Efficient delta based updates for read-only filesystem images

An applied study in how to efficiently update the software of an ECU

ELLINOR WESTERBERG
Efficient delta based updates for read-only filesystem images

An applied study in how to efficiently update the software of an ECU

ELLINOR WESTERBERG

Master’s Programme in Computer Science
Date: February 27, 2021
Supervisor: Somayeh Aghanavesi
Examiner: Robert Lagerström
School of Electrical Engineering and Computer Science
Host company: BMW
Swedish title: Effektiv delta-baserad uppdatering av filsystem med enbart läsrättigheter
Swedish subtitle: En tillämpad studie om effektiv uppdatering av mjukvaran i en styrenhet
Efficient delta based updates for read-only filesystem images

Effektiv delta-baserad uppdatering av filsystem med enbart läsrättigheter

© 2021 Ellinor Westerberg
Abstract

This thesis investigates a method for efficiently updating the software of an Electronic Control Unit (ECU) in a car. The patch sent to the car should be as small as possible and optimally only contained the changed part of the software. A popular algorithm for creating the patch is `bsdiff`. However, it is not made for filesystem images, but for binaries. Therefore, an alternative is investigated. The alternative algorithm method is based on the update engine in Android. A standalone version of the Android A/B Update is implemented and compared to `bsdiff`, in the aspect of the time it takes to generate the patch and the size of the patch. The result shows that `bsdiff` generates a slightly smaller patch. However, `bsdiff` is also a lot slower at generating the patch. Furthermore, the time increases linearithmic with the size of the filesystem image. This gives reason to believe that the Android A/B Update algorithm might be a better solution when updating an ECU that contains a full filesystem. However, this depends on if it is most valuable that the patch is as small as possible, or that the process of generating it is fast.

Keywords

Android A/B Update, Android Update Engine, bsdiff, Delta Updates, Dmv-verify, ECU
Sammanfattning


Nyckelord

Android A/B Update, Android Update Engine, bsdiff, Delta-Updatering, Dmveriy, ECU
Acknowledgments

I would like to thank BMW for letting me do my thesis at their company. Especially thanks to Michael Weiner who was my supervisor. I would also like to thank Somayeh Aghanavesi for supervising me in the writing process.

Stockholm, February 2021
Ellinor Westerberg
Contents

1 Introduction 1
  1.1 Purpose ........................................ 3
  1.2 Structure of thesis ................................. 3
  1.3 Problem Statement ................................. 4
  1.4 Research Questions ............................... 5
  1.5 Scope ........................................... 5
  1.6 Research Methodology ............................. 6

2 Background 8
  2.1 Electronic Control Units ........................... 8
  2.2 Filesystems ...................................... 9
  2.3 Authenticity of persistent storage ................ 10
    2.3.1 Booting process for embedded systems ....... 10
    2.3.2 Secure boot ................................ 10
    2.3.3 Device-Mappers Verity ....................... 12
  2.4 Firmware update concepts ........................ 14
    2.4.1 Over-The-Air updates ......................... 14
    2.4.2 Delta Updates ................................ 15
  2.5 Differential Compression Algorithms ............ 16
  2.6 bsdiff .......................................... 17
  2.7 Android ......................................... 18
  2.8 Android Update Engine ............................ 18
    2.8.1 Android A/B Update overview ............... 19
    2.8.2 Example .................................... 20
    2.8.3 Patch file .................................. 21
    2.8.4 Major and Minor version .................... 23
    2.8.5 The Applier ................................ 23
3 Related Work

3.1 Security

3.1.1 Security issues in cars

3.1.2 Secure updates

3.2 The updating process

3.2.1 Over The Air (OTA) updates in cars

3.2.2 `bsdiff` in cars

3.2.3 `bsdiff` in other situations

3.3 Related Work Summary

4 Method

4.1 Tools

4.1.1 Languages

4.1.2 Bazel

4.1.3 Protocol buffers

4.1.4 `sha256sum`

4.2 Implementation of patch generator

4.2.1 Downloading the Android source code

4.2.2 Extracting the A/B Update

4.2.3 Including External Dependencies

4.2.4 Major Changes

4.3 Implementation of the Applier

4.3.1 `bspatch`

4.3.2 Remove hashes and signature

4.4 Evaluation of the Standalone A/B Update

4.4.1 Test vectors

4.4.2 Building the Android Source Code

4.4.3 Generating the patch

4.4.4 Applying the patch

4.4.5 Test environment

4.5 Choice of method

5 Result

5.1 Code complexity

5.2 Performance

5.2.1 Time

5.2.2 Compression rate
List of Figures

2.1 Overview of Secure boot .................................................. 12
2.2 Hash tree used in Device-Mappers verity (dm-verity) ............ 13
2.3 Overview of delta update process ...................................... 15
2.4 Generating and applying the patch ................................... 16
2.5 Example of operations .................................................... 21
# List of Tables

1.1 Size of test data ................................................. 7

2.1 Operations in A/B Update ....................................... 20

4.1 Implemented operations ......................................... 37

4.2 Linux distributions used as test vectors ...................... 40

4.3 The block size used for the different distributions in the generating process ........................................... 42

5.1 Code Complexity comparison .................................... 46

5.2 Comparison of time it takes to create patch. Ordered by size of the images ............................................ 47

5.3 Comparison of the size of the patch (bytes). Ordered by size of the images ............................................ 48

E.1 sha256 hash after the ext2/ext4 images were extracted .... 73
Listings

2.1 Patch file format ........................................... 22
4.1 bzip2 in the BUILD file ................................. 34
4.2 e2fsprogs in the BUILD file ............................ 35
4.3 libdivsufsort in the BUILD file ...................... 35
4.4 protobuf in the BUILD file ............................ 36
4.5 bspatch in the BUILD file .............................. 38
4.6 Extraction of the ext2/ext4 image .................... 40
4.7 How to build the Android source code .............. 41
List of acronyms and abbreviations

AOSP  Android Open Source Project. 2, 7, 33

CAN  Controller Area Network. 27

dm-verity  Device-Mappers verity. xi, 2, 3, 6, 11–14, 54

DRAM  Dynamic RAM. 10

ECU  Electronic Control Unit. 1–3, 5, 8, 14, 17, 19, 24–29, 50, 53–55

IMA  Integrity Measurement Architecture. 6, 11

OEM  Original Equipment Manufacturer. 2, 14, 24

OTA  Over The Air. viii, 1–3, 14, 24–26, 28, 29, 54

POC  proof of concept. 6, 26

RAM  Random-access Memory. 9, 11, 27, 28

ROM  Read-only Memory. 10, 11

SPL  Secondary Program Loader. 10, 11

SRAM  Static RAM. 10

TPL  Third stage Program Loader. 10
Chapter 1

Introduction

Safety has always been a top priority in the car industry, simply because the consequences of an accident are dire. Throughout history, there has been a step by step development of features for the safety of the passengers, such as seat belts, padded dashboards and ABS. Now, autonomous vehicles are supposed to eliminate one of the biggest threats; the driver. This development has naturally lead to a huge increase in software to the extent where modern cars are dependent on computers for each functionality. While it has increased safety, it also introduces new potential risks. The safety on the road is no longer only dependent on the drivers, but also on the security of the systems they are operating. The attack vectors have multiplied as the vehicles get access to the public network and the consequences of malfunction software become even larger in an environment of self-driving cars, that interact with each other. We can therefore assume that the security issues we are facing with the current systems will not only endure but increase.

The increasing amount and complexity of software, also make it unrealistic to believe that the development process will be done when the car leaves the factory. New security threats and bugs will be discovered and the Electronic Control Unit (ECU)s will have to be updated after the car has left the vendor. It seems to be likely that the process will be similar to that used on mobile phones, where new updates regularly are sent to the device and the user can choose to update the phone when suited. These kinds of OTA updates provides the user the opportunity to update the car when it best suits the user, and new updates can also be distributed quickly.
To send large patches to each car would however quickly become very expensive for the Original Equipment Manufacturer (OEM) since it is the manufacturer, and not the user who pays for the mobile network needed (as opposed to the phone industry). In many cases, it would be unnecessary to send the whole update. Instead, it would be better to just send the changed part of the software, i.e. perform a delta update.

For smaller ECUs where the software is a simple binary, a delta patch can be generated by bsdiff, a tool used for delta compression. Different versions of bsdiff has been suggested as a solution in earlier studies (see chapter 3). For larger ECUs, that contain a whole filesystem this might not be optimal. Theoretically, bsdiff could be used on the whole filesystem image, but it is made to be used on binaries and handles smaller amounts of data better and might be better to use on each file in the filesystem. Since most ECUs historically is quite small and only runs a single program, there are not many studies investigating different solutions.

While regular OTA updates might prove a solution to some security issues in itself, it can be a risk if not handled correctly. It is important that the authentication and integrity of the new software can be ensured. There are many ways to do that but one way is to implement a mechanism that ensures that the software running on the device is correct. A solution for this could be to use secure boot and dm-verity to verify the filesystem images. But this requires that the updated software is bit by bit identical to a reference image in the back-end and prevents us from simply running bsdiff on a single file that requires update. The metadata in the filesystem might have changed as well and when applying the patch the resulting filesystem image might not be identical to the reference image.

As already stated, the phone industry has used OTA updates for a much longer time than the car industry, and dm-verity is currently used to verify the software in Android. One approach could be to look at how this is done in an open-source project, such as Android Open Source Project (AOSP). There, image-based updates are made, but the comparison is done on smaller parts of the files-system images.
1.1 Purpose

The purpose of this study is to see if the Android Update Engine could be a good alternative for updating an ECU containing, not only a single binary but a filesystem. Many previous studies focus on updating a single binary, and mostly on how different versions of bsdiff can be used (see chapter 3). There are reasons to believe that with the development of autonomous vehicles, the ECUs will grow because of the extended functionality. It is therefore sensible to develop a method that does not have any constraints on the filesystem size. The aim here is to investigate a method that could be a reasonable alternative to bsdiff and help in further understanding what is required for a well-functioning update process of these larger ECUs.

There are multiple reasons for looking at the Android Update Engine. First of all, the mobile phone industry has used OTA updates for a longer period of time and it is therefore reasonable that their processes are more mature and could be used for inspiration. Secondly, we are looking specifically at ECUs containing a filesystem and an operating system built on Linux. It is therefore reasonable to look at the updating process for compatible filesystems. Thirdly, a requirement for the updating system is that it should be possible to use with dm-verity or a similar authentication mechanism, where the filesystem image is verified as a whole. Lastly, an obvious yet limiting requirement is of course that the source code we want to look at is publicly available and allows us to copy some functionality. Since Android is open-source, based on Linux and uses dm-verity, it is suitable for the purpose of this study.

1.2 Structure of thesis

Next in this chapter, the Problem Statement (1.3) is described, and then the Research Questions (1.4) are outlined. The following section, 1.5, then describes the scope and hence the focus on the study and what related parts were omitted. Lastly, the Research Methodology section (1.6) describes the goals and overall method to achieve them and why this particular method was chosen.

The next chapter, Background (2), goes through tools, terminology, and functionality used in the thesis. A large part focuses on the Android Updating Process since this is the central part of the investigation. The Background is
followed up by chapter 3, Related Work, which describes previous work in the field. Chapter 4, Method, outlines how this research was conducted; the solution, and how it was implemented together with how it was tested. Then, chapter 5, Result, describes the results of the research. This is followed up by chapter 6, Discussion, that analyzes the results; what they mean, the reason for them, and how this can be used in the future research. Lastly, chapter 7, Conclusions, summarizes the conclusions drawn from the research.

1.3 Problem Statement

The Android A/B Update (described in section 2.8) seems to be a good solution for an effective update of remote devices. Since it only uses $bsdiff$ for the necessary blocks in the filesystem, and not for the whole filesystem image, it is reasonable that it is more efficient than $bsdiff$ when creating the patches. As stated in section 2.6, $bsdiff$ is not very fast when creating the update. However, it is of course not only the time to create the patch that is important when considering the update method. The whole point with delta updates is to decrease the amount of data that needs to be sent to the client. Hence, it is also important to compare the compression rate for the two different methods, before deciding which one is better.

A problem with the Android A/B Update is that it is very dependent on the rest of the Android source code and relies on other parts of the code and many external dependencies. Therefore, it is very large to download and not suitable to use as is. Hence, it would be good to build another, Standalone version, that relies on the same algorithm but does not have all these large dependencies. This would rather follow the Unix principle (Simmonds, 2015) of having one program doing one single thing good, instead of the approach here; that is to have one complex program that does a lot of different things. Since Android is open-source, it is possible to study the code and reuse some parts to create such a version.
1.4 Research Questions

The following two research questions will be answered by this study:

1. How can a Standalone version of the Android A/B Update be built with as few external dependencies as possible?

2. How well would such an implementation perform (regarding time it takes to create the patch and compression rate) compared to;

   (a) the original Android A/B Update?
   
   (b) bsdiff?

By answering these questions, it can be estimated whether the Android A/B Update is a suitable algorithm to use for the update of an ECU. Taking into account the time it takes to generate a patch, how large the patch is compared to a full update, and how easy it would be to implement such a program.

The hypothesis is that bsdiff is not suitable for updating larger filesystems, mainly because it will take too much time to generate the patch. Instead, the Android A/B Update would be a better solution.

1.5 Scope

In this thesis, we mainly look at how an ECU with a filesystem can be updated. For smaller ECUs that only contain a single binary, many earlier studies have been conducted (see chapter 3). Furthermore, this thesis focuses on the ext4 filesystem format (see section 2.2) which is commonly used in different Linux distributions such as in Android and Raspbian (which are both based on the Linux core).

The main focus is also on how to create the patch and not how to apply it. In reality, it might be important to look at how long time it takes to apply the patch as well. This should be weighed in when deciding what update method to use. However, it was here assumed that the difference in time for applying the patch would be smaller than for the generating process when comparing the Android A/B update and bsdiff. To apply the patch is a much simpler process and as stated in section 2.6, bspatch is significantly faster than bsdiff. Considering how the patch is generated and applied in the Android Update Engine, the time difference should be similar there. When generating the patch, this difference
can be a lot larger since the process is much more time and memory-consuming in the first place.

*dm-verity* is here considered to be enough to ensure the integrity of the filesystem. As stated in section 3 there is still discussion on how to best ensure that the updated software is correct. However, it can be assumed that if the integrity of the filesystem is not checked at all, there is a larger problem. There are also other ways to ensure the integrity of a filesystem image. One of them is *fs-verity*, where each file in the filesystem is verified, rather than the whole image\(^1\). However, it is quite new compared to *dm-verity* and is not as well-tried. Another is Integrity Measurement Architecture (IMA), but it is supposed to be slow compared to *dm-verity* and has more requirement from the surrounding systems\(^2\). Therefore, only an updating process where *dm-verity* is a requirement will be considered.

### 1.6 Research Methodology

This is an applied study where the research questions will be answered by attaining the following goals:

- Implement a Standalone version of the Android A/B Update as a proof of concept (POC).
- Ensure generated patch results in a bit by bit identical filesystem image as the target image when applied to the source image.
- Compare the time it takes to generate the patch and the patch-size to the existing Android A/B Update to ensure that the performance is similar.
- Compare the time it takes to generate the patch and the size of the patch to *bsdiff* to estimate which should be used.

The implementation is done by downloading the original Android source code and then extract the code needed for the update. This code is modified so that only the external dependencies necessary for the update are downloaded and included. The building tool Bazel is used to download the external dependencies and build the source code. When the Standalone version has

---


\(^2\)https://lwn.net/Articles/459420/ (Accessed: 2020-12-18)

\(^3\)https://lwn.net/Articles/394170/ (Accessed: 2020-12-18)
been built, performance tests are conducted to compare the different update methods; the Standalone version implemented here, the original Android A/B Update, and bsdiff. The integrity of the patch generated by the Standalone version is ensured by applying it to the old image and using \textit{sha256sum}\textsuperscript{4} to compare the new image to the reference image it was created from. An illustration of this can be found in figure 2.4.

As test vectors, three Linux distributions were used; piCore, openWrt, and Raspbian. They were chosen based on size and filesystem format. They all use the ext4 format and the sizes can be found in table 1.1.

\begin{table}[h]
\centering
\begin{tabular}{|l|c|}
\hline
\textbf{Image} & \textbf{Approx. size} \\
\hline
piCore & 80 000 512 B \\
openWrt & 268 435 456 B \\
raspbian & 1 323 302 912 B \\
\hline
\end{tabular}
\caption{Size of test data}
\end{table}

The time and the compression rate is measured in order to compare the performance between the different algorithms and implementations. The time is measured using the Linux program \textit{time}. The compression rate is calculated using the following formula:

\[
\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}
\]

The analysis of the complexity is quite simple. Ideally, a thorough analysis using cyclomatic complexity or something similar would be used. However, it has been deemed as too time consuming for this project, due to the size and complexity of the AOSP.

Instead simple measurements as the number of external dependencies and the size of the binary has been used. This cannot give a completely fair comparison of complexity, but it can give a indication of the reduced amount of code.

\textsuperscript{4}https://linux.die.net/man/1/sha256sum (Accessed: 2020-01-17)
Chapter 2

Background

In this chapter, concept relevant for understanding the rest of the thesis is described. This includes the updating process in cars and the authentication method that it is assumed that the ECU requires. A large focus will then be to investigate the Android Update Engine in order to see how it should be used in a new context.

2.1 Electronic Control Units

Electronic Control Units (ECU) are in short computers in cars that control different functions. In the past few decades, there has been a rapid development in the number of ECUs and their importance. Today, a car can contain over 70 ECUs and would not even start without them. All functions are now regulated via an ECU, even though the mechanics are very similar to the first developed cars. If you push on the breaks or lock a door, a modern car will perform that action using an ECU. Functioning, safe, and secure software is hence crucial in the automobile industry.

The size of an ECU can differ depending on the functionality. Most are small chips that only run a single program. However, there are also larger ECUs today, with separate persistent storage that contains a filesystem and runs a whole operating system. They are typically Linux based and requires more complicated authentication mechanisms than a simple binary file.
2.2 Filesystems

The focus of this thesis is to look at the updating process of software that is stored in persistent storage. This is where data is stored that should be retained when the computer is powered off. Unlike Random-access Memory (RAM), whose contents are lost when there is a power loss, persistent-storage devices keep such data intact. This is where the kernel loads the root filesystem. (Arpaci-Dusseau and Arpaci-Dusseau, 2014; Pate, 2003)

A filesystem is a way to structure the data so that it is easy to interact with. The user should not have to care exactly where or how on the hard drive the data is stored. Two key concepts are files and directories. A file is a linear array of bytes that can be written to or read from. The file has a low-level name, called inode number that the user normally is not aware of. The directories are containers that include either another directory or a file. That way, a directory tree is built. (Arpaci-Dusseau and Arpaci-Dusseau, 2014; Pate, 2003)

The tool mkfs is used to create a filesystem. It needs to be supplied with a device (such as a disk partition or an image/file) and a filesystem type. Once a filesystem is created it also needs to be mounted. This is what actually makes it accessible within the uniform filesystem tree so that the user knows where to find it. The root filesystem is mounted by the kernel during system startup. Each filesystem can then be mounted on any directory in the root filesystem, (except on root). The directory where the filesystem is mounted is called a mount point. (Arpaci-Dusseau and Arpaci-Dusseau, 2014; Pate, 2003)

There are different kinds of filesystems. Normally it divides the disk into blocks of a specified size. These blocks contain the data that needs to be persistently stored but also meta-data describing the filesystem. There are for example an inode table, allocation structures, and superblocks, that contain information about the specific filesystem. (Arpaci-Dusseau and Arpaci-Dusseau, 2014; Pate, 2003)

In this thesis, the filesystem format used is ext2 and ext4. Ext4 was created in 2006 as the next generation of ext3 and is the standard filesystem in Linux systems (Mathur et al., 2007). The block size is 512, 1024, 2048, or 4096 bytes. It also has a journal that keeps track of all the changes done to the
filesystem so that it can be restored in case of a crash. Ext2 is the predecessor of ext3 and does not contain a journal. (Arpaci-Dusseau and Arpaci-Dusseau, 2014; Pate, 2003)

2.3 Authenticity of persistent storage

2.3.1 Booting process for embedded systems

The first program to run when the device is powered-on is the Read-only Memory (ROM) code. This piece of code is programmed in the chip when it is manufactured and is therefore proprietary and cannot be replaced. Because it is so specific, the ROM code cannot make any assumptions of any external hardware and cannot make use of the main system memory. It only has access to a small amount of Static RAM (SRAM). The purpose of the ROM is to load a chunk of code, called Secondary Program Loader (SPL), from a preprogrammed location into this SRAM. The last thing the ROM code does is to jump to the beginning of the SPL. (Simmonds, 2015)

The SPL sets up the memory controller and loads the Third stage Program Loader (TPL) into the main memory; Dynamic RAM (DRAM). At the end of this phase the SPL jumps to the area where the TPL is loaded. If the SRAM is large enough, this phase may be skipped and the bootloader loaded directly into the SRAM by the ROM code. (Simmonds, 2015)

The TPL normally consists of a bootloader. This can be an open source alternative such as U-boot or Barebox. This phase normally allows the user to perform maintenance tasks through a command-line user interface. The main purpose of bootloader is to load the kernel into the memory and pass the control to it. It also passes some basic information over to the kernel, such as details of detected hardware. When the kernel boots up it has to mount a root filesystem and execute an initial program, (default name of this is init). This can be done by the initramfs process.¹ (Simmonds, 2015)

2.3.2 Secure boot

Secure boot is used to ensure that the firmware running on a computer has not been tampered with, intentionally by unauthorized users or unintentionally by

authorized/authentic users. It builds on a Chain of Trust where the software described in each phase above is secured by the previous running software. An illustration of this can found in figure 2.1.

Secure boot starts with a Root of Trust. This is the asymmetric key pair (for example RSA) used to sign the ROM code. The public key, that is used to verify the signature, is stored in an immutable part of the memory, just as the code. The private part of the key, that is used to sign the code, is only known by the manufacturer.² (Oshana and Kraeling, 2019)

Once authenticated, the ROM code can be used to authenticate the SPL. The SPL can then authenticate the bootloader and the bootloader can authenticate the kernel. This is how the Chain of Trust works; each level authenticates the next-level code before handing over the control to it. (Oshana and Kraeling, 2019)

After secure boot has authenticated the first levels there are different ways to authenticate rootfs, the root filesystem. In some cases, standard secure boot can continue to be used. In this case, rootfs will be expanded in RAM and no new application or image can be added at runtime. This will of course not work for larger partitions, that does not fit into memory but is placed on a persistent storage device. In that case, some other verification mechanism is needed. There are different methods for this, one being dm-verity. Alternatives to dm-verity is fs-verity³ or IMA⁴. (Oshana and Kraeling, 2019)

⁴https://lwn.net/Articles/394170/ (Accessed: 2020-12-18)
2.3.3 Device-Mappers Verity

Device-Mapper Verity (dm-verity) is normally used to ensure the integrity of a root filesystem. It requires the target to be read-only to ensure that it does not change, once verified. It implements a device mapper target that validates the data blocks contained in a filesystem against a list of cryptographic hash values. These hash values are stored in a hash-tree as shown in figure 2.2. All the leaf-nodes maps to the blocks in the filesystem and the root hash can be used to validate the whole tree. If the hash for a specific block does not come out as expected, the module assumes that the device has been tampered with and causes the access attempt to fail.\(^5\)^\(^6\)^\(^7\)

\(^7\)https://lwn.net/Articles/459420/ (Accessed: 2020-06-15)
The general process for generating the hash tree is as follows:

1. Choose a random salt
2. Unsparsify your system image into blocks (normally 4k)
3. For each block, get its (salted) sha256 hash
4. Concatenate these hashes to form a level
5. Pad the level with 0s to a 4k block boundary
6. Concatenate the level to your hash tree
7. Repeat steps 2-6 using the previous level as the source for the next until you have only a single hash, the root hash.

The root hash and the salt is then used to create the \textit{dm-verity} mapping table. The mapping table also contains the location for the hash-tree, the block size for the hash and the data, the block device for the data and the number of blocks.

Once the mapping table is generated, it is signed as well. This table signature is the first to be validated when a partition is verified. The mapping table and table signature is then bundled into verity metadata. Then the system image, the verity metadata and hash tree are concatenated.

\textsuperscript{9}https://gitlab.com/cryptsetup/cryptsetup (Accessed: 2020-08-25)
itself can not ensure the integrity of the software. *dm-verity* operates in the kernel, so if the kernel is not authenticated, it can still be compromised. It is important that *dm-verity* is proceeded by secure boot (as described above) or some other similar process. Otherwise, there is no way of actually ensuring that the key, that the boot image is verified against, is correct.  

An important notice is that *dm-verity* requires the filesystem image to be identical to the one on the manufacturing side. It is not enough that each file and directory is identical but also all the metadata, bit by bit. This is what makes the updating process challenging.

### 2.4 Firmware update concepts

#### 2.4.1 Over-The-Air updates

With Over-The-Air (OTA) updates, the patch is sent to the device over a mobile connection. OTA updates have been standard in the mobile phone industry for many years. Some car manufactures have used it as well for quite some time but it is still fairly immature in the automotive industry.

Figure 2.3 shows the overview of an OTA update of an ECU. A patch, in this case a delta patch, is generated in the back-end. It is sent to the car over a mobile connection and then used to update an ECU, one partition at the time. To send the data over the mobile connection is expensive for the OEM, and the primary reason that we want to make the patch as small as possible.

The ECU contains two identical partitions so that one can be updated while the other is running. The goal of this is to make the update process as seamless as possible. If there was just one partition, the car would have to be turned off during the update. Now it just need to be restarted to switch to the newly updated partition. This way, there is also a "backup" if the update process goes wrong. This can be referred to as A/B updates.

---

2.4.2 Delta Updates

To reduce the size of the patch, delta updates can be used. In delta updates, differential compression algorithms (see section 2.5) are used to create a patch that only contains the part of the software that has changed. Instead of trying to find repetition within the files, as normal compression algorithms do, differential compression algorithms find repetition between two files and creates a delta file that contains this difference, and how it should be applied to one of the files to make it identical to the other.

An illustration of how the patch is generated and applied can be found in figure 2.4. As stated in section 2.3, the updated software has to be bit by bit identical to the reference image it was created from.
2.5 Differential Compression Algorithms

A differential compression algorithm is described by Suel (2019) as follows:

Delta compression techniques encode a target file with respect to one or more reference files, such that a decoder who has access to the same reference files can recreate the target file from the compressed data.

Typical applications scenarios include revision control system and versioned filesystems that store many versions of a file, or updates over networks where the recipient already has an older version of the data.

Many different delta compression algorithms has been developed over the years. In 1996 rsync was presented as an algorithm for updating a file on one machine to be identical to a file on another machine. It was evaluated on tar files created of the Linux kernel source for two different versions. Another delta compression algorithm is Zdelta, that is built on the zlib compression library (Trendafilov et al., 2002). Xia et al. (2014) developed Ddelta as a faster alternative to Zdelta and Xdelta\(^\text{11}\).

bsdiff was developed specifically for binary files, since the above mentioned algorithms did not always perform well for binaries. Later, Ni et al. (2016) compared three of these incremental update algorithms; rsync, bsdiff and

\(^{11}\)http://xdelta.org/ (Accessed:2020-12-28)
\textit{Xdelta} in order to evaluate which one was the best. It depends on the file, but on average \textit{bsdiff} turned out to have the highest compression ratio. As discussed in chapter 3, \textit{bsdiff} has been used to build an updating process for ECUs in many previous studies.

\section*{2.6 \textit{bsdiff}}

\textit{bsdiff} is a common tool for differential compression on binary files and is popular to use to perform delta updates. In order to apply the binary patch to the source file \textit{bspatch} is used. \textit{bsdiff} and \textit{bspatch} can be used on files up to $2^{61} - 1 \approx 2.306 \text{EB}$ but \textit{bsdiff} is memory-intensive and producing the patch can take long time. \textit{bsdiff} requires $\text{max}(17 \ast n, 9 \ast n + m) + O(1)$ bytes of memory, where $n$ is the size of the source file and $m$ is the size of the target file. \textit{bspatch} requires $n + m + O(1)$ bytes. \textit{bsdiff} runs in $O((n + m) \log n)$ time and \textit{bspatch} runs in $O(n + m)$ time. (Percival, 2003)

It is important to notice that an algorithm that runs $O((n + m) \log n)$ has a significantly worse time complexity than a linear and that it hence, takes much longer time to create that patch than it takes to apply it. When the time complexity is linear-arithmatic ($O(n \log n)$) the time to run the algorithm will increase not only as much as the input increases but multiplied with $\log n$.

Here follows a simplified description of how the patch file is created with \textit{bsdiff}:

1. Use suffix sorting to perform indexing on the source file.
2. Go through the target file and find regions that match against regions in the source file.
3. Of these regions, record the ones that contain at least 8 bytes not matching the forward extension of the previous match.
4. Generate pairwise disjoint sets of approximate matches by extending the matches in each direction. Every suffix of the forward and backward extension has to match at least 50\%. 

5. Now the unmatched regions of the target file corresponds to the new source code. Approximate matches will correspond to unchanged code. Create the patch with three files:

- **Control file** with ADD and INSERT instructions. (ADD are for the approximate matches, INSERT for the new data).
- **Difference file** with bytewise differences of the approximate matches.
- **Extra file** containing the data that corresponds to the new code.

6. Compress the patch using the compression algorithm bzip2.

The strength of the algorithm is that it creates repetition. The three files that the patch consists of are actually larger than the original target file, but highly compressible with bzip2. (Percival, 2003)

As stated above, bsdiff is made to be used on binary files and not on whole filesystem images. Therefore, it is likely that it is not the best option for generating a patch for a whole filesystem image. Instead, it would likely be preferable to use bsdiff on each updated file in the filesystem image. This is done in the Android Update Engine.

### 2.7 Android

Android is one of the largest spread operating systems in the cell phone industry. It is built on the Linux kernel and distributed under the Apache 2.0 license\(^{12}\). This makes it possible to redistribute the software, with or without modifications, as long as changed files are noted and the original copyright notice left unaltered. Android supports many different filesystem formats, amongst them JFFS2 and, more commonly, ext4.\(^{13}\)

### 2.8 Android Update Engine

Android’s Update Engine contains three different update models: a Full Update, an A/B Update, and an In-place Update. Mainly the A/B Update will be considered in this thesis. Both the A/B Update and the In-Place Update performs a delta update and are very similar. However, the A/B Update utilizes

---

\(^{13}\)https://android.tutorials.how/android-file-system/ (Accessed: 2020-08-28)
two partitions and updates one at a time as is described in section 2.4. This is the scenario when updating an ECU. The In-place Update is used when there only is one partition. The Full Update is irrelevant since it simply sends the whole new version of the firmware to the client and this is what should be avoided.

2.8.1 Android A/B Update overview

The A/B Update utilizes different compression methods to generate a patch that is as small as possible. Simplified, it looks at each file in the image and decides what compression method generates the smallest compressed version of that file. An operation representing that compression method is generated for that file. The operations and the data needed to transfer the older version to the newer version is added to a patch file. Once all the operations have been generated, the patch file is sent to the client, where it is applied, in order, to the old image. That way, a new image, identical to the requested version, is generated. ¹⁴

Table 2.1 describes the how the different operations function. The order, in which these operations are assigned to files or blocks is described in the list below. A more detailed description of the patch-generating algorithm can be found in Appendix A.

1. The ZERO operation is assigned to the blocks that are supposed to be zero-filled in the target partition.

2. All the blocks that have not changed between the source and the target are assigned the SOURCE_COPY operation.

3. List all the files (and their associated blocks) in the source and target partitions and remove blocks (and files) which we have already generated operations for in the last two steps.

4. For all new files, generate a REPLACE, REPLACE_XZ, or REPLACE_BZ operation depending on which generates the smallest data blob. Go through these files concurrently.

5. For all other files and metadata, that has changed, generate SOURCE_BSDIFF or PUFFDIFF depending on which generates the smallest data blob. Go through these files concurrently.

### 2.8.2 Example

Figure 2.5 illustrates a simplified example of how the operations can be created and applied in order to go from one version of the filesystem image to another. In this illustration it is for example shown how a changed file, `foo`, is utilizing the operation BSDIFF while the unchanged file `bar`, that only has moved in the filesystem, uses SOURCE_COPY. In this case, `foo` has increased in size, but this would of course not be necessary in reality for BSDIFF to be used.

---


16[Inspired from https://sites.google.com/a/chromium.org/dev/chromium-os/chromiumos-design-docs/autoupdate-details (Accessed: 2020-09-02)]
2.8.3 Patch file

The format of the patch is represented by the following struct pseudocode, in listing 2.1. A `delta_update_file` contains all the data and instructions needed to update a system from one specific version to another specific version. The `DeltaArchiveManifest` protobuf is an ordered list of `Operation` objects. These objects are stored in a linear array in the `DeltaArchiveManifest`. It is worth mentioning that the hash of each operation is added to the patch. It is assumed that this is done to make the debugging easier, in case a update goes wrong.
struct delta_update_file {
    char magic[4] = "CrAU";
    uint64 major_version;
    // Size of protobuf DeltaArchiveManifest
    uint64 manifest_size;
    // Only present if format_version > 1:
    uint32 metadata_signature_size;
    // The Bzip2 compressed DeltaArchiveManifest
    char manifest[];
    // The signature of the metadata (from the
    // beginning of the payload up to this location,
    // not including the signature itself).
    // This is a serialized Signatures message.
    char metadata_signature_message[
        metadata_signature_size
    ];
    // Data blobs for files, no specific format.
    // The specific offset and length of each data
    // blob is recorded in the DeltaArchiveManifest.
    struct {
        char data[];
    } blobs[];
    // These two are not signed:
    uint64 payload_signatures_message_size;
    char payload_signatures_message[];
};

Listing 2.1 – Patch file format
2.8.4 Major and Minor version

Except of having different kinds of update (Full Update, A/B Update and In-place Update), the delta updates also have different versions that allows different operations and configurations. The version is defined with the major and minor version and these numbers are hard coded into the client.

The major version is sent to the client and specifies information about the version that is relevant to the applier. A new major version is created when the changes are so large so the DeltaArchiveManifest has to be updated. Currently, there are only two major versions: 1, and 2, with major version 1 being deprecated. The other version, the minor version, does not contain as large configurations as the major. It can for example describe what different operations that are allowed by the client.\(^\text{17}\)

2.8.5 The Applier

The Android Update Engine contains a Delta Applier and a Delta Installer. The Installer is used in production to download and apply the patch. The Applier is a simpler Python script that can be used for testing. It takes the source image and the patch as input and outputs the new, updated image. Simplified, it just goes through all operations in the DeltaArchiveManifest and applies them in order to the source image. The new data that is needed for some of the operations (REPLACE, BSDIFF, etc.) are supplied in the blobs[]-struct (see 2.8.3). Before applying the operations, the Applier verifies that the signatures of the patch and the operations are correct.

Chapter 3
Related Work

There are many studies looking into how to perform OTA updates in a car. It is a subject that has been relevant for over a decade and there is still no established solution for how to do it securely and efficiently.

That said, it has been a challenge to find work concerning the updating process of ECUs containing a whole filesystem. There are likely two reasons for that. First of all, it is a new issue. Up until recently, ECUs only contained a single program, and even now, most of the ECUs in a car do; there has not been an extended amount of research on how to update a larger ECU. Secondly, the OEMs normally do not share their source code so there is no way of knowing how their update process works either. Hence, the researches mentioned below mostly focus on how to update a single binary on an ECU, or the overall process. bsdiff seem to be a common choice for updating ECUs but also in other contexts.

While the main focus in this study is the updating process, the relevance of an efficient updating process is motivated by studies of the security in cars. Therefore, the first section in this chapter describes the research done in the security field and how the absence of a working update process can be troublesome. It then moves on to the security issues with the OTA update itself, together with some solutions. The next section describes earlier work that has been done to make the update process more efficient. Lastly, there is a short, conclusive summary of the previous work.

3.1 Security

This section first describes how earlier studies have been conducted concerning the security of cars. It then goes through how updates have been developed to be more secure.
3.1.1 Security issues in cars

There are many studies showing how easily a car can be subject of a cyber attack. Already in 2010 Koscher et al. (2010) showed in an empirical study what attackers could do if they were able to compromise a car’s internal communication network. The result showed that they could get access to more or less every ECU and do damage of huge magnitudes, such as turning off the engine and prevent or force the car to break. They managed to access the network both via inserting a malicious component and via a remote wireless vulnerability. A study from 2011 focus more on how to access the internal network of a car and proved that an ECU could be compromised via a long-range wireless channel (more than 1km). One of their suggested mitigation was to improve code robustness. Miller and Valasek (2013) demonstrated how to remotely control two vehicles (Toyota and Ford) from 2010. A similar study was made in 2015 in which they demonstrated a remote attack to many Fiat-Chrysler vehicles (Miller and Valasek, 2015). This resulted in Fiat-Chrysler recalling 1.4 million cars1.

These examples show how important it is to perform regular software updates of the ECUs. It is simply not feasible to expect all the software to completely bug free for all future when the car is created. Neither can the cars life span be shortened. OTA updates were recommended already in 2007 (Shavit et al., 2007) even though the discussion at that time was quite immature and for example stated that ”the number of software updates needs to stay as low as possible”. Today, most researchers would agree that the car for security reasons should be updated regularly.

3.1.2 Secure updates

Parkinson et al. (2017) identified this updating process as one of the most important future challenges for securing cars against cyber-attacks. While they did not see a certain solution for how the updating process should be designed they did recommend OTA updates. Even though it from a security perspective seem necessary to regularly perform updates, and even though OTA in many ways seem like the most efficient (and therefore also most secure) way to do it, the updates in them-self entails a risk. They are both a new attack surface where malicious code could willingly be injected and a component where an error could be made without meaning to. This has motivated multiple studies to look into how OTA updates can be performed but still ensure that the intended software runs in the car.

There are many different suggestions for how to secure the process of OTA updates (Halder et al., 2019). The most common suggestion seems to be to use securely stored symmetric and asymmetric keys to ensure the integrity and confidentiality of

the software (Idrees et al., 2011; Mahmud et al., 2005; Steger et al., 2016).

Nilsson and Larson (2008) presented quite early a study that promotes OTA updates but they wanted to solve the security problems it introduced. They presented a lightweight protocol for secure firmware OTA updates in intelligent vehicles. The protocol provides data integrity, data authentication, data confidentiality, and freshness. A hash chain is created of the firmware and the first packet is signed by a trusted source, thus authenticating the whole chain. Packets are encrypted using symmetric keys that prevent an attacker from modifying a firmware in transit. The protocol has especially been designed with respect to the limited hardware resources in vehicles and with careful considerations to the low bandwidth and the risk of packet loss.

10 years later, Steger et al. (2018) suggested using blockchain to tackle the implicated security and privacy challenges of future connected vehicles. They stated that their POC can be used to distribute software OTA. They evaluated the overhead and latency using this method and compared it to a certificate-based system. They concluded that their method was beneficial and plan to further improve it.

3.2 The updating process

In this section work of the updating process is described. First, general considerations about efficiency updates are discussed. Since the most researched method seem to be some modifications of bsdiff, this is then described in its own section. Lastly, it is also outlined how bsdiff has been suggested in other situations. Unfortunately, no studies were found on how bsdiff can be used on a whole filesystem image or how larger systems should be patched. Neither were any related studies on the Android Update Engine found. However, since the Android Update Engine mostly focuses on how to use bsdiff on a filesystem image, it is reasonable to focus on the bsdiff algorithm.

3.2.1 OTA updates in cars

In many studies, the goal is to decrease the time it takes to update the software in a car. Lee et al. (2015) and Herberth et al. (2019) suggested to update multiple ECUs in parallel. This does decrease the update time when there are multiple ECUs to update but as Bogdan et al. (2016) mentioned, it might increase the update time when only one ECU is to be updated. Furthermore, it does nothing to decrease the costs of transferring the data, and might not be the best single solution for OTA updates.

Terashima et al. (2017) proposed a method for reducing the update time where each ECU is updated in a distributed manner. Many other studies seem to agree that the
transition time is what uptakes most of the time during the update, which could be solved by producing smaller patches (Bogdan et al., 2016; Onuma et al., 2017; Teraoka et al., 2016).

Kiyohara et al. (2012) suggested a new delta technique for software upgrade services for on-vehicle information devices. It is not built on bsdiff but on another compression algorithms. Onuma et al. (2017) proposed that for ECUs with a very small RAM a regular compression method such as deflate should be used. However, as they mentioned themselves, this is only a good idea for ECUs where a delta compression method is not possible to use. A regular compression method is better used for data containing a lot of repetition, such as images and sounds. The software in a car might not have such repetitions and with the overhead, the size of the delta payload could actually increase in size. And even if it does not, it could take a longer time to reprogram the device since the data must be decompressed (Bogdan et al., 2016). Therefore, this is only a suitable solution where the RAM of the ECU is very small, and regular compression is the only solution.

### 3.2.2 bsdiff in cars

In a study made by Ni et al. (2016), bsdiff has the highest compression rate on average when compared to rsync and xdelta and most previous studies suggests bsdiff as a suitable delta compression algorithm in order to minimize the patch size in a car update.

Komano et al. (2018) build their solution on bsdiff but wanted to solve two issues they identified from earlier studies. First of all, they wanted to create a verifiable update for a hierarchical and multi-vendor system. The second aim was to create an efficient rollback mechanism. Their solution was to use bsdiff with some modifications that made it possible to use only one patch file both for the update and the rollback. They introduced a verifiable end-to-end firmware update using hash-chain. Their tests also showed that bsdiff was mostly useful if the change ratio is smaller than about 50%.

Onuma et al. (2016) put forth other ideas on how to change bsdiff to work better for the updating process of ECUs. The main reason, according to them, is to improve the update time, and the main problem is the amount of time it takes to transfer the update in the car. The reason being the low bandwidth of Controller Area Network (CAN). They also focused on binary updates and made modifications for bsdiff to work better on ECUs. bsdiff is optimized for a non-fixed-length operation set (i.e. Intel) while most ECUs use an ARM architecture with a fixed-length operation set. This was their focus.

Nakanishi et al. (2013) also used bsdiff for their updates of ECUs. They stated that
bsdiff can produce quite a lot smaller differential files than other tools they had looked into, especially for binaries. They identified three challenges for their selected tool and found solutions for two of them. The first challenge was that ECUs normally has a limited amount of memory, whereas bsdiff is made for PCs with more RAM. The solution for this was to create an in-place update that generates the newer binary by editing the older binary directly, instead of generating the newer binary on the buffer in the independent place from the older binary. The second challenge was that ECUs normally has an unstable power supply; the driver can suddenly decide to turn off the car. This was solved by making it possible for bspatch to resume from an unanticipated cut of power supply by saving the context of the update process in non-volatile memory. The third and last challenge identified was unstable radio wave condition; the car could lose connection in a tunnel for example. This was however considered out of scope for the update process and not looked further into.

Teraoka et al. (2016) focused on resource-constrained micro-controllers in ECUs. Their solution was also built on bsdiff. However, they stated that bspatch was unsuitable due to the memory resource constraints of in-vehicle ECUs. To reduce memory usage, they introduced three modifications to bsdiff: block-based comparison, two-stage compression (use LZMA instead of bzip2), and data-format serialization.

### 3.2.3 bsdiff in other situations

bsdiff has also been proposed in other situations in order to minimize the update file. In 2012 Samteladze and Christensen (2012) proposed bsdiff be used for updating applications in Android in order to save cost and energy. Their tests showed a 48% reduction of the app update size on average when they tried their method on the 50 most popular Android apps.

It has also been suggested to be used for OTA updates of robot swarms (Varadharajan et al., 2018). In this study, a way to efficiently update robot swarms was proposed. The suggested approach uses binary deltas for minimizing the bandwidth requirement and update time. When a robot shares a new update with its peers, it generates a patch using bsdiff.

Lastly, bsdiff was also proposed by both Stolikj et al. (2012) and Stolikj et al. (2013) for efficient reprogramming of wireless sensor network.

### 3.3 Related Work Summary

Earlier studies show that OTA updates are necessary for the future and that the size of the patch hence must be reduced. bsdiff seem to have been a popular algorithm
for this and most of the studies look into different ways to make it a better tool when updating ECU s. There have also been many studies on how to make the update as secure as possible since OTA updates also introduces new attack vectors. Most of the identified challenges consider the resource constraints of an ECU s compared to a PC. However, none of the above mention studies takes into account that in the future, there is a great chance that the ECU s will be larger and that the update should not only be for a single program but a complete filesystem and possibly an operating system. bsdiff is not made for that. This is a clear gap that this thesis will examine.
Chapter 4

Method

This chapter first goes through the different tools and programs used to build the update in section 4.1. Then, in section 4.2, it is outlined how the standalone version was implemented and how it differs from the original Android A/B Update. This section also describes the necessary external dependencies in subsection 4.2.3, as well as how they are included in the code. Then, in section 4.4, it is described how the test to compare the different update methods are conducted. Lastly, in section 4.5, the choice of method is discussed; what could have been done differently in the implementation and why.

4.1 Tools

This section describes the different tools and programs that are used for the implementation and the performance test.

4.1.1 Languages

Python and C++ are used in the implementation. The Android Update Engine is mainly written in C++. C++ is generally considered a fast language that is suitable for this usage.

The Applier is built in Python. Because of the relatively poor performance of Python, this is not optimal. But since it is only used to confirm that the patch is correctly generated it is sufficient here. A good thing with Python is that it is easy to include C/C++ programs in the code when built with Bazel (see section 4.3).
4.1.2 Bazel

Android uses its own build system called Soong\textsuperscript{1}. In this project, Bazel is used instead. Bazel is another build system developed by Google\textsuperscript{2}. They both use a high-level build language and are suitable for larger projects, but Bazel is also open-source.

In Bazel, each project has a \textsc{workspace} file in the root directory. It contains the references to external dependencies if there are any.\textsuperscript{3} Each package (normally a subdirectory of the root directory) contains a \textsc{build} file. The \textsc{build} file describes how the code should be built i.e., what files should be included and what dependencies are needed to build the code. \textsuperscript{4}

When using Bazel, it is easier to integrate other projects that use Bazel. Among the external dependencies in this project, most does not use Bazel but \textit{GNU Make}.\textsuperscript{5} \texttt{rules_foreign_cc}\textsuperscript{6} is therefore used. It is a collection of Bazel rules that can be used for foreign build systems inside Bazel projects, such as \texttt{cmake} and \texttt{configure make}.

4.1.3 Protocol buffers

Protocol buffers, or \textit{protobuf}, is a protocol for serializing structured data, similar to JSON or XML. It was created in 2001 and can be found on GitHub.\textsuperscript{7} The idea is to create a way to easily send and store data independently of language or platform. The data is stored in \texttt{.proto} files that can be compiled into language-specific files. If for example data from \texttt{example.proto} should be used in a C++ code, you first have to compile \texttt{example.proto} into two C++ compatible files; \texttt{example.cc} and \texttt{example.h}. These classes and structs of these files can then be included and utilized in C++ code.

\textit{Protobuf} also has support for Python and \texttt{.proto} files can be compiled to work for both C++ and Python by Bazel.\textsuperscript{8} In the Android Update Engine, \textit{protobuf} is the protocol used to store and sent the patch. It is open-source and is, therefore, suitable to use for the standalone version as well.

\begin{thebibliography}{9}
\bibitem{1}https://source.android.com/setup/build (Accessed: 2021-01-17)
\bibitem{2}https://opensource.google/projects/bazel (Accessed: 2021-01-17)
\bibitem{3}https://docs.bazel.build/versions/master/build-ref.html (Accessed: 2021-01-17)
\bibitem{4}https://docs.bazel.build/versions/master/build-ref.html#BUILD_files (Accessed: 2021-01-17)
\bibitem{5}https://www.gnu.org/software/make/ (Accessed: 2020-12-18)
\bibitem{6}https://github.com/bazelbuild/rules_foreign_cc (Accessed: 2020-12-18)
\bibitem{7}https://github.com/protobuf/protobuf (Accessed: 2020-12-18)
\end{thebibliography}
4.1.4 sha256sum

*sha256sum* is a program in Linux that computes a sha256 message digest. Since sha256 is collision-resistant it can be used to verify that a filesystem image has been patched correctly (Dang, 2008). The message digest of the patched image can be compared to the message digest of a reference image. An illustration of how this is done can be seen in figure 2.4 (see section 2.4).

4.2 Implementation of patch generator

In this section, it is described how the Standalone version of the Android A/B Update is implemented.

4.2.1 Downloading the Android source code

First, the Android source code is downloaded so that it is possible to take a closer look at the algorithm and see how it originally was implemented.11

```
$ mkdir WORKING_DIRECTORY
$ cd WORKING_DIRECTORY
$ repo init -u
    https://android.googlesource.com/platform/manifest
$ repo sync
```

The `WORKING_DIRECTORY` now contains the full Android Source Code. The Update Engine code can be found in the directory `WORKING_DIRECTORY/system/update_engine`.

4.2.2 Extracting the A/B Update

The most significant difference is that the A/B Update is extracted from the rest of the code. In the original Android Update Engine, all the different update versions are put together. Many functions and classes are the same and instead uses *if-statements* to change the functionality for the different update versions. The Update Engine also depends on other packages in the code that with small modifications would not be necessary for the updating process. That way, the Update Engine is closely intertwined with code that has very little to do with the updating process. In the Standalone

---

9https://linux.die.net/man/1/sha256sum (Accessed: 2020-12-18)
    4b28a53926c27d47fac7784be553c494ae44753f/payload_generator/ (Accessed: 2020-12-18)
version, all unnecessary dependencies are removed, both external and from the other parts of the AOSP. The A/B Update is extracted without the Full Update or In-place Update.

This is done by simply extracting the files that are necessary for the A/B Update. In Appendix B the paths to files in the original source code can be found for each of the included files. Then these files are reviewed and all unnecessary dependencies are removed. For each file, all parts of the code that are only used by the Full Update or In-place Update are removed as well. It is not always clear if a part of the code is necessary for the A/B Update. In that case, it is possible to investigate it by removing that part of the code from the full Android Source Code and see if it is possible to run the original A/B Update without that code. How to run the original A/B Update can be found in section 4.4.3. The files included from the Android Source code can be found in appendix B.

The original code is built with Soong, but in the standalone version, Bazel is used instead. This is changed by removing the Android.bp files (that are used to define the build process in Soong) and replacing them with WORKSPACE and BUILD files, defining which files should be built and how they are dependent on each other. More about this can be read about in section 4.1.2 and 4.2.3.

4.2.3 Including External Dependencies

The Android source code version used here includes 321 external dependencies (these can be found under the folder external) but most of them are not used at all in the Update Engine. In Android, all the files for the external dependencies are downloaded and built with Soong, though originally most of them use GNU make. For the standalone version, it was instead decided to use Bazel to dynamically download the dependencies in the build process and then build them using rules_foreign_cc; a set of rules created to use GNU make, configure and cmake in Bazel.

In this section, each external dependency that is necessary will be described; the purpose, the source, and how it is built. How these are included and built is one of the larger differences from the original source code. The http_archive rule is used to download the repositories and makes its targets available for binding. This is defined in the WORKSPACE file and can be found in Appendix C.

bsdiff/bspatch

bsdiff is used for delta compression on the blocks/files in the filesystem image that benefits from delta compression. bpatch is used in the Applier to apply the SOURCE_BSDIFF-operation. bsdiff/bspatch is one of the few external libraries that
are not completely built using rules_foreign_cc. The bsdiff/bspatch in Android is slightly different from the original bsdiff/bspatch. It utilizes the same code but also includes some wrappers for bsdiff and bspatch that makes it possible to use them on a range of blocks instead of files. Therefore, the necessary files were copied directly from the Android repository (directory: external/bsdiff) and compiled using Bazel. The BUILD-file that defines which files are needed for bsdiff can be found in Appendix D.

**bzip2**

*bzip2* is a compression algorithm. It is mainly used by bsdiff and therefore added to the BUILD file in the bsdiff directory. It is also used as just a compression algorithm for the REPLACE_BZ operation but can then be included from the bsdiff directory. It is built with make. The make rule is loaded from the rules_foreign_cc rules. The code below shows how it is included in the BUILD file.

```python
load("@rules_foreign_cc//tools/build_defs:make.bzl", "make")

make(
  name = "bzip2",
  lib_source = "@bzip2//:all",
  static_libraries = ["libbz2.a"],
  visibility = ["//:__pkg__"],
)
```

Listing 4.1 – *bzip2* in the BUILD file

**e2fsprogs**

Despite what the name might imply, *e2fsprogs* supports ext3 and ext4 as well as ext2. It is used to create an Ext2Filesystem object to open and handle the filesystem image. Most importantly, this contains a vector with the blocks of a file, encoded in Extents. *e2fsprogs* is built with configure make and the following is added to the BUILD file in the ext2_filesystem directory to make that happen:

---

Method

load(
    "@rules_foreign_cc//tools/build_defs:configure.bzl",
    "configure_make"
)

configure_make(
    name = "e2fsprogs",
    lib_source = "@e2fsprogs//:all",
    make_commands = ["make", "install", "make install-libs"],
    static_libraries = ["libext2fs.a", "libcom_err.a"],
)

Listing 4.2 – e2fsprogs in the BUILD file

libdivsufsort

libdivsufsort is a library used for suffix sorting\(^\text{13}\). libdivsufsort is used by bsdiff and is therefore added to the same BUILD file as bsdiff. It is built with cmake and added to the BUILD file as shown below:

load(
    "@rules_foreign_cc//tools/build_defs:cmake.bzl",
    "cmake_external"
)

cmake_external(
    name = "libdivsufsort",
    lib_source = "@libdivsufsort//:all",
    cache_entries = {
        "BUILD_DIVSUFSORT64": "ON",
        "CMAKE_BUILD_TYPE": "Release",
        "CMAKE_INSTALL_PREFIX": "/usr/local",
    },
    shared_libraries = [
        "libdivsufsort.so.3",
        "libdivsufsort64.so.3"
    ],
)

Listing 4.3 – libdivsufsort in the BUILD file

 Protobuf

Protobuf is the protocol used for the patch file (see section 2.8.3 and 4.1.3). It is supported by Bazel and does not require rules_foreign_cc. How it is included in the WORKSPACE file can be seen in Appendix C. In the BUILD file, the following is added:

```build
load(
  "@rules_cc//cc:defs.bzl",
  "cc_binary",
  "cc_proto_library"
)

load(
  "@rules_proto//proto:defs.bzl",
  "proto_library"
)

proto_library(
  name = "update_metadata_proto",
  srcs = ["update_metadata.proto"],
  deps = ["@com_google_protobuf//:timestamp_proto"],
)

cc_proto_library(
  name = "update_metadata_cc_proto",
  deps = ["update_metadata_proto"],
  visibility = [
    "/ext2_filesystem:__pkg__",
    "/payload_consumer:__pkg__",
    "/common:__pkg__",
    "/test:__pkg__",
    "/scripts:__pkg__",
  ],
)
```

Listing 4.4 – protobuf in the BUILD file

This compiles update_metadata.proto to a update_metadata.pb.h and update_metadata.pb.cc that can be included in C++ files. For the targets where this should be used, update_metadata_cc_proto is added as a dependency.
4.2.4 Major Changes

Some changes are made to the Standalone A/B Update compared to the original implementation. This section describes these changes and why they are made.

Fewer Operations

Fewer operations are included in the Standalone A/B Update. As the table 4.1 shows, the REPLACE_XZ and the PUFFIN operations are not included. The reason for this is to minimize the code-base and add as few dependencies as possible.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZERO</td>
<td>Assigned to zero-filled block in target image. Discards the associated blocks.</td>
</tr>
<tr>
<td>SOURCE_COPY</td>
<td>Assigned to all blocks that has not changed between the source and the target image.</td>
</tr>
<tr>
<td>REPLACE</td>
<td>Assigned to new file</td>
</tr>
<tr>
<td>REPLACE_BZ</td>
<td>Assigned to new file and compresses with bzip2</td>
</tr>
<tr>
<td>SOURCE_BSDIFF</td>
<td>Produce binary diffs between a source and target data blob.</td>
</tr>
</tbody>
</table>

Table 4.1 – Implemented operations

Major/minor version

As stated in 2.8.4 there are different versions of the Android A/B Update. The algorithm described in the Background has major version 2 and minor version 4. The support for different versions is removed in the Standalone version in order to simplify the code. Instead, only one simplified version is implemented.

Concurrency

In the original Android A/B Update, the operations are generated for each file concurrently when producing the patch. For simplicity, this is removed as well so that nothing is done concurrently in the Standalone A/B Update. Instead, each file and its blocks are processed sequentially.

Multiple images

In the original A/B Update, there is support for generating patches for multiple images in one run of the program. This functionality is removed in the standalone version since it is considered redundant and adds unnecessary complexity to the code.
**Hashes**

In the original Android version, each operation is hashed. This hash is then included along with the operation in the patch file. The reason for this is probably to facilitate the debugging process if the update is incorrect. This hash is removed in the Standalone version for multiple reasons. First of all, it simplifies the code and makes it possible to remove an external dependency that is used to calculate the hash. Secondly, it decreases the size of the patch slightly.

**Signature**

The patch file is no longer signed. This is also motivated by simplifying the code. For compatibility with the Applier, it is still a field in the payload format, but it is always empty and not checked by the Applier.

### 4.3 Implementation of the Applier

To be able to test if the patch generator is correct the patch needs to be applied. The Android Update Engine contains an applier in Python that can be used for this purpose (see section 2.8). Some minor modifications have to be done in order to use it.

#### 4.3.1 `bspatch`

The Applier has to use `bspatch` in the application process, to apply those files/blocks that are created using `bsdiff`. `bspatch` is added as an external dependency together with `bsdiff` (see section 2.6). To run from the Python code, `bspatch` has to be added as data dependency in the Appliers `BUILD` file.

```python
py_binary(
    name = "paycheck",
    srcs = ["paycheck.py"],
    data = ["//bsdiff:bspatch"],
)
```

Listing 4.5 – `bspatch` in the `BUILD` file

The rest of the applier can run without any building system such as Bazel since Python is a scripting language.

#### 4.3.2 Remove hashes and signature

As stated in section 4.2.4 the original Android A/B Update includes the hash for each operation in the patch. The Applier then also expects these hashes when applying the
patch. Therefore, this feature has to be removed from the Applier. The same way, the signature of the patch is removed from the Applier.

### 4.4 Evaluation of the Standalone A/B Update

Tests are conducted in order to evaluate the implemented Standalone A/B Update. The goal is to measure the performance and compare the implementation to the already existing Android A/B Update as well as to `bsdiff`. These tests are also used to ensure that the image is patched correctly.

In this section, the images used as test vectors is first specified. Then it will be described how the Android Source Code is built. After that it will be described how the patches are generated and how they are applied. Lastly, the hardware used in the tests is provided.

#### 4.4.1 Test vectors

As test vectors, three different Linux distributions are used. For each distribution, two different versions are chosen so that a patch can be created from them and then applied to the "older" version; the target image (as illustrated in figure 2.4). For each distribution and version, the zip-file is downloaded and the ext2/ext4 filesystem image extracted. Table 4.2 shows the distributions and versions that are used and where they can be retrieved. It also includes the sha256 hash so that the download can be verified.
<table>
<thead>
<tr>
<th>Name</th>
<th>Link</th>
<th>sha256sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>piCore 1</td>
<td><a href="http://tinycorelinux.net/11.x/armv6/test_releases/RPi/piCore-11.0alpha1a.zip">http://tinycorelinux.net/11.x/armv6/test_releases/RPi/piCore-11.0alpha1a.zip</a></td>
<td>2690f187297e89d317ed1e137eff6e9531b73389a7b2d5640fc46fb32fa820fc5</td>
</tr>
<tr>
<td>piCore 2</td>
<td><a href="http://tinycorelinux.net/11.x/armv6/test_releases/RPi/piCore-11.0beta2.zip">http://tinycorelinux.net/11.x/armv6/test_releases/RPi/piCore-11.0beta2.zip</a></td>
<td>ead81b19e5040a2c7b4c25d8b4bc5fe0ff8b0afaf180d30b72b2624493618621</td>
</tr>
<tr>
<td>openWrt 1</td>
<td><a href="https://downloads.openwrt.org/releases/19.07.2/targets/brcm2708/bcm2709/openwrt-19.07.2-brcm2708-bcm2709-rpi-2-ext4-factory.img.gz">https://downloads.openwrt.org/releases/19.07.2/targets/brcm2708/bcm2709/openwrt-19.07.2-brcm2708-bcm2709-rpi-2-ext4-factory.img.gz</a></td>
<td>620bf5d2e89057b1b9a74f022b636cb0fxcf3062e6c1474b4fb3d3f36175ff</td>
</tr>
<tr>
<td>openWrt 2</td>
<td><a href="https://downloads.openwrt.org/releases/07.3/targets/brcm2708/bcm2709/openwrt-19.07.3-brcm2708-bcm2709-rpi-2-ext4-factory.img.gz">https://downloads.openwrt.org/releases/07.3/targets/brcm2708/bcm2709/openwrt-19.07.3-brcm2708-bcm2709-rpi-2-ext4-factory.img.gz</a></td>
<td>621087f0b50d0e8734ec0ecbb219b439875c563d11759840b019c3b9faeb1d94</td>
</tr>
<tr>
<td>Raspbian 1</td>
<td><a href="https://downloads.raspberrypi.org/raspbian_lite/images/raspbian_lite-2017-02-27/2017-02-16-raspbian-jessie-lite.zip">https://downloads.raspberrypi.org/raspbian_lite/images/raspbian_lite-2017-02-27/2017-02-16-raspbian-jessie-lite.zip</a></td>
<td>d10c127806b2c7e37fa1cd2e55db627f8d4a43e87f947b4129d30b9da60f40a7</td>
</tr>
<tr>
<td>Raspbian 2</td>
<td><a href="https://downloads.raspberrypi.org/raspbian_lite/images/raspbian_lite-2017-03-03/2017-03-02-raspbian-jessie-lite.zip">https://downloads.raspberrypi.org/raspbian_lite/images/raspbian_lite-2017-03-03/2017-03-02-raspbian-jessie-lite.zip</a></td>
<td>aa30736371ab6688af8091f8b61e0ceb1237fa0117b788341711848011e94052</td>
</tr>
</tbody>
</table>

Table 4.2 – Linux distributions used as test vectors

For each file the ext2/ext4 image needs to be extracted. The following series of commands can be used for that. piCore is used as an example.

```
# Decompress the download.
# (Use gzip -d for the .gz file.)
$ unzip piCore-11.0alpha1a.zip

# Find starting block for ext2/4 image
$ sfdisk -l -uS piCore-11.0alpha1a.img

# Extract the file system image
$ dd if=piCore-alpha-ext4.img Sskip=195693 bs=512
```

Listing 4.6 – Extraction of the ext2/ext4 image
Appendix E contains a table with the sha256 message digest after the ext2/ext4 image has been extracted. Once extracted, these images are used as test vectors. They will here be referred to as <distribution> 1 and <distribution> 2, where 1 is the old version image and 2 is the new version image (as seen in table 4.2).

### 4.4.2 Building the Android Source Code

The Android source code is also needed so that the original Android A/B Update can be built. The A/B Update is referred to as the Delta Generator in the Android Source code. See section 4.2.1 for how to download the source code. 4.4.2 shows how the Delta Generator and the Applier can be built once downloaded. It requires protobuf to be installed. The Applier is mainly in Python but uses bspatch. For that reason, bspatch has to be built as well before the Applier can be used.

#### Listing 4.7 – How to build the Android source code

```
# Install protobuf for Python
$ pip install protobuf

# Initialize the environment
$ . build/envsetup.sh

# Build the Delta Generator
$ m delta_generator

# Build bspatch that is used in the Applier
$ m bspatch
```

### 4.4.3 Generating the patch

For each test vector, three patches are generated; one by bsdiff, one by the Standalone A/B Update, and one by the original Android A/B Update. The generation process is timed using the Linux program `time`. As described in section 2.2 the underlying structure of a filesystem consists of blocks of a specific size. Since the Android Update Algorithm operates on the blocks, the block size has to be specified for both the original and the standalone version. bsdiff handles the filesystem image as a single binary file and is therefore not affected by the block size.
<table>
<thead>
<tr>
<th>Distribution</th>
<th>Block size</th>
</tr>
</thead>
<tbody>
<tr>
<td>piCore</td>
<td>512</td>
</tr>
<tr>
<td>openwrt</td>
<td>4096</td>
</tr>
<tr>
<td>raspbian</td>
<td>4096</td>
</tr>
</tbody>
</table>

Table 4.3 – The block size used for the different distributions in the generating process

**Standalone A/B Update**

$ bazel build //:generate_delta
$ time ./bazel-bin/generate_delta <source_image> <target_image> <patch_file> <block_size>

The program is designed so that the block size is set to 1024 if not specified.

**Android A/B Update**

$ time ./out/host/linux-x86/bin/delta_generator
   --minor_version=4 --old_partitions=source_image
   --new_partitions=target_image --partition_names=root
   --out_file=patch_file

In the Original Android A/B Update the block size is hard coded to be 4096. This constant has to be changed manually between the runs. This is done by changing the variable `kBlockSize` that can be found in the file `system/update_engine/payload_generator/delta_diff_generator.cc`. The major version is 2 if we don’t specify it. The minor version is specified 4 (that allows the operation PUFFDIFF to be used). The partition name can be anything (the name is mainly relevant when there are multiple images in the input). However, it cannot be empty. If you want to check the partition name, you can use the command `e2label <filesystem_image>`.

**bsdiff**

$ time bsdiff <source_image> <target_image> <patch_file>

`bsdiff` does not require any specific configurations.
4.4.4 Applying the patch

This is how each patch is applied to the source filesystem image. It is also verified using sha256 to ensure that the generated image is bit by bit identical to the target image.

**Standalone A/B Update**

```
$ ./scripts/paycheck.py <patch_file> --part_sizes 0 --part_names root --src_part_paths <source_image> --out_dst_part_paths <new_image> --dst_part_paths <new_image> --block-size <block_size> --allow-unhashed --bspatch-path ./bazel-bin/bsdiff/bspatch

$ sha256sum new_image target_image
```

**Android A/B Update**

```
$ ./system/update_engine/scripts/paycheck.py <patch_file> --part_sizes 0 --part_names root --src_part_paths <source_image> --out_dst_part_paths <new_image> --dst_part_paths <new_image>

$ sha256sum <new_image> <target_image>
```

**bspatch**

```
$ bspatch <source_image> <new_image> <patch_file>
$ sha256sum <new_image> <target_image>
```

4.4.5 Test environment

The tests were conducted on a Ubuntu Virtual machine on a Windows host. All tests ran on the same machine.
Virtual Ubuntu Machine (Virtual box) configurations:

RAM: 15G
CPU(s): 1
Thread(s) per core: 1
Core(s) per socket: 1

Windows computer:

CPU: Intel(R) Core(TM) i7-8850H CPU @2.60GHz 2.59GHz.
RAM: 32GB

4.5 Choice of method

Here it is described what could have been changed in the standalone implementation and different approaches that has been considered.

Overall approach

In this approach, large parts of the code are extracted from the existing code from Android Update Engine and used almost identically. The parts that were not needed are removed. Another approach could be to look at the code and try to separately implement the same thing without following the same approach as Android has. The result is now dependent on a structure that is used in Android and might not be suitable for a smaller update engine.

However, large parts of the code would still be identical, and there is no use reinventing the wheel. It is also likely that some parts of the implementation would not be as good since the Android Update Engine is both developed, maintained, and supported by people with more resources available.

Build system

Many alternatives could have been used instead of Bazel. A common tool is of course GNU make. However, it was considered easier to use Bazel for a larger project such as this. It is also easier to use with protobuf.
rules_foreign_cc

rules_foreign_cc is not officially supported by Google and these rules might not work with future versions of Bazel. But the alternative would be to write these Bazel rules yourself, which would also lack support for future versions unless updated manually.
Chapter 5

Result

In this section, the result is described. First, a comparison of the source code for the Standalone version and the original Android A/B Update is conducted. This is followed by the result of the performance comparison.

5.1 Code complexity

The goal to create a Standalone version of the Android A/B Update was successful. It resulted in a delta patch generator of a much smaller code base with fewer external dependencies to build but with mainly the same functionality as the original A/B Update. The Applier was implemented in Python and more or less identical to the Android version.

If the number of external dependencies is compared, it is clear that the new implementation requires significantly less external dependencies (5, while the original version contains over 300). Consequently, it takes a significantly shorter amount of time to download and build the Standalone version.

Furthermore, as seen in table 5.1, the size of the original A/B Update binary was almost ten times as large as the Standalone version binary. This difference is the result of a much larger code base with more alternatives and versions.

<table>
<thead>
<tr>
<th></th>
<th>Standalone A/B Update</th>
<th>Android A/B Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>External dependencies</td>
<td>5</td>
<td>321</td>
</tr>
<tr>
<td>Binary size (bytes)</td>
<td>2 276 928</td>
<td>21 664 744</td>
</tr>
</tbody>
</table>

Table 5.1 – Code Complexity comparison
5.2 Performance

Mainly two things have been considered when measuring the performance of the update: the time it takes to create the patch and the compression rate for the different methods.

5.2.1 Time

The time it took to generate the patch can be found in table 5.2. It was significantly shorter for the original Android version and the Standalone version, compared to bsdiff. The smallest images, piCore, had approximately the same time for the Android A/B Update and the Standalone version but for the larger Raspbian image, the original Android A/B Update was clearly faster. However, the largest difference was, as expected, bsdiff, which took significantly more time for the smaller images, piCore, and OpenWrt. For Raspbian it ran for over three days before it was aborted because it took too much time.

<table>
<thead>
<tr>
<th>Image</th>
<th>Android A/B Update</th>
<th>Standalone A/B Update</th>
<th>bsdiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>piCore</td>
<td>43 s</td>
<td>33 s</td>
<td>10 min</td>
</tr>
<tr>
<td>openWrt</td>
<td>7 s</td>
<td>15 s</td>
<td>9 min</td>
</tr>
<tr>
<td>Raspbian</td>
<td>15 min</td>
<td>422 min</td>
<td>&gt;3 days</td>
</tr>
</tbody>
</table>

Table 5.2 – Comparison of time it takes to create patch. Ordered by size of the images

5.2.2 Compression rate

As stated in section 1.6, the compression rate is defined as the ratio between the uncompressed data and the compressed data:

\[
Compression \ Ratio = \frac{Uncompressed \ Size}{Compressed \ Size}
\]

As Uncompressed Size, the full image size of the target filesystem image. As Compressed Size the patch is used. Hence, a larger compression rate is better since it means that the compressed file is smaller.

The compression rate was very similar for the smallest image, piCore; 6-7 for each method. For the openWrt image, bsdiff performed better than the Android versions, with a compression rate of 3231. The Android A/B Update had a compression rate of 1776 and the Standalone version slightly better; 1879. For the Raspbian image, bsdiff does not have any result since it took too much time to generate the patch. The
Android A/B Update and the Standalone A/B Update are very similar on the Raspbian with a compression rate of 354.8 and 356.8, respectively.

<table>
<thead>
<tr>
<th>Image</th>
<th>Full Image size</th>
<th>Android patch (ratio)</th>
<th>Standalone patch (ratio)</th>
<th>bsdiff patch (ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>piCore</td>
<td>80 000 512 B</td>
<td>12 557 729 B (6.371)</td>
<td>11 851 660 B (6.750)</td>
<td>11 753 850 B (6.806)</td>
</tr>
<tr>
<td>openWrt</td>
<td>268 435 456 B</td>
<td>151 122 B (1.776)</td>
<td>142 845 B (1.879)</td>
<td>83 084 B (3.231)</td>
</tr>
<tr>
<td>Raspbian</td>
<td>1 323 302 912 B</td>
<td>3 729 244 B (354.8)</td>
<td>3 714 039 B (356.3)</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.3 – Comparison of the size of the patch (bytes). Ordered by size of the images.
Chapter 6

Discussion

This section discusses the result and what it means. Firstly, what method should be considered the best is discussed with respect to the time it takes to apply the patch and the compression rate. Secondly, possible sources of errors will be discussed. Lastly, it is discussed what future work might be conducted and what changes are recommended to use the Standalone version in production.

6.1 Performance

6.1.1 Time

As stated in section 1.1 the main purpose of this thesis is to study an alternative to bsdiff and see how well it performs compared to bsdiff. The hypothesis was that bsdiff is very slow to generate the patch and that the Android A/B Update is a better solution for an ECU containing a filesystem and not just a single binary. According to the results, this seems to be accurate. This is also supported by the theoretical time complexity mentioned in section 2.6, where it is stated that the time it takes to generate the patch is linearithmic. If for example the filesystem size doubles, the time it takes to generate the patch will not only double, but increase even more. Furthermore, with bsdiff, the time it takes to generate the patch also only depends on how large the filesystem images are, and not the inner structure of them; such as the files, or data within the files, or how much the two filesystem images differ from each other.

This is different for the Android A/B Update. It is much faster at generating the patch, especially for larger filesystem images. It would be much more complex to compute the time complexity for this algorithm and hence it has been omitted in this thesis. But it is clearly dependent on the number of files (since each file is processed) and also on how large parts have changed from one image to the updated image. The reason is that
blocks that have not been changed will be assigned the SOURCE_COPY operation and then skipped when each file is processed. Further, practical investigations would have to be conducted in order to find out how much this would affect the time, but it is good that when smaller changes have been made, it takes a shorter time to generate.

There is, however, not only a difference between bsdiff and the Android A/B Update but also between the original Android A/B Update and the Standalone version that was implemented here. It is not as large a difference, and it still performs much better than bsdiff, but it is still a clear difference for the larger filesystem image. The reason is probably that these filesystem images contain many larger files, and the original Android A/B Update processes the files concurrently. As stated in section 4.2.4 the concurrency was removed in order to simplify the implementation. If this was to be used in production, it is now clear that it would probably be a good idea to re-implement.

Overall, it is clear that concerning the time it takes to generate the patch, the Android A/B Update performs significantly better than bsdiff. From how the Standalone version was implemented here, it becomes evident that the processing of the files should be done concurrently.

### 6.1.2 Compression rate

The time it takes to generate the patch is irrelevant if the size of the patch is not considered; to reduce the patch size is the reason to perform a delta compression in the first place. As the results show, the Android A/B Updates do perform quite well. However, bsdiff is still quite a lot better, especially for the larger, openWrt image. These are probably more accurate to look at than the piCore images. The reason is that the piCore image has a different inner structure. When studying the content of the image, it was found that the files and folders are already compressed, as opposed to the openWrt and Raspbian images, which have a more classic Linux structure. This highlights how important it is to know the structure of the particular kind of filesystem that is used in the ECU if it is desired to optimize the updating method.

Something that might be worth looking into before making a decision is which method has the highest potential for further improvements. For example, the last step of bsdiff is to compress the delta with bzip2 (see section 2.6). Hence, compressing the patch once again would not yield much difference. The overhead might even make the result larger than after just one compression. However, only some parts of the patch, the Operations in the DeltaArchiveManifest (see section 2.8) is compressed in the A/B Update, and it might therefore be possible to improve the size by just using bzip2 on the whole patch. There might be other solutions that are even better, but since bsdiff has been more researched and used (without any larger changes in the
compression rate), the A/B Update likely has a larger chance of improvements.

The difference in the compression rate between the original Android A/B Update and the Standalone version is not very large. It is expected since the algorithms are very similar. The difference that might affect the compression rate is that the Android A/B Update has two more operations to chose from: PUFFDIFF and REPLACE_XZ and that the original Android A/B Update also adds the hash of the operation. What might be a bit surprising here is that PUFFDIFF and REPLACE_XZ do not seem to make a significant difference. Since the Standalone version renders slightly smaller patches in the tests conducted here, it seems that it at least makes a larger difference to remove the hashed operations than it does to add these alternative compression methods. It might still be useful to have the hashed operations since it might be important for troubleshooting in case the updating process goes wrong, but it would have to be weighed against how small you want the patch to be.

Conclusively, bsdiff does yield a better compression rate than the Android A/B Update. However, the difference is not that large, if you look at the percentage, and both methods yield a decent compression rate. Furthermore, the PUFFDIFF and REPLACE_XZ operations do not seem to be very important for the compression rate.

### 6.2 Sources of errors

The main source of error in the comparison between bsdiff, the Standalone version, and the original Android A/B Update is that more different filesystem images should be tested. Since the result seems to depend on the structure and size of the filesystem it is important to test on images similar to the ones that are going to be used. If that is not possible, at least test on many different sizes and structures.

It is also possible that the behavior and result of the algorithm depend on how large parts of the software were changed. Delta compression is only useful if large parts of the source and target image are similar. This is not taken into consideration in this thesis, neither are there any considerations of what would happen with two identical images. The test vectors used in this thesis are probably a realistic representation of an update from one version to the next, but it is never actually confirmed how large parts of the image that has changed from the source to the target image.

Another thing that can affect the result is the hardware used. In this case, only one core was utilized in the virtual machine. That might affect how efficient the concurrent part of the updating system in the Android A/B Update Engine is.
6.3 Future Work

This section is divided into two parts. The first part describes what possible features you might want to add in the future to the Standalone version of the A/B Update if it were to be used in production as well as what further investigations can be conducted to improve it. The second part describes on a higher level future work that should be conducted concerning the updating process of ECUs.

6.3.1 Improving the Standalone version

The implementation of this study would have to be improved before it could work in production. Multiple things should be considered and features might be added.

Support for different filesystem formats

The current solution only supports the ext2/4 filesystem format. Even though this is the most common format for Linux, it cannot be assumed that all ECUs use that format for the distribution of software. In the future, it would be appropriate to support different filesystems, such as squashfs.

Add Concurrency

In the chosen methods, one of the goals was to make the implementation as simple as possible. Therefore, the concurrency was not included in the Standalone version and each file was processed sequentially. As discussed above, this is probably the main reason the time performance was worse in the Standalone version than the original. To implement such functionality would probably not be very hard, and therefore definitely worthwhile. It is probably possible to make more functionality concurrent. This should be investigated.

Include more compression algorithms

Another thing that was removed due to simplicity was the different compression methods. Only one was included; bzip. In the original Android Update Engine, xz was also used to compress the data. These compression methods were then compared and the method that yielded the smallest result was chosen. This would most likely result in a slightly better overall delta compression.

There was also another delta compression algorithm in the original Android A/B Update; PUFFDIFF and this could be included for the same reason. However, as discussed above, the addition of these compression algorithms might not make a large difference to the compression rate. However, if it were time to include them, and they
did not slow down the process of generating the patch (which they do not seem to), it might be worth the effort.

It could also be considered to introduce compression algorithms that have not been used in the Android A/B Update. In section 2.5, other delta compression algorithms are mentioned and new ones are developed. There might be some that are worth adding to the Standalone version in the future. It might also be possible to improve \texttt{bsdiff} to work better for cars. As mentioned in section 3.2.2, Onuma et al. (2016) has introduced a new version of \texttt{bsdiff} that where the architecture of an ECU is considered.

**Hash operations and partitions**

In the original Android Update Engine, a hash was added for each operation as well as for the partitions. If something goes wrong in the update, this could be useful as a debugging tool to identify what operation went wrong, or if any of the partitions were faulty. This will also result in a slightly larger patch file but it might be worth it if it simplifies the debugging process enough.

**Change variables and settings for the Standalone version**

There are variables in the Android A/B Update that could be tweaked in order to perform the best update. For example, there is a limit for how large files you can perform \texttt{bsdiff} on. It is possible that by changing such settings, the size of the patch would decrease. A way to find out the best settings could be to simply perform tests on how changing different variables would change the performance of the update.

**The Applier**

This thesis focused on the delta generator and did not take into consideration how long time it would take to apply the path. The reason is that the application takes a lot shorter time (normally just a couple of seconds, regardless of method) and it was not considered as important as the generation. The absolute time difference is assumed to be smaller between the different appliers since it overall is so much faster than the generator.

Some of the previous studies have emphasized the importance of delta updates because it shortens the time it takes to update the car (see section 3). For security reasons, it can be considered important to speed up the update. However, this is most important when the updates are done in-place and the car must be turned off during the process, which is not the case for these A/B Updates (see section 2.4).

Never-the-less, the time it takes to apply the patch, and the consequences this generates
should be investigated and considered when choosing the delta update method. To do so, the applier would have to be implemented in C/C++ as well, instead of Python. This would make it possible to compare it to the applier for bsdiff and see if it is a better solution from that perspective.

6.3.2 Overall updating methods

Other compression methods

This study focused on looking at Android to find an update process that would work for an ECU. Android was considered since the source code is easily found and the scenario is similar to that of a car. It might also be interesting to look at how other projects are creating OTA updates and do similar work on them as has been done here. There might be completely new ways to do it, that were not considered at all in this thesis.

Other verification methods

In this thesis the focus was on dm-verity. There are other ways of verifying a filesystem image. One is fs-verity\(^1\) where each file is verified, instead of the whole image at once. If this was used, it would change the requirements of the update process. This would mean that the patched filesystem image would not have to be identical to the reference image (as described in figure 2.4). Therefore, the update could be done on each file instead of the whole filesystem image. It would actually make it possible to run bsdiff on a single file that needs to be updated.

Security

As mentioned in the chapters 1 and 3, it is important to consider the security in OTA updates. Here secure boot and dm-verity are used to ensure the integrity of the updated image. As discussed in chapter 3, this might not be enough to ensure that the software has not been altered. Therefore, further studies should be done on this subject to ensure that the update process contributes to increasing security and not just becoming another attack surface.

\(^1\)https://www.kernel.org/doc/html/latest/filesystems/fsverity.html
Chapter 7

Conclusions

The goals of the study were fulfilled; the Standalone version of the Android A/B Update was created, the updated image was ensured identical to the backend reference, and the performance was compared to the original Android A/B Update and bsdiff.

In the process it was shown that it is possible to create a standalone version of the Android A/B Update that results in a lot smaller executable and source code than the original version. The different tools and libraries used here gave a satisfying result and are recommended in future implementations. However, some modifications should be done to the Standalone version before it is used. Since the Standalone version suffered from performance loss (compared to the original Android A/B Update), the most important would be to add concurrency to some parts of the code. That way, the process of generating the patch would probably be faster.

If the A/B Update, both the original and the Standalone version, is compared to bsdiff, it is found that the A/B Update is a lot faster when creating the patch, especially for larger images. However, bsdiff seems to create a smaller patch. To decide which one is best to use, the cost for creating the patch has to be weighed against the cost of sending a slightly larger patch to the remote devices over the mobile network. Before making a decision, it should also be taking into consideration which method has the largest opportunity to improve with further work. In this study, the Android version is considered to be the better candidate if the ECU is large enough.
References


REFERENCES


Appendix A

A/B update algorithm

Algorithm

The next version is here called *new image* and the version that is updated is called *old image*.

1. Create a blob file.
2. Create a mapping of all blocks from the new and old image.
3. Assign the blocks (in the new image) that should be zeroed the ZERO-operation.
4. Assign the blocks that are identical in the new and the old image MOVE_COPY-operation.
5. Create a FileDeltaProcessors list.
6. Go through all files in new image:
   (a) If the file has no associated blocks or already has operation; skip.
   (b) Find the corresponding file in the old image if there is any.
   (c) Put the file (and information such as the associated blocks, old image file) in the FileDeltaProcessors list.
7. Add the rest of the blocks to the FileDeltaProcessors list, like a separate file. These are blocks that are not associated with any file.
8. Concurrently for each file in the FileDeltaProcessors list:
   (a) For each chunk of blocks in the file:
      i. Disable BSDIFF and PUFFERDIFF if the data is too large.
ii. Generate the smallest full operation (REPLACE, REPLACE_XZ or REPLACE_BZ)

iii. If the corresponding data in the old file is the same as the new file; change operation to SOURCE_COPY.

iv. Else if a diff operation is better than full operation:
   A. Create BSDIFF operation for the chunk of data.
   B. Create PUFFDIFF operation for the chunk of data.
   C. Chose the smallest operation for the chunk of data.

v. Add the chunk of data to the operation

vi. Write the data (from the full operations or the delta compression operations) to the blob file.

9. For each operation; add the hash of the operation.

10. Write all operations and the blob file to the patch file.

Comments

1.

The blob file contains all the data that are necessary to execute the operation. For example, if the operation is BSDIFF, it requires the patch data to perform a bspatch. Or if the operation is REPLACE, it requires all the new data to write over all data in the blocks.

2.

The mapping is constructed so that blocks with the same data has the same block id and vice versa. Blocks with id 0 is block with all zeroes.

6.b

Levenshtein distance is used to found the corresponding file in the old image. It is supposed that they have similar names and the one with the most similar name is chosen. In the old and the new image, the file names often differ with just a number that represents the version.

8.a

There is a maxsize for how many blocks can be handled by an operation. Therefore, if the file has more blocks associated to it than the max size, it is divided into chunks of blocks.
Appendix B

Included files

B.1 Files from Android source code

These are the files that were included from the original Android source code. Note, however, that most of them has undergone larger or smaller alterations in order to work without the rest of the Android source code.

B.1.1 Patch generator

These are the files that were included from the original source code and used to create the patch.

```
system/update_engine/payload_generator/ab_generator.cc
system/update_engine/payload_generator/ab_generator.h
system/update_engine/payload_generator/annotated_operation.cc
system/update_engine/payload_generator/annotated_operation.h
system/update_engine/payload_generator/blob_file_writer.cc
system/update_engine/payload_generator/blob_file_writer.h
system/update_engine/payload_generator/block_mapping.cc
system/update_engine/payload_generator/block_mapping.h
system/update_engine/payload_generator/bzip.cc
system/update_engine/payload_generator/bzip.h
system/update_engine/payload_generator/delta_diff_utils.cc
system/update_engine/payload_generator/delta_diff_utils.h
system/update_engine/common/error_code.h
system/update_engine/payload_consumer/file_writer.cc
system/update_engine/payload_consumer/file_writer.h
system/update_engine/payload_generator/payload_file.cc
system/update_engine/payload_generator/payload_file.h
```
Appendix B: Included files

external/bsdiff/bsdiff.cc
external/bsdiff/include/bsdiff/bsdiff.h
external/bsdiff/bspatch.cc
external/bsdiff/include/bsdiff/bspatch.h
external/bsdiff/bsdiff/bspatch_main.cc
external/bsdiff/bsdiff/buffer_file.cc
external/bsdiff/bsdiff/buffer_file.h
external/bsdiff/bsdiff/bz2_compressor.cc
external/bsdiff/bsdiff/bz2_compressor.h
external/bsdiff/bsdiff/bz2_decompressor.cc
external/bsdiff/bsdiff/bz2_decompressor.h
external/bsdiff/bsdiff/bzlib.h
external/bsdiff/include/bsdiff/common.h
external/bsdiff/compressor_buffer.cc
external/bsdiff/compressor_buffer.h
external/bsdiff/compressor_interface.h
external/libdivsufsort/android_include/config.h
external/bsdiff/include/bsdiff/constants.h
external/bsdiff/include/bsdiff/control_entry.h
external/bsdiff/decompressor_interface.cc
external/bsdiff/decompressor_interface.h
external/bsdiff/bsdiff/diff_encoder.cc
external/bsdiff/bsdiff/diff_encoder.h
external/bsdiff/bsdiff/extents.cc
external/bsdiff/bsdiff/extents.h
external/bsdiff/bsdiff/file.cc
external/bsdiff/bsdiff/file.h
external/bsdiff/include/bsdiff/file_interface.h
external/bsdiff/bsdiff/memory_file.cc
external/bsdiff/bsdiff/memory_file.h
external/bsdiff/bsdiff/patch_reader.cc
external/bsdiff/bsdiff/patch_reader.h
external/bsdiff/bsdiff/patch_writer.cc
external/bsdiff/bsdiff/patch_writer.h
external/bsdiff/bsdiff/patch_writer_factory.cc
external/bsdiff/include/bsdiff/patch_writer_factory.h
external/bsdiff/include/patch_writer_interface.h
external/bsdiff/bsdiff/sink_file.cc
external/bsdiff/bsdiff/sink_file.h
B.1.2 Patch applier

These are the files that were included from the original source code and used to apply the patch.

scripts/paycheck.py
scripts/update_metadata_pb2.py
scripts/update_payload/__init__.py
scripts/update_payload/applier.py
scripts/update_payload/checker.py
scripts/update_payload/common.py
scripts/update_payload/error.py
scripts/update_payload/format_utils.py
scripts/update_payload/histogram.py
scripts/update_payload/payload.py

B.2 New files

These are the new files that were created in order to make the update process work.

generate_update_main.cc (main file that generates the patch)
WORKSPACE (see appendix C)
BUILD (one for each directory)
Appendix C

WORKSPACE file

This is the content of the WORKSPACE file that is used to specify all external dependencies that should be downloaded.

```python
workspace(name = "workspace")
# load http_archive (used to dynamically download # the other dependencies)
load("@bazel_tools//tools/build_defs/repo:http.bzl", " http_archive")

# Download protobuf
http_archive(  
    name = "com_google_protobuf",
    strip_prefix = "protobuf-master",
    urls = [
        "https://github.com/protocolbuffers/protobuf/ archive/master.zip"
    ],
)
load(  "@com_google_protobuf//:protobuf_deps.bzl",
    "protobuf_deps"
)protobuf_deps()

# Config for rules_foreign_cc
all_content =
```
"""filegroup(
    name = "all",
    srcs = glob(['**']),
    visibility = ['//:visibility:public']
)"
"

# Specifying from where to retrieve rules_foreign_cc
http_archive(
    name = "rules_foreign_cc",
    strip_prefix = "rules_foreign_cc-master",
    url = "https://github.com/bazelbuild/rules_foreign_cc/archive/master.zip",
)

# Load rules and dependencies
load("@rules_foreign_cc//:workspace_definitions.bzl",
    "rules_foreign_cc_dependencies")
rules_foreign_cc_dependencies()

# To build unit tests
http_archive(
    name = "gtest",
    url = "https://github.com/google/googletest/archive/release-1.7.0.zip",
    sha256 = "b58cb7547a28b2c718d1e38ae18a3659c9e3ff52440297e965f5edffe34b6d0",
    build_file = "@//:gtest.BUILD",
    strip_prefix = "googletest-release-1.7.0",
)

# External dependencies used in the updating process
http_archive(
    name = "e2fsprogs",
    build_file_content = all_content,
    strip_prefix = "e2fsprogs-1.45.6",
    urls = ["https://github.com/tytso/e2fsprogs/archive/v1.45.6.tar.gz"],
)

http_archive(
    name = "libdivsufsort",
    build_file_content = all_content,
strip_prefix = "libdivsufsort-2.0.1",
urls = ["https://github.com/y-256/libdivsufsort/
        archive/2.0.1.tar.gz"],
}

http_archive(
    name = "bzip2",
    urls = ["https://sourceware.org/pub/bzip2/bzip2
            -1.0.6.tar.gz"],
    build_file_content = all_content,
    strip_prefix = "bzip2-1.0.6",
)
Appendix D

BUILD file for bsdiff/bspatch

cc_library(
    name = "bsdiff_lib",
    hdrs = [
        "common.h",
        "compressor_interface.h",
        "control_entry.h",
        "patch_writer.h",
        "suffix_array_index.h",
        "bz2_compressor.h",
        "compressor_buffer.h",
        "constants.h",
        "diff_encoder.h",
        "patch_writer_interface.h",
        "suffix_array_index_interface.h",
    ],
    srcs = [
        "bz2_compressor.cc",
        "compressor_buffer.cc",
        "diff_encoder.cc",
        "patch_writer.cc",
        "suffix_array_index.cc"
    ],
    deps = ["libdivsufsort", "logging", "bzip2"],
)

cc_library(
    name = "bspatch_lib",
    hdrs = [ 
"bspatch.h",
"common.h",
"control_entry.h",
"buffer_file.h",
"extents.h",
"extents_file.h",
"file.h",
"file_interface.h",
"patch_reader.h",
"sink_file.h",
"utils.h",
"memory_file.h",
"decompressor_interface.h",
"bz2_decompressor.h",
"constants.h",
],
srcs = [
  "bspatch.cc",
  "file.cc",
  "buffer_file.cc",
  "extents_file.cc",
  "extents.cc",
  "patch_reader.cc",
  "decompressor_interface.cc",
  "sink_file.cc",
  "utils.cc",
  "memory_file.cc",
  "bz2_decompressor.cc",
],
deps = ["logging", "bzip2"],
visibility = ["//test:__pkg__"],
)

cc_library(
  name = "bsdiff",
  hdrs = ["bsdiff.h", "patch_writer_factory.h"],
  srcs = ["bsdiff.cc", "patch_writer_factory.cc"],
  deps = ["bsdiff_lib", "logging"],
  visibility = ["//test:__pkg__", "//:__pkg__"],
)

cc_binary(
name = "bspatch",
srcs = ["bspatch_main.cc"],
deps = ["bspatch_lib"],
visibility = ["//test:__pkg__", "//scripts:__pkg__"
                 ],
)
# Appendix E

## Hashes

<table>
<thead>
<tr>
<th>Name</th>
<th>sha256sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>picore 1</td>
<td>78156c3a1b0b60057369f0c8daeaad4449382512d30a8da02b0336eeccc64754464</td>
</tr>
<tr>
<td>picore 2</td>
<td>3e49399303ce22f8d01023959802523b7a872dcac8ba8b98c9f5ed9f6ebc66</td>
</tr>
<tr>
<td>openwrt 1</td>
<td>8883d7f124d1c3b85552891c3575aa45420920453930b3cc5e9c9a96ed29b4</td>
</tr>
<tr>
<td>openwrt 2</td>
<td>be524df08f03a3a3e1323c0f7f8a7580f048e1156687715096c3b672</td>
</tr>
<tr>
<td>raspbian 1</td>
<td>993e8b66a53c87154e90d60191691872e3bf520874471164526a90399</td>
</tr>
<tr>
<td>raspbian 2</td>
<td>5db101c2a369309aadd34153cb0d6c5ac66d9937903870aef71d014d2d995cf0d</td>
</tr>
</tbody>
</table>

Table E.1 – sha256 hash after the ext2/ext4 images were extracted