environment is invalid and U-Boot is forced to use its default environment.

Without a serial connection to the device in order to read the bootlog, even an experienced user would be unable to differentiate those scenarios from a hardware failure. But as already described in section 5.1.3 there is one user interaction we can expect even from unskilled users: Turning the device off and on again.

The suggested recovery scheme always works by switching back to the previously installed system, so it only supports the scenarios 1-3, so these will be investigated first:

6.4.1. Switch to alternative firmware at next boot

In order to switch between the original and the alternative firmware a single variable in the U-Boot environment must be changed. In order to support scenario 1 where the system attempts to boot the new firmware, but fails during the boot process, a very simple trick is used. The boot command is extended to automatically toggle the firmware run at next boot. This way if the boot fails the next boot is guaranteed to run the previous working version again. Listing 6.7 shows the changes needed in the U-Boot environment in order to implement this behaviour.

Listing 6.7: Zyxel U-Boot with recovery

```
boot_flash=setenv bootcmd boot_flash2;saveenv;run setmtdparts
    flashargs addtty addmtd;fsload ${loadaddr}
    /boot/vmlinux.lzma.uImage;bootm ${loadaddr}
boot_flash2=setenv bootcmd boot_flash;saveenv;run setmtdparts
    flashargs2 addtty addmtd loadKernel2;bootm ${loadaddr}
bootcmd=run boot_flash
```

This solution however still has the disadvantage that every reboot switches between firmwares, even if there was no error. So this approach still needs to be refined.

6.4.2. Only switch to alternative firmware if there is an error

If the router runs for long periods of time the solution suggested in the previous section can already be used. If the device is rebooted regularly, for example by a central switch used to turn off power over night, it would cause the device to always return to the older firmware, even if there is nothing wrong with the current one.

This issue can easily be avoided by defining a script that is run after the boot process has been completed. This script checks that key functions of the system are running as expected and updates the boot command again if this is the case. This supports recovery scenarios 1 and 2 but by only checking key components the automatic recovery now ignores the third recovery scenario where only side packages do no longer function. This decision was made because the two goals of recovering from any possible problem and not recovering if there is no problem are conflicting and every solution will be a compromise between them. For the prototype implemented for this paper, a very simple startup script was created that only checks if an internet connection can be established and if the user can still access the device via ssh and http.

Listing 6.8: Startup Script to check recovery status

```
1  #!/bin/sh
2
3  checkErrorCode()
4  {
5          local _retVal=""
6          if [[ $1 -eq "0" ]]; then
7                  _retVal=1
8          else
9                  _retVal=0
10         fi
11         echo $_retVal
12 }
```

```
13
14 #give system time to set up
15 sleep 5m
16
17 #Check if System is online
18 wget -q --tries=10 --timeout=20 --spider http://google.com
19 isOnline=$(checkErrorCode $?)
20 #Check if webserver is running
21 ps | grep uhttpd | grep -v grep
22 httpRunning=$(checkErrorCode $?)
23 #Check if ssh server is running
24 ps | grep dropbear | grep -v grep
25 sshRunning=$(checkErrorCode $?)
26
27 currentBootCmd=$(fw_printenv -n bootcmd)
28
29 if $currentBootCmd EQUALS "run boot_flash"; then
30        newBootCmd="run boot_flash2";
31 else
32        newBootCmd="run boot_flash";
33 fi
34
35 if [[ $sshRunning -eq 1 ]] && [[ $httpRunning -eq 1 ]] && [[
      $isOnline -eq 1 ]]; then
36        echo "Startup successful, switching boot command to
             $newBootCmd"
37        fw_setenv bootcmd $newBootCmd
38 else
39        echo "System Startup failed, going for Reboot"
40        reboot
41 fi
```

This decision to keep the script this simple was made on the assumption that most users only need the basic functions of this device. Should this assumption turn out to be invalid, this script can easily be extended to validate the function of other parts of the system. Actually it is expected that this script would be extended massively if this recovery scheme is used by a larger audience, but for the prototyping phase it was found to be sufficient.

6.4.3. Restore U-Boot environment

The last scenario involves problems caused through an invalid U-Boot environment. Considering the fact that the suggested solution requires regular write operations on the environment, it must be considered that the environment might be corrupted at times through an invalid write operation. In this case U-Boot returns to its default environment, which runs the boot_flash command. This is the main reason why the original boot_flash command was not modified to load the kernel the same way as boot_flash2, because it would prevent the default environment from booting successfully.

Therefore a corrupt U-Boot environment would not prevent the Zyxel device from booting. But it does prevent ever switching to the other system again, unless the Environment is restored. To make sure the environment is restored if necessary, a slight change to the startup script is necessary.

Listing 6.9: Extend Statup Script to check U-Boot Env

```
1 #Check if U-Boot environment is OK
2 if [[ $currentBootCmd -eq "Warning: Bad CRC, using default
     environment" ]]
3    mtd -r write /var/recovery/u_boot_env.backup mtd1
4 else
5    #Do Nothing
6 fi
```

This check validates that the current boot command could be read from the environment. If the check fails a new valid environment is flashed to mtd1 to restore dual boot support for further updates.

At this point there is one obvious scenario where the recovery scheme still fails. If the U-Boot environment is damaged by the update (for example by a power loss during the write operation) and the firmware installed on the default rootfs partition fails to boot for any reason, the implemented recovery strategy fails. So far no solution has been found that would allow recovery from such an event, it would require more testing and experience to see if this theoretical scenario might ever happen in practice.

6.5. Securing the update process

The last question to be discussed in detail refers to the security of the update procedure. Given the fact that the goal is to increase security by rolling out updates automatically, the entire endeavour would become useless if attackers could use the update mechanism to make the device install illegal updates. Note: As the OpenWRT project does not publish regular updates to their firmwares and testing security required the capability to simulate an attack, a simple webserver was set up to test delivering updates to clients.

In this context security refers mainly to integrity, not so much to privacy and availability. Considering the fact that Zyxel does package openssl and ”wget-ssl” with their firmware the first prototype simply downloaded the new firmware via a SSL secured http connection. To verify the download a separate request was made to get the expected hash of the downloaded file. Only if those match the image is written to the flash and the hash is used again to verify that the write process was successful before rebooting into the new partition.

This solution has the disadvantage that it relies on SSL, which is not available on all devices because it requires a significant amount of memory. The following alternative, inspired by OPKG, does not require encryption but still ensures that no attacker can inject his own updates. Every update is digitally signed by the provider of the firmware and the matching public key is packaged with the firmware. Therefore even if the image is downloaded insecurely, as long as the digital signature verification passes, the update is valid. Of course this also has its drawbacks because a lot of information about the device and the software running on it are disclosed to an attacker inspecting the network traffic.

6.6. Summary

At the end of this section it is time to remember the requirements define at the beginning in section 2.3.4. The fully automatic update trigger was not discussed indivdually in this chaper, because a simple cronjob to run a script to find, download and install the newest firmware version was not found to be interesting enough. Interested users can find the version used during the development process in the appendix.

Most of the implementation work focused on the essential feature of the suggested update procedure, the non-manual recovery scheme. The implementation of this scheme was also the main limiting factor to the number of devices supported by the suggested update procedure. It is limited to devices with enough space to hold two distinct OpenWRT systems and use U-Boot as bootloader. This may limit the number of devices more than intended, but it still supports a reasonable percentage of devices, so this requirement was fulfilled as well.

The integrity of the update process was also discussed in this chapter, for the device in question even a rather privacy friendly solution was found, which certainly meets the security requirement. This leaves only one requirement that was not addressed during implementation, namely that a user's work should never be interrupted by the update process. This does not mean that the suggested update procedure is incapable of fulfilling this requirement, it just means that the time did not suffice to implement it during the prototyping phase for this paper.

Ultimately it can be said that the successful implementation of the prototype proves that the suggested update process has the potential to safely and automatically upgrade embedded devices based on U-Boot and OpenWRT.

7. Future work

The final chapter of this thesis will discuss future work left to be done in order to make the suggested update procedure usable for a wider public. In addition to that some functions will be suggested to refine the presented prototype.

7.1. Prepare build environment

A task that was explicitly not part of this thesis, but is vital in order to provide automatic updates, is a build environment that automatically builds new firmware images based on the current master version of the OpenWRT repository and publishes them. Preferable this should be done by the OpenWRT project itself but in the past the project has not been very active in this regard. The most current firmware version for the Zyxel NBG6716 available for download was more than a year old at the time this was written. It should be pointed out however that the LEDE project, which is about to remerge with the OpenWRT project is building firmware updates more regularly, so there is a good chance that this situation will improve in the future.

Another possible improvement that should be discussed at this point is the attempt to make the builds of OpenWRT reproducible [9]. This would enable anyone to set up his own build environment and validate that the firmware provided by the update server is created from the unmodified sources published by the OpenWRT project. Such a feature would massively improve the trust into delivered updates and even allow users to verify the validity of an image by comparing it to images build by other update providers. While there is significant effort being put into this topic, the current reproducible jenkins build does not even create a single reproducible firmware yet, so this topic still requires a lot of work before it can be used.

7.2. Test produced images

Another strategy to prevent errors caused by updates would be to automatically test the firmware images, before they are released publicly. This could be achieved by having one test device for every supported device type that is updated before all others. Afterwards the features of the test device could be tested automatically from a central test instance. The following list gives a short list of possible tests:

- Check if the device can still be reached via HTTP and SSH.
- Check if the device could still establish an internet connection via all supported methods (DHCP, VPN, LTE, ...).
- Check if a client can still connect to the wireless network and that the performance of the wireless network has not decreased.
- Check that all expected services are running.

Naturally that list was mainly inspired by routers and would need some specification towards the individual firmware under test. But it would certainly increase the reliability of an image for the majority of users, which hardly deviates from the suggested configuration. Finally it would effectively prevent errors like the scenario described in the last paragraph of section 6.4.3, because the odds of the boot process failing due to unusual configuration changes made by a user are minimal.

7.3. Prevent user disruption

As already discussed in section 2.3.4, the update process should not trigger, while a user is using the device. In theory it should be trivial for an embedded device to determine, if it is being used or not, but the example of a router proves the opposite. Of course there are trivial attempts to test this, like checking if there are any active DHCP leases or network interfaces. But in most cases such solutions will never report an unused system, because there is always at least one device in a house, like a phone or a printer, that is not turned off and therefore keeps an active network connection.

A more promising idea would be to measure network traffic and active connections instead of active network devices. This has the downside that even devices not currently in use do create a certain amount of network traffic through features like network discovery and some devices are managed through the cloud or receive push notifications and therefore keep a TCP connection open permanently. Looking at all these possible ways to detect user activity and the potential issues linked with all of them, one has to accept that this topic requires more work and effort in order to provide a reasonable prototype.

Independent of how user activity is checked the goal of preventing user disruption must never be able to permanently prevent an update. So if a device is busy for too long, the update should eventually be triggered, even if it means disrupting the user.

7.4. Create minimal recovery partition

The prototype presented in the previous chapter used two distinct partitions to store the same firmware twice on one device. But many devices currently supported by OpenWRT do not have the necessary flash size to store the full operating system twice. For these cases it is necessary to prepare a recovery partition, which automatically downloads and reinstalls the last working firmware version.

Such a firmware should be stripped of every functionality not required for its task. It needs to connect to the internet in order to download a working firmware version, but other things, even basic functionality for a router like WLAN or a DHCP server are not necessary for this image.

OpenWRT has been ported successfully to devices with as little as 2MB of memory and this should be seen as an upper limit for a recovery partition. If the size can be contained to this degree the recovery partition should fit on every OpenWRT device with at least 8MB of available flash memory. Devices with less memory will not be supported by the suggested update procedure unless a more memory efficient recovery scheme is found.

For this paper some alternative ideas like extending U-Boot with IP/TCP capabilities to download the image directly or creating a U-Boot standalone

application to handle the recovery were investigated and dismissed as infeasible. The only possible option to extend the recovery scheme to devices with less than 8MB of flash memory would be using an external storage like a USB stick or a SD card. If the router provides the needed interface and U-Boot has been compiled with the commands necessary to access that storage, it could be used to restore the firmware directly within U-Boot.

7.5. Track necessary recoveries

Another feature still missing the the prototype is tracking of successful and failed updates. The prototype is capable of recovering from a failed update, but that would not stop him from retrying the update again. If the update failed due to a non reproducible error, this is a correct behaviour, but if the update itself is faulty it would be installed and recovered during every update cycle until a newer update is released.

That alone would be acceptable, especially if the updates are optimized to not disrupt a user's work. But tracking recoveries is also relevant for the provider of an update. If a new firmware is published and more than 90% of the devices downloading it revert the update right after applying it, the update is faulty. To enable providers of firmware updates to quickly respond to problems caused by a new update, a feedback mechanism about an update is necessary, preferably one that identifies a device as well as its current firmware version and any firmware versions that were reverted by that device. This feature could easily be extended to provide more useful information, like the distribution of running firmware versions or how many devices are already running at the newest firmware and how long it usually takes until an update has been delivered to all target devices.

7.6. Bring update procedure to more devices

The final and most obvious future work is porting the suggested update procedure to more hardware platforms and see if any unexpected difficulties are

encountered there. Just by thinking about the two other devices analysed in the hardware chapter a few possible issues come to mind:

The TP-Link Archer C7 does not have a U-Boot environment by default, so a modified image containing an environment must be flashed in order to fix this. But there is no guarantee that the U-Boot installed on that device does support an environment at all, since it is not needed in general. If that should be the case, this device could not support the suggested update procedure without building and installing a new bootloader, which is an action that carries the risk of bricking the device.

The Linksys device already supports a similar update mechanism, so the implementation will meet less problems there. But it should be considered that the chosen partition layout is slightly confusing with partition within partitions and it would take some time to determine how to best install a new firmware without risking any damage to the file system.

8. Conclusion

The defined goal of this thesis was to suggest a solution how to provide automatic updates to embedded devices owned by home users. The chapters on the state of the art and hardware should provide an insight into the already established update procedures, their strengths and weaknesses as well as the environment they are targeting.

For developers of embedded devices, who intend to provide automatic updates to their product, this paper should provide a good starting point to take a look at existing solutions in order to find, which one is best suited for their specific use case. Should they find that their strategy selection is similar to the one presented in chapter 5 the implementation steps outlined in chapter 6 are a good starting point.

Beyond that there is the hope that the community behind OpenWRT will be able to base a community driven automatic update process on the suggested solution. It was demonstrated that in theory a vast majority of OpenWRT routers could be supported by the suggested solution and if a process is started that leads to all these devices receiving updates automatically, it would be a huge success for this paper and the security of thousands of devices.

Bibliography

[1] Open Handset Alliance. *OTA Updates*. URL: https://source.android.com/devices/tech/ota/ (visited on 06/19/2017) (cit. on pp. 24, 25).

[2] Open Handset Alliance. *Platform Versions*. URL: https://developer.android.com/about/dashboards/index.html (visited on 06/19/2017) (cit. on p. 26).

[3] Dima Bekerman Ben Herzberg and Igal Zeifman. *Breaking Down Mirai: An IoT DDoS Botnet Analysis*. URL: https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html (visited on 07/29/2017) (cit. on p. 2).

[4] Freifunk Community. *Freifunk Firmware Gluon*. URL: https://wiki.freifunk.net/Freifunk_Firmware_Gluon (visited on 04/01/2017) (cit. on pp. 29, 30).

[5] John Crispin. *openwrt and lede - remerge proposal V3*. URL: http://lists.infradead.org/pipermail/lede-dev/2017-May/007771.html (visited on 07/30/2017) (cit. on p. 8).

[6] Wolfgang Denk. *U-Boot Manual*. URL: https://www.denx.de/wiki/DULG/Manual (visited on 07/13/2017) (cit. on pp. 10, 11).

[7] Wolfgang Denk. *U-Bootdoc History*. URL: http://www.denx.de/wiki/view/U-Bootdoc/History (visited on 03/08/2017) (cit. on p. 9).

[8] Bedřich Košata. *Turris Omnia - Opensource SOHO router*. URL: https://archive.fosdem.org/2016/schedule/event/turrisopensourcerouter (visited on 04/02/2017) (cit. on p. 31).

[9] Holger Levson et al. *OpenWRT - Reproducible Wireless Freedom ?* URL: https://tests.reproducible-builds.org/openwrt/openwrt.html (visited on 07/26/2017) (cit. on p. 71).

[10] Linksys. *Creating a Linksys cloud account.* URL: https://www.linksys.com/us/support-article?articleNum=143662 (visited on 07/30/2017) (cit. on p. 34).

[11] Heather J. Meeker. *Open Source and the Legend of Linksys.* 2005. URL: http://www.linuxinsider.com/story/43996.html (visited on 03/08/2017) (cit. on p. 7).

[12] Allison Nixon, John Costello, and Z Wilkholm. *An after-action analysis of the mirai botnet attacks on dyn.* URL: https://www.flashpoint-intel.com/blog/cybercrime/action-analysis-mirai-botnet-attacks-dyn/ (visited on 07/29/2017) (cit. on p. 2).

[13] OpenWRT. *About OpenWRT.* URL: https://wiki.openwrt.org/about (visited on 03/08/2017) (cit. on p. 7).

[14] OpenWRT. *Filesystems.* URL: https://wiki.openwrt.org/doc/techref/filesystems (visited on 06/21/2017) (cit. on p. 47).

[15] Ralf. *Were 900k Deutsche Telekom routers compromised by Mirai?* URL: https://comsecuris.com/blog/posts/were_900k_deutsche_telekom_routers_compromised_by_mirai/ (visited on 07/29/2017) (cit. on pp. 2, 3).

[16] Thomas Reinhold. *Nachlese zum "Hackerangriff" auf das Telekom-Netz: Analyse und Implikationen.* URL: https://cyber-peace.org/2016/12/02/nachlese-zum-hackerangriff-auf-das-telekom-netz-analyse-und-implikationen/ (visited on 07/29/2017) (cit. on pp. 2, 3).

[17] Michal Vaner. *Automatic updates on the Turris routers.* URL: http://openwrtsummit.org/slides/automatic-updates.pdf (visited on 03/20/2017) (cit. on p. 33).

chap

Appendix A.

U-Boot Environments

TP Link Archer C7

As already stated in section 4.1, this device does not come with a real U-Boot environment, the parameters are hardcoded into the U-Boot binary image. Instead of manually extracting the information from the binary, the information provided by the OpenWRT community (see https://wiki.openwrt.org/toh/tp-link/tl-wdr7500) was used:

```
bootargs=console=ttyS0,115200 root=31:02 rootfstype=jffs2
   init=/sbin/init mtdparts=ath-nor0:256k(u-boot) ,64k(u-boot-env),
   6336k(rootfs), 1408k(uImage), 64k(mib0),64k(ART) bootcmd=bootm
   0x9f020000 bootdelay=1 baudrate=115200
ethaddr=0xba:0xbe:0xfa:0xce:0x08:0x41
ipaddr=192.168.1.111 serverip=192.168.1.100
dir=
lu=tftp 0x80060000 ${dir}u-boot.bin&&erase 0x9f000000
   +$filesize&&cp.b $fileaddr 0x9f000000 $filesize
lf=tftp 0x80060000 ${dir}ap135${bc}-jffs2&&erase 0x9f050000
   +0x630000&&cp.b $fileaddr 0x9f050000 $filesize
lk=tftp 0x80060000 ${dir}vmlinux${bc}.lzma.uImage&&erase 0x9f680000
   +$filesize&&cp.b $fileaddr 0x9f680000 $filesize
stdin=serial
stdout=serial
stderr=serial
ethact=eth0
```

Zyxel NBG6716

```
baudrate= 115200
ipaddr=192.168.1.1
serverip=192.168.1.33
uboot_env_ver=1.5
img_prefix=nbg6716-
loadaddr=80400000
readonly=ro
setmtdparts=setenv mtdparts
    mtdparts=spi0.0:${ldr_psize}(u-boot)${readonly},
    ${env_psize}(env)${readonly}, ${rfdat_psize}(RFdata)${readonly},
    -(nbu)\; ath79-nand:${rfsdat_psize}(rootfs_data),
    ${romd_psize}(romd), ${hdr_psize}(header), ${rfs_psize}(rootfs),
    ${hdr1_psize}(header1), ${rfs1_psize}(rootfs1),
    ${bu1_psize}(bu1), -(bu2)
flashargs=setenv bootargs board=NBG6716 root=/dev/mtdblock7
    rootfstype=jffs2 noinitrd ${bootmode} ${zld_ver}
addtty=setenv bootargs ${bootargs} console=ttyS0,${baudrate}
addmtd=setenv bootargs ${bootargs} ${mtdparts}
boot_flash=run setmtdparts flashargs addtty addmtd;fsload
    ${loadaddr} /boot/vmlinux.lzma.uImage;bootm ${loadaddr}
bootcmd=run boot_flash
ldr_paddr=0x9f000000
ldr_psize=0x40000
env_paddr=0x9f040000
env_psize=0x10000
rfdat_paddr=0x9f050000
rfdat_psize=0x10000
rfsdat_paddr=0
rfsdat_psize=0x200000
romd_paddr=0x200000
romd_psize=0x200000
hdr_paddr=0x400000
hdr_psize=0x100000
rfs_paddr=0x500000
rfs_psize=0x1E00000
hdr1_paddr=0x2300000
hdr1_psize=0x100000
```

```
rfs1_paddr=0x2400000
rfs1_psize=0x1E00000
bu1_paddr=0x4200000
bu1_psize=0x200000
lu=tftp ${loadaddr} ${dir}u-boot.bin;zflash erase ${ldr_paddr}
    ${ldr_psize};zflash write ${fileaddr} ${ldr_paddr} ${filesize}
lf=tftp ${loadaddr} ${dir}${img_prefix}rootfs.jffs2;nand erase
    ${rfs_paddr} ${rootfs_psize};nand write ${fileaddr} ${rfs_paddr}
    ${filesize}
hostname=NBG6716
countrycode=E1
ethaddr=E8:37:7A:7A:20:B8
serialnum=S150A41002908
```

Linksys WRT1200AC

```
CASset=max
MALLOC_len=5
MPmode=SMP
SMT-2D=NGVIF2202CF8X01
altFwSize=0x2800000
altKernAddr=0x3200000
altKernSize=0x0600000
altnandboot=setenv bootargs console=ttyS0,115200 root=/dev/mtdblock7
    ro rootdelay=1 rootfstype=jffs2 earlyprintk $mtdparts;nand read
    $defaultLoadAddr $altKernAddr $altKernSize; bootm
    $defaultLoadAddr
autoload=no
baudrate=115200
boot_order=hd_scr usb_scr mmc_scr hd_img usb_img mmc_img pxe net_img
    net_scr
bootargs_dflt=$console $nandEcc $mtdparts $bootargs_root
    nfsroot=$serverip:$rootpath ip=$ipaddr:$serverip$bootargs_end
    $mvNetConfig video=dovefb:lcd0:$lcd0_params
    clcd.lcd0_enable=$lcd0_enable clcd.lcd_panel=$lcd_panel
bootargs_end=:10.4.50.254:255.255.255.0:Armada38x:eth0:none
bootargs_root=root=/dev/nfs rw
```

```
bootcmd_auto=stage_boot $boot_order
bootcmd_fdt=tftpboot 0x2000000 $image_name;tftpboot $fdtaddr
   $fdtfile;setenv bootargs $console $nandEcc $mtdparts
   $bootargs_root nfsroot=$serverip:$rootpath
   ip=$ipaddr:$serverip$bootargs_end $mvNetConfig
   video=dovefb:lcd0:$lcd0_params clcd.lcd0_enable=$lcd0_enable
   clcd.lcd_panel=$lcd_panel; bootz 0x2000000 - $fdtaddr;
bootcmd_fdt_boot=tftpboot 0x2000000 $image_name; setenv bootargs
   $console $nandEcc $mtdparts $bootargs_root
   nfsroot=$serverip:$rootpath ip=$ipaddr:$serverip$bootargs_end
   $mvNetConfig video=dovefb:lcd0:$lcd0_params
   clcd.lcd0_enable=$lcd0_enable clcd.lcd_panel=$lcd_panel; bootz
   0x2000000 - $fdtaddr;
bootcmd_fdt_edit=tftpboot $fdtaddr $fdtfile; fdt addr $fdtaddr;
   setenv bootcmd $bootcmd_fdt_boot
bootcmd_lgcy=tftpboot 0x2000000 $image_name; setenv bootargs
   $bootargs_dflt; bootm 0x2000000;
bootdelay=3
cacheShare=no
console=console=ttyS0,115200
defaultLoadAddr=0x2000000
device_partition=0:1
disaMvPnp=no
eeeEnable=no
enaClockGating=no
enaCpuStream=no
enaFPU=yes
enaMonExt=no
enaWrAllo=no
eth1addr=00:50:43:00:08:01
eth1mtu=1500
eth2addr=00:50:43:00:00:01
eth2mtu=1500
eth3addr=00:50:43:08:00:00
eth3mtu=1500
ethact=egiga0
ethaddr=C0:56:27:B8:D2:B9
ethmtu=1500
ethprime=egiga0
```

```
fdt_addr=2040000
fdt_skip_update=no
fdtaddr=0x1000000
fdtfile=armada-38x-modular.dtb
firmwareName=caiman.img
flash_alt_image=tftpboot $defaultLoadAddr $firmwareName && nand
    erase $altKernAddr $altFwSize && nand write $defaultLoadAddr
    $altKernAddr $filesize
flash_pri_image=tftpboot $defaultLoadAddr $firmwareName && nand
    erase $priKernAddr $priFwSize && nand write $defaultLoadAddr
    $priKernAddr $filesize
ide_path=/
image_name=uImage
initrd_name=uInitrd
ipaddr=192.168.1.1
kernel_addr_r=2080000
lcd0_enable=0
lcd0_params=640x480-16@60
lcd_panel=0
loadaddr=0x02000000
loads_echo=0
mtddevname=uboot
mtddevnum=0
mtdids=nand0=armada-nand
mtdparts=mtdparts=armada-nand:2048K(uboot)ro,256K(u_env),256K(s_env),1m@9m(devinfo),40m@10m(
mvNetConfig=mv_net_config=4,(00:50:43:11:11:11,0:1:2:3),mtu=1500
mv_pon_addr=00:50:43:01:00:00
nandEcc=nfcConfig=4bitecc
nandboot=setenv bootargs console=ttyS0,115200 root=/dev/mtdblock5 ro
    rootdelay=1 rootfstype=jffs2 earlyprintk $mtdparts;nand read
    $defaultLoadAddr $priKernAddr $priKernSize; bootm
    $defaultLoadAddr
netbsd_en=no
netmask=255.255.255.0
netretry=no
partition=nand0,0
pcieTune=no
pexMode=RC
priFwSize=0x2800000
```

```
priKernAddr=0x0a00000
priKernSize=0x0600000
pxe_files_load=:default.arm-armadaxp-db:default.arm-armadaxp:default.arm
pxefile_addr_r=3100000
ramdisk_addr_r=2880000
rootpath=/srv/nfs/
sata_delay_reset=0
sata_dma_mode=yes
script_addr_r=3000000
script_name=boot.scr
serverip=192.168.1.254
standalone=fsload 0x2000000 $image_name;setenv bootargs $console
    $nandEcc $mtdparts root=/dev/mtdblock0 rw
    ip=$ipaddr:$serverip$bootargs_end; bootm 0x2000000;
stderr=serial
stdin=serial
stdout=serial
update_both_images=tftpboot $defaultLoadAddr $firmwareName && nand
    erase $priKernAddr $priFwSize && nand erase $altKernAddr
    $altFwSize && nand write $defaultLoadAddr $priKernAddr $filesize
    && nand write $defaultLoadAddr $altKernAddr $filesize
usb0Mode=host
usbActive=0
usbType=2
vxworks_en=no
yuk_ethaddr=00:00:00:EE:51:81
bootcmd=run altnandboot
boot_part_ready=3
boot_part=2
auto_recovery=yes
```

Appendix B.

Relevant OpenWRT Scripts

Contents of nand.sh

In Figure 6.1 a reference is made to nand.sh, its content are only summarized, so it is provided here to complete the description of possible update paths.

Listing B.1: /package/system/procd/files/nand.sh

```
#!/bin/sh
# Copyright (C) 2014 OpenWrt.org
#

. /lib/functions.sh

# 'kernel' partition on NAND contains the kernel
CI_KERNPART="kernel"

# 'ubi' partition on NAND contains UBI
CI_UBIPART="ubi"

ubi_mknod() {
   local dir="$1"
   local dev="/dev/$(basename $dir)"

   [ -e "$dev" ] && return 0

   local devid="$(cat $dir/dev)"
   local major="${devid%%:*}"
```

```
21    local minor="${devid##*:}"
22    mknod "$dev" c $major $minor
23 }
24
25 nand_find_volume() {
26    local ubidevdir ubivoldir
27    ubidevdir="/sys/devices/virtual/ubi/$1"
28    [ ! -d "$ubidevdir" ] && return 1
29    for ubivoldir in $ubidevdir/${1}_*; do
30       [ ! -d "$ubivoldir" ] && continue
31       if [ "$( cat $ubivoldir/name )" = "$2" ]; then
32          basename $ubivoldir
33          ubi_mknod "$ubivoldir"
34          return 0
35       fi
36    done
37 }
38
39 nand_find_ubi() {
40    local ubidevdir ubidev mtdnum
41    mtdnum="$( find_mtd_index $1 )"
42    [ ! "$mtdnum" ] && return 1
43    for ubidevdir in /sys/devices/virtual/ubi/ubi*; do
44       [ ! -d "$ubidevdir" ] && continue
45       cmtdnum="$( cat $ubidevdir/mtd_num )"
46       [ ! "$mtdnum" ] && continue
47       if [ "$mtdnum" = "$cmtdnum" ]; then
48          ubidev=$( basename $ubidevdir )
49          ubi_mknod "$ubidevdir"
50          echo $ubidev
51          return 0
52       fi
53    done
54 }
55
56 nand_get_magic_long() {
57    dd if="$1" skip=$2 bs=4 count=1 2>/dev/null | hexdump -v -n 4 -e
        '1/1 "%02x"'
58 }
```

```
59
60 get_magic_long_tar() {
61    ( tar xf $1 $2 -O | dd bs=4 count=1 | hexdump -v -n 4 -e '1/1
        "%02x"') 2> /dev/null
62 }
63
64 identify_magic() {
65    local magic=$1
66    case "$magic" in
67       "55424923")
68          echo "ubi"
69          ;;
70       "31181006")
71          echo "ubifs"
72          ;;
73       "68737173")
74          echo "squashfs"
75          ;;
76       "d00dfeed")
77          echo "fit"
78          ;;
79       "4349"*)
80          echo "combined"
81          ;;
82       *)
83          echo "unknown $magic"
84          ;;
85    esac
86 }
87
88
89 identify() {
90    identify_magic $(nand_get_magic_long "$1" "${2:-0}")
91 }
92
93 identify_tar() {
94    identify_magic $(get_magic_long_tar "$1" "$2")
95 }
96
```

```
97  nand_restore_config() {
98      sync
99      local ubidev=$( nand_find_ubi $CI_UBIPART )
100     local ubivol="$( nand_find_volume $ubidev rootfs_data )"
101     [ ! "$ubivol" ] &&
102         ubivol="$( nand_find_volume $ubidev rootfs )"
103     mkdir /tmp/new_root
104     if ! mount -t ubifs /dev/$ubivol /tmp/new_root; then
105         echo "mounting ubifs $ubivol failed"
106         rmdir /tmp/new_root
107         return 1
108     fi
109     mv "$1" "/tmp/new_root/sysupgrade.tgz"
110     umount /tmp/new_root
111     sync
112     rmdir /tmp/new_root
113 }
114
115 nand_upgrade_prepare_ubi() {
116     local rootfs_length="$1"
117     local rootfs_type="$2"
118     local has_kernel="${3:-0}"
119     local has_env="${4:-0}"
120
121     local mtdnum="$( find_mtd_index "$CI_UBIPART" )"
122     if [ ! "$mtdnum" ]; then
123         echo "cannot find ubi mtd partition $CI_UBIPART"
124         return 1
125     fi
126
127     local ubidev="$( nand_find_ubi "$CI_UBIPART" )"
128     if [ ! "$ubidev" ]; then
129         ubiattach -m "$mtdnum"
130         sync
131         ubidev="$( nand_find_ubi "$CI_UBIPART" )"
132     fi
133
134     if [ ! "$ubidev" ]; then
135         ubiformat /dev/mtd$mtdnum -y
```

```
136         ubiattach -m "$mtdnum"
137         sync
138         ubidev="$( nand_find_ubi "$CI_UBIPART" )"
139         [ "$has_env" -gt 0 ] && {
140            ubimkvol /dev/$ubidev -n 0 -N ubootenv -s 1MiB
141            ubimkvol /dev/$ubidev -n 1 -N ubootenv2 -s 1MiB
142         }
143      fi
144
145      local kern_ubivol="$( nand_find_volume $ubidev kernel )"
146      local root_ubivol="$( nand_find_volume $ubidev rootfs )"
147      local data_ubivol="$( nand_find_volume $ubidev rootfs_data )"
148
149      # remove ubiblock device of rootfs
150      local root_ubiblk="ubiblock${root_ubivol:3}"
151      if [ "$root_ubivol" -a -e "/dev/$root_ubiblk" ]; then
152         echo "removing $root_ubiblk"
153         if ! ubiblock -r /dev/$root_ubivol; then
154            echo "cannot remove $root_ubiblk"
155            return 1;
156         fi
157      fi
158
159      # kill volumes
160      [ "$kern_ubivol" ] && ubirmvol /dev/$ubidev -N kernel || true
161      [ "$root_ubivol" ] && ubirmvol /dev/$ubidev -N rootfs || true
162      [ "$data_ubivol" ] && ubirmvol /dev/$ubidev -N rootfs_data || true
163
164      # update kernel
165      if [ "$has_kernel" = "1" ]; then
166         if ! ubimkvol /dev/$ubidev -N kernel -s $kernel_length; then
167            echo "cannot create kernel volume"
168            return 1;
169         fi
170      fi
171
172      # update rootfs
173      local root_size_param
174      if [ "$rootfs_type" = "ubifs" ]; then
```

```
175         root_size_param="-m"
176     else
177         root_size_param="-s $rootfs_length"
178     fi
179     if ! ubimkvol /dev/$ubidev -N rootfs $root_size_param; then
180         echo "cannot create rootfs volume"
181         return 1;
182     fi
183 
184     # create rootfs_data for non-ubifs rootfs
185     if [ "$rootfs_type" != "ubifs" ]; then
186         if ! ubimkvol /dev/$ubidev -N rootfs_data -m; then
187             echo "cannot initialize rootfs_data volume"
188             return 1
189         fi
190     fi
191     sync
192     return 0
193 }
194 
195 nand_do_upgrade_success() {
196     local conf_tar="/tmp/sysupgrade.tgz"
197 
198     sync
199     [ -f "$conf_tar" ] && nand_restore_config "$conf_tar"
200     echo "sysupgrade successful"
201     reboot -f
202 }
203 
204 # Flash the UBI image to MTD partition
205 nand_upgrade_ubinized() {
206     local ubi_file="$1"
207     local mtdnum="$(find_mtd_index "$CI_UBIPART")"
208 
209     [ ! "$mtdnum" ] && {
210         CI_UBIPART="rootfs"
211         mtdnum="$(find_mtd_index "$CI_UBIPART")"
212     }
213 
```

```
214     if [ ! "$mtdnum" ]; then
215        echo "cannot find mtd device $CI_UBIPART"
216        reboot -f
217     fi
218 
219     local mtddev="/dev/mtd${mtdnum}"
220     ubidetach -p "${mtddev}" || true
221     sync
222     ubiformat "${mtddev}" -y -f "${ubi_file}"
223     ubiattach -p "${mtddev}"
224     nand_do_upgrade_success
225  }
226 
227  # Write the UBIFS image to UBI volume
228  nand_upgrade_ubifs() {
229     local rootfs_length=`(cat $1 | wc -c) 2> /dev/null`
230 
231     nand_upgrade_prepare_ubi "$rootfs_length" "ubifs" "0" "0"
232 
233     local ubidev="$( nand_find_ubi "$CI_UBIPART" )"
234     local root_ubivol="$(nand_find_volume $ubidev rootfs)"
235     ubiupdatevol /dev/$root_ubivol -s $rootfs_length $1
236 
237     nand_do_upgrade_success
238  }
239 
240  nand_upgrade_tar() {
241     local tar_file="$1"
242     local board_name="$(cat /tmp/sysinfo/board_name)"
243     local kernel_mtd="$(find_mtd_index $CI_KERNPART)"
244 
245     local kernel_length=`(tar xf $tar_file
         sysupgrade-$board_name/kernel -O | wc -c) 2> /dev/null`
246     local rootfs_length=`(tar xf $tar_file
         sysupgrade-$board_name/root -O | wc -c) 2> /dev/null`
247 
248     local rootfs_type="$(identify_tar "$tar_file"
         sysupgrade-$board_name/root)"
249 
```

```
250     local has_kernel=1
251     local has_env=0
252
253     [ "$kernel_length" != 0 -a -n "$kernel_mtd" ] && {
254         tar xf $tar_file sysupgrade-$board_name/kernel -O | mtd write
             - $CI_KERNPART
255     }
256     [ "$kernel_length" = 0 -o ! -z "$kernel_mtd" ] && has_kernel=0
257
258     nand_upgrade_prepare_ubi "$rootfs_length" "$rootfs_type"
         "$has_kernel" "$has_env"
259
260     local ubidev="$( nand_find_ubi "$CI_UBIPART" )"
261     [ "$has_kernel" = "1" ] && {
262         local kern_ubivol="$(nand_find_volume $ubidev kernel)"
263         tar xf $tar_file sysupgrade-$board_name/kernel -O | \
264             ubiupdatevol /dev/$kern_ubivol -s $kernel_length -
265     }
266
267     local root_ubivol="$(nand_find_volume $ubidev rootfs)"
268     tar xf $tar_file sysupgrade-$board_name/root -O | \
269         ubiupdatevol /dev/$root_ubivol -s $rootfs_length -
270
271     nand_do_upgrade_success
272 }
273
274 # Recognize type of passed file and start the upgrade process
275 nand_do_upgrade_stage2() {
276     local file_type=$(identify $1)
277
278     [ ! "$(find_mtd_index "$CI_UBIPART")" ] && CI_UBIPART="rootfs"
279
280     case "$file_type" in
281         "ubi")      nand_upgrade_ubinized $1;;
282         "ubifs") nand_upgrade_ubifs $1;;
283         *)     nand_upgrade_tar $1;;
284     esac
285 }
286
```

```
287 nand_upgrade_stage2() {
288    [ $1 = "nand" ] && {
289       [ -f "$2" ] && {
290          touch /tmp/sysupgrade
291
292          killall -9 telnetd
293          killall -9 dropbear
294          killall -9 ash
295
296          kill_remaining TERM
297          sleep 3
298          kill_remaining KILL
299
300          sleep 1
301
302          if [ -n "$(rootfs_type)" ]; then
303             v "Switching to ramdisk..."
304             run_ramfs ". /lib/functions.sh; include /lib/upgrade;
                    nand_do_upgrade_stage2 $2"
305          else
306             nand_do_upgrade_stage2 $2
307          fi
308          return 0
309       }
310       echo "Nand upgrade failed"
311       exit 1
312    }
313 }
314
315 nand_upgrade_stage1() {
316    [ -f /tmp/sysupgrade-nand-path ] && {
317       path="$(cat /tmp/sysupgrade-nand-path)"
318       [ "$SAVE_CONFIG" != 1 -a -f "$CONF_TAR" ] &&
319          rm $CONF_TAR
320
321       ubus call system nandupgrade "{\"path\": \"$path\" }"
322       exit 0
323    }
324 }
```

```
325 append sysupgrade_pre_upgrade nand_upgrade_stage1
326
327 # Check if passed file is a valid one for NAND sysupgrade. Currently
        it accepts
328 # 3 types of files:
329 # 1) UBI - should contain an ubinized image, header is checked for
        the proper
330 #    MAGIC
331 # 2) UBIFS - should contain UBIFS partition that will replace
        "rootfs" volume,
332 #    header is checked for the proper MAGIC
333 # 3) TAR - archive has to include "sysupgrade-BOARD" directory with
        a non-empty
334 #    "CONTROL" file (at this point its content isn't verified)
335 #
336 # You usually want to call this function in platform_check_image.
337 #
338 # $(1): board name, used in case of passing TAR file
339 # $(2): file to be checked
340 nand_do_platform_check() {
341    local board_name="$1"
342    local tar_file="$2"
343    local control_length=`(tar xf $tar_file
           sysupgrade-$board_name/CONTROL -O | wc -c) 2> /dev/null`
344    local file_type="$(identify $2)"
345
346    [ "$control_length" = 0 -a "$file_type" != "ubi" -a "$file_type"
           != "ubifs" ] && {
347       echo "Invalid sysupgrade file."
348       return 1
349    }
350
351    echo -n $2 > /tmp/sysupgrade-nand-path
352    cp /sbin/upgraded /tmp/
353
354    return 0
355 }
356
357 # Start NAND upgrade process
```

```
358 #
359 # $(1): file to be used for upgrade
360 nand_do_upgrade() {
361     echo -n $1 > /tmp/sysupgrade-nand-path
362     cp /sbin/upgraded /tmp/
363     nand_upgrade_stage1
364 }
```

Hardcoded memory layout

In section 6.2.1 a fragment taken from the Makefile in /target/linux/ar71xx is showed to describe how a custom image can be produced, which grants write access to the U-Boot environment. The entire file is too large to be included here, but a complete list of all devices using a hardcoded environment might be interesting.

Listing B.2: Taken from /target/linux/ar71xx/Makefile

```
alfa_ap120c_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 13312k(rootfs), 1536k(kernel),
    1152k(unknown)ro, 64k(art)ro;spi0.1:-(unknown)
alfa_ap96_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    256k(u-boot-env)ro, 13312k(rootfs), 2048k(kernel),
    512k(caldata)ro, 15360k@0x80000(firmware)
alfa_mtdlayout_8M=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    6144k(rootfs), 1600k(kernel), 64k(nvram), 64k(art)ro,
    7744k@0x50000(firmware)
alfa_mtdlayout_16M=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 15936k(firmware), 64k(nvram), 64k(art)ro
all0258n_mtdlayout=mtdparts=spi0.0:256k(u-boot), 64k(u-boot-env),
    6272k(firmware), 1536k(failsafe), 64k(art)
all0315n_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 256k(u-boot-env),
    13568k(firmware), 2048k(failsafe), 256k(art)ro
ap81_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    5120k(rootfs), 2688k(kernel), 64k(art)ro, 7808k@0x50000(firmware)
ap83_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 128k(u-boot-env)ro,
    4096k(rootfs), 3648k(kernel), 64k(art)ro, 7744k@0x60000(firmware)
```

```
ap96_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro, 64k(u-boot-env)ro,
    6144k(rootfs), 1728k(kernel), 64k(art)ro, 7872k@0x40000(firmware)
ap113_mtd_layout=mtdparts=spi0.0:64k(u-boot), 3008k(rootfs),
    896k(uImage), 64k(NVRAM), 64k(ART), 3904k@0x10000(firmware)
ap121_mtdlayout_2M=mtdparts=spi0.0:64k(u-boot)ro, 1216k(rootfs),
    704k(kernel), 64k(art)ro, 1920k@0x10000(firmware)
ap121_mtdlayout_4M=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 2752k(rootfs), 896k(kernel), 64k(nvram),
    64k(art)ro, 3648k@0x50000(firmware)
ap132_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    1408k(kernel), 6400k(rootfs), 64k(art), 7808k@0x50000(firmware)
ap135_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    14528k(rootfs), 1472k(kernel), 64k(art)ro,
    16000k@0x50000(firmware)
ap136_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    6336k(rootfs), 1408k(kernel), 64k(mib0), 64k(art)ro,
    7744k@0x50000(firmware)
bxu2000n2_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 1408k(kernel), 8448k(rootfs), 6016k(user),
    64k(cfg), 64k(oem), 64k(art)ro
cameo_ap81_mtdlayout=mtdparts=spi0.0:128k(u-boot)ro, 64k(config)ro,
    3840k(firmware), 64k(art)ro
cameo_ap91_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro, 64k(nvram)ro,
    3712k(firmware), 64k(mac)ro, 64k(art)ro
cameo_ap99_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro, 64k(nvram)ro,
    3520k(firmware), 64k(mac)ro, 192k(lp)ro, 64k(art)ro
cameo_ap121_mtdlayout=mtdparts=spi0.0:64k(u-boot)ro, 64k(art)ro,
    64k(mac)ro, 64k(nvram)ro, 192k(language)ro, 3648k(firmware)
cameo_ap121_mtdlayout_8M=mtdparts=spi0.0:64k(u-boot)ro, 64k(art)ro,
    64k(mac)ro, 64k(nvram)ro, 256k(language)ro,
    7680k@0x80000(firmware)
cameo_ap123_mtdlayout_4M=mtdparts=spi0.0:64k(u-boot)ro,
    64k(nvram)ro, 3712k(firmware), 192k(lang)ro, 64k(art)ro
cameo_db120_mtdlayout=mtdparts=spi0.0:64k(uboot)ro, 64k(nvram)ro,
    15936k(firmware), 192k(lang)ro, 64k(mac)ro, 64k(art)ro
cameo_db120_mtdlayout_8M=mtdparts=spi0.0:64k(uboot)ro, 64k(nvram)ro,
    7872k(firmware), 128k(lang)ro, 64k(art)ro
cap4200ag_mtdlayout=mtdparts=spi0.0:256k(u-boot), 64k(u-boot-env),
    320k(custom)ro, 1536k(kernel), 12096k(rootfs), 2048k(failsafe),
```

```
   64k(art), 13632k@0xa0000(firmware)
cpe510_mtdlayout=mtdparts=spi0.0:128k(u-boot)ro,
   64k(pation-table)ro, 64k(product-info)ro, 1536k(kernel),
   6144k(rootfs), 192k(config)ro, 64k(ART)ro,
   7680k@0x40000(firmware)
eap300v2_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env),
   320k(custom), 13632k(firmware), 2048k(failsafe), 64k(art)ro
db120_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
   6336k(rootfs), 1408k(kernel), 64k(nvram), 64k(art)ro,
   7744k@0x50000(firmware)
dgl_5500_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro, 64k(nvram)ro,
   15296k(firmware), 192k(lang)ro, 512k(my-dlink)ro, 64k(mac)ro,
   64k(art)ro
dlan_pro_500_wp_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
   64k(u-boot-env)ro, 64k(Config1)ro, 64k(Config2)ro,
   7680k@0x70000(firmware), 64k(art)ro
dlan_pro_1200_ac_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
   64k(u-boot-env)ro, 64k(Config1)ro, 64k(Config2)ro,
   15872k@0x70000(firmware), 64k(art)ro
cameo_ap94_mtdlayout=mtdparts=spi0.0:256k(uboot)ro, 64k(config)ro,
   6208k(firmware), 64k(caldata)ro, 1600k(unknown)ro,
   64k@0x7f0000(caldata_copy)
cameo_ap94_mtdlayout_fat=mtdparts=spi0.0:256k(uboot)ro,
   64k(config)ro, 7808k(firmware), 64k(caldata)ro,
   64k@0x660000(caldata_orig), 6208k@0x50000(firmware_orig)
esr900_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro, 64k(u-boot-env)ro,
   1408k(kernel), 13248k(rootfs), 1024k(manufacture)ro,
   64k(backup)ro, 320k(storage)ro, 64k(caldata)ro,
   14656k@0x40000(firmware)
esr1750_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro, 64k(u-boot-env)ro,
   1408k(kernel), 13248k(rootfs), 1024k(manufacture)ro,
   64k(backup)ro, 320k(storage)ro, 64k(caldata)ro,
   14656k@0x40000(firmware)
epg5000_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro, 64k(u-boot-env)ro,
   1408k(kernel), 13248k(rootfs), 1024k(manufacture)ro,
   64k(backup)ro, 320k(storage)ro, 64k(caldata)ro,
   14656k@0x40000(firmware)
ew-dorin_mtdlayout_4M=mtdparts=spi0.0:256k(u-boot)ro,
   64k(u-boot-env), 3712k(firmware), 64k(art)
```

```
ew-dorin_mtdlayout_16M=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env), 16000k(firmware), 64k(art)ro
f9k1115v2_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env),
    14464k(rootfs), 1408k(kernel), 64k(nvram)ro, 64k(envram)ro,
    64k(art)ro, 15872k@0x50000(firmware)
dlrtdev_mtdlayout=mtdparts=spi0.0:256k(uboot)ro, 64k(config)ro,
    6208k(firmware), 64k(caldata)ro, 640k(certs), 960k(unknown)ro,
    64k@0x7f0000(caldata_copy)
dlrtdev_mtdlayout_fat=mtdparts=spi0.0:256k(uboot)ro, 64k(config)ro,
    7168k(firmware), 640k(certs), 64k(caldata)ro,
    64k@0x660000(caldata_orig), 6208k@0x50000(firmware_orig)
dragino2_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 16000k(firmware),
    64k(config)ro, 64k(art)ro
dw33d_mtdlayout=mtdparts=spi0.0:256k(u-boot), 64k(u-boot-env)ro,
    14528k(rootfs), 1472k(kernel), 64k(art)ro,
    16000k@0x50000(firmware);ar934x-nfc:96m(rootfs_data),
    32m(backup)ro
hiwifi_hc6361_mtdlayout=mtdparts=spi0.0:64k(u-boot)ro,
    64k(bdinfo)ro, 1280k(kernel), 14848k(rootfs), 64k(backup)ro,
    64k(art)ro, 16128k@0x20000(firmware)
mr12_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 256k(u-boot-env)ro,
    13440k(rootfs), 2304k(kernel), 128k(art)ro,
    15744k@0x80000(firmware)
mr16_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 256k(u-boot-env)ro,
    13440k(rootfs), 2304k(kernel), 128k(art)ro,
    15744k@0x80000(firmware)
pb92_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    2752k(rootfs), 896k(kernel), 64k(nvram), 64k(art)ro,
    3648k@0x50000(firmware)
planex_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    7744k(firmware), 128k(art)ro
ubntxm_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    7552k(firmware), 256k(cfg)ro, 64k(EEPROM)ro
uap_pro_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    1536k(kernel), 14208k(rootfs), 256k(cfg)ro, 64k(EEPROM)ro,
    15744k@0x50000(firmware)
ubdev_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    7488k(firmware), 64k(certs), 256k(cfg)ro, 64k(EEPROM)ro
whrhpg300n_mtdlayout=mtdparts=spi0.0:248k(u-boot)ro,
```

```
    8k(u-boot-env)ro, 3712k(firmware), 64k(art)ro
wlr8100_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro, 64k(u-boot-env)ro,
    1408k(kernel), 14080k(rootfs), 192k(unknown)ro, 64k(art)ro,
    384k(unknown2)ro, 15488k@0x40000(firmware)
wpj344_mtdlayout_16M=mtdparts=spi0.0:192k(u-boot)ro,
    16128k(firmware), 64k(art)ro
wpj531_mtdlayout_16M=mtdparts=spi0.0:192k(u-boot)ro,
    16128k(firmware), 64k(art)ro
wpj558_mtdlayout_16M=mtdparts=spi0.0:192k(u-boot)ro,
    16128k(firmware), 64k(art)ro
wndap360_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 1728k(kernel), 6016k(rootfs), 64k(nvram)ro,
    64k(art)ro, 7744k@0x50000(firmware)
wnr2200_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(u-boot-env)ro,
    7808k(firmware), 64k(art)ro
wnr2000v3_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 3712k(firmware), 64k(art)ro
wnr2000v4_mtdlayout=mtdparts=spi0.0:192k(u-boot)ro,
    64k(u-boot-env)ro, 3776k(firmware), 64k(art)ro
r6100_mtdlayout=mtdparts=ar934x-nfc:128k(u-boot)ro, 256k(caldata),
    256k(caldata-backup), 512k(config), 512k(pot), 2048k(kernel),
    122240k(ubi), 25600k@0x1a0000(firmware), 2048k(language),
    3072k(traffic_meter)
wndr4300_mtdlayout=mtdparts=ar934x-nfc:256k(u-boot)ro,
    256k(u-boot-env)ro, 256k(caldata), 512k(pot), 2048k(language),
    512k(config), 3072k(traffic_meter), 2048k(kernel), 23552k(ubi),
    25600k@0x6c0000(firmware), 256k(caldata_backup), -(reserved)
zcn1523h_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 6208k(rootfs), 1472k(kernel),
    64k(configure)ro, 64k(mfg)ro, 64k(art)ro, 7680k@0x50000(firmware)
mynet_n600_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    64k(u-boot-env)ro, 64k(devdata)ro, 64k(devconf)ro,
    15872k(firmware), 64k(radiocfg)ro
mynet_rext_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
    7808k(firmware), 64k(nvram)ro, 64k(ART)ro
zyx_nbg6716_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro, 64k(env)ro,
    64k(RFdata)ro, -(nbu);ar934x-nfc:2048k(zyxel_rfsd), 2048k(romd),
    1024k(header), 2048k(kernel), -(ubi)
qihoo_c301_mtdlayout=mtdparts=spi0.0:256k(u-boot)ro,
```

```
64k(u-boot-env), 64k(devdata), 64k(devconf), 15744k(firmware),
64k(warm_start), 64k(action_image_config),
64k(radiocfg)ro;spi0.1:15360k(upgrade2), 1024k(privatedata)
```

Update Script

This is the script used for automatic updates during the development process. It is included only for completeness and is not meant to be used in any real scenario. It is not even recommended to use this script as a basis for future work, it serves only as a proof of concept and should be completely reimplemented for any other use case.

Listing B.3: AutoUpdateScript

```
1  #!/bin/sh
2
3  cd /etc/updater
4  wget http://192.168.1.17:80/masterarbeit/openwrt/latest.hash
5
6  hashDiff=$(cmp installed.hash latest.hash)
7
8  if [[$hashDiff -eq ""]]
9  then
10         #No update needed
11    exit 0
12 else
13    http://192.168.1.17:80/masterarbeit/openwrt/latest.bin
14    verify=md5sum -c latest.hash
15    case "$verify" in
16       *WARNING*)
17         #TODO Retry download instead of aborting
18            exit 1
19       ;;
20    esac
21
22    #Do Update
23    currentBootCmd=$(fw_printenv -n bootcmd)
24    if $currentBootCmd EQUALS "run boot_flash"; then
```

```
25          mtd write /etc/updater/latest.bin mtd9
26        newBootCmd="run boot_flash2";
27     else
28        mtd write /etc/updater/latest.bin mtd7
29          newBootCmd="run boot_flash";
30     fi
31     fw_setenv bootcmd $newBootCmd
32     reboot
33  fi
```

Recovery Script

During chapter 6 the contents of the script run when starting OpenWRT and used to recovery in case of an error was built up piece by piece. The complete version is included here. Note: This script is a proof-of-concept and should not be used for anything but further development.

Listing B.4: AutoRecoveryScript

```
1  checkErrorCode()
2  {
3          local _retVal=""
4          if [[ $1 -eq "0" ]]; then
5                  _retVal=1
6          else
7                  _retVal=0
8          fi
9          echo $_retVal
10 }
11
12 #give system time to set up
13 #sleep 5m
14
15 #Check if System is online
16 wget -q --tries=10 --timeout=20 --spider http://google.com
17 isOnline=$(checkErrorCode $?)
18 #Check if webserver is running
19 ps | grep uhttpd | grep -v grep
```

```
20 httpRunning=$(checkErrorCode $?)
21 #Check if ssh server is running
22 ps | grep dropbear | grep -v grep
23 sshRunning=$(checkErrorCode $?)
24
25 currentBootCmd=$(fw_printenv -n bootcmd)
26
27 #Check if U-Boot environment is OK
28 if [[ $currentBootCmd -eq "Warning: Bad CRC, using default
      environment" ]]
29    mtd -r write /var/recovery/u_boot_env.backup mtd1
30 else
31    #Do Nothing
32 fi
33
34 if $currentBootCmd EQUALS "run boot_flash"; then
35         newBootCmd="run boot_flash2";
36 else
37         newBootCmd="run boot_flash";
38 fi
39
40 if [[ $sshRunning -eq 1 ]] && [[ $httpRunning -eq 1 ]] && [[
      $isOnline -eq 1 ]]; then
41         echo "Startup successful, switching boot command to
              $newBootCmd"
42         #fw_setenv bootcmd $newBootCmd
43 else
44         echo "System Startup failed, going for Reboot"
45         #reboot
46 fi
```

europass Curriculum Vitae

PERSONAL INFORMATION

Tobias Höller, Bsc

Jagern 45, 4761 Enzenkirchen, Austria

+43 688 8157405

TobiasHoeller@gmx.at

WORK EXPERIENCE

April 2013 – Today

Software Engineer

Catalysts Gmbh, Linz, Austria

- Development of custom software
- Development of Java based web applications

October 2015 – October 2016

Research Assistent

Institute for Networks and Security, Johannes Kepler University, Linz, Austria

- Research into security of embedded devices (focus on OpenWRT)
- Research into how to securely and automatically update embedded devices

EDUCATION AND TRAINING

September 2010 – January 2014

Bachelor of Science in Computer Science

Johannes Kepler University, Linz, Austria

- General studies giving an introduction into every aspect of computer science
- Spent one Semester abroad at Oxford Brookes University, GB

Jannuary 2014 – Today

Master of Science in Computer Science with main Topic Networks and Security

Johannes Kepler University, Linz, Austria

- Secure Code
- System Security
- Computer Forensics
- Network Management and Security

October 2016 – Today

Master of legal and business aspects in technics

Johannes Kepler University, Linz, Austria

- Basic Economic and management classes
- Basic Law classes
- Focus on IT Law (Datenschutzgesetz, E-Government-Gesetz)

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to thesis.jku.at is identical to the present master‘s thesis.

Date

Signature