BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PORTING TANG TO OPENWRT

PORTOVANIE TANG NA OPENWRT

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TIBOR DUDLÁK**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDREJ LICHTNER**
VEDOUCÍ PRÁCE

BRNO 2018

Abstract

This thesis describes the encryption and its application to secure the computer's hard drive. It describes the structure of the encrypted disk's partition according to the LUKS specification on Linux operating systems. The thesis focus on describing possibilities of automating the disk decryption process using an external server that enters the process as a third party. It describes the principles of Key Escrow and Tang server. Steps required to compile and configure the Tang server are described too. Also described is Tang server's client – Clevis. The main objective of this work was to port and document the process of porting the Tang server and its dependencies to OpenWrt system, described in this thesis too, which is designed for embedded devices such as WiFi routers. The thesis also includes a documented process of contributing changes and newly created OpenWrt packages to corresponding Open Source projects.

Abstrakt

Hlavným cieľom tejto práce je sprístupnenie serveru Tang na vstavané zariadenia typu WiFi smerovač s plne modulárnym operačným systémom OpenWrt. Tým dosiahneme anonymnú správu šifrovacích kľúčov pre domáce siete a siete malých firiem. Preto táto práca popisuje problematiku šifrovania a jeho využitie na zabezpečenie pevného disku počítača. Oboznámuje čitateľa so štruktúrou šifrovaného diskového oddielu podľa LUKS špecifikácie na operačných systémoch typu Linux. Práca rozoberá možnosti automatizácie odomykania šifrovaných diskov použitím externého servera, ktorý vstupuje do procesu ako tretia strana. Sú v nej popísané princípy serverov Key Escrow a Tang. Dosiahnutie hlavného cieľa je možné vďaka procesu portovania a cross-kompilácie na platforme Linux. Práca obsahuje zdokumentovaný postup prispievania zmien a novo vytvorených balíkov pre OpenWrt do príslušných Open Source projektov.

Keywords

porting, Tang, server, Clevis, client, Escrow, OpenWrt, operating system, embedded device, encryption, LUKS, hard drive, disk partition, encryption key, automation, cross-compiling, package system

Kľúčové slová

portovanie, Tang, server, Clevis, klient, Escrow, OpenWrt, operačný systém, vstavané zariadenie, šifrovanie, LUKS, pevný disk, diskový oddiel, šifrovací kľúč, automatizácia, cross-kompilácia, balíkový systém

Reference

DUDLÁK, Tibor. *Porting Tang to OpenWrt.* Brno, 2018. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondrej Lichtner

Porting Tang to OpenWrt

Declaration

Hereby I declare that this term project was prepared as an original author's work under the supervision of Ing. Ondrej Lichtner. The supplementary information was provided by Jan Pazdziora, Ph. D. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Tibor Dudlák
May 8, 2018

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Contents

Chapter 1

Introduction

We spend our time searching for security, and hate it when we get it.

John Steinbeck[22]

Nowadays, the whole world uses information technologies to communicate and to spread knowledge in form of bits to the other people. But there are pieces of personal information such as photos from family vacation, videos of our children as they grow, contracts, testaments which we would like to protect.

Encryption, as described in chapter 2 How we use Encryption, protects our data and privacy even when we do not realize that. An unauthorized party may be able to access secured data but will not be able to read the information from it without the proper key. With an increasing number of encryption keys to store and protect, there might be necessary to consider using Key management server. One of the possible solutions for persistent Key management is to deploy *key escrow* server described in section 3.1Key Escrow. Another solution is server *Tang*, which principles are mentioned in section 3.2 Tang server.

Tang is completely anonymous key recovery service aiming to solve an early boot decryption of the system volumes encrypted with LUKS specification described in section **??** Disk encryption with LUKS. In contrast to Key Escrow server, Tang does not know any key. It only provides mathematical operation for its clients to recover them.

The goal of this bachelor thesis is to port the *Tang* server, and its dependencies listed in section 3.2.2 to the OpenWrt system. *OpenWrt*, characterized in section 4.1 OpenWrt system, is Linux based operating system for *embedded* devices such as Wireless routers. Porting packages for the OpenWrt as described in chapter4 Software portability is done with cross-compilation tools available from OpenWrt's buildroot.

The process of porting missing dependecies for the Tang server is described in chapter 5 Porting the dependencies. Work required for the Tang itself is divided into chapter 6 Porting Tang and chapter 7 Tang on OpenWrt.

Section 7.5 Tang's limitations sums up not only the limitations that are present on OpenWrt platform but also a limitation that was discovered while testing solution.

With accomplishing thesis goal, we will be able to automatize process of unlocking encrypted drives on our private home or small office network, therefore securing our data stored on personal computer's hard drive and/or NAS (Network-attached storage) server if it is stolen. There will be no need for any decryption or even Key Escrow server but the *OpenWrt* device running the *Tang* server itself only.

Chapter 2

How we use Encryption

We may not realize this, but we use encryption every day. The purpose of encryption is to keep us safe when we are browsing the internet or just storing our sensitive information on digital media. In general encryption is used to secure our data, whether transmitted around the internet or stored on our hard drives, from being compromised. Encryption protects us from many threats.

It protects us from identity theft. Our personal information stored all over governmental authorities should be secured with it. Encryption takes care for not revealing sensitive information about ourselves, to protect our financial details, passwords along with others, mainly when we bank online from being defraud.

It looks after our conversation privacy. To be more specific, our cell phone conversations from eavesdroppers and our online chatting with acquaintances or colleagues. It also allows attorneys to communicate privately with their clients and it aims to secure communication between investigation bureaus to exchange sensitive information about lawbreakers.

If we encrypt our laptop or desktop computer's hard drive, encryption protects our data in case the computer or hard drive is stolen.

2.1 Encryption and security

Security is not binary; it is a sliding scale of risk management. People are used to mark things, for example good and bad, expensive, and cheap. But we know that people may differ on image/sense. For example, there is no such thing as line or sign which tells us that this part of town is secure, and this is not. The way we reason about security is that we study environment entering or observing it, and we begin to decide whether it is secure or not. Especially at enterprise sector. Encryption, on its own, might not be enough to make our data or infrastructure secure but is definitely a critical aspect of security.

Companies define their own security strategies which may include encryption or may not at all. Security strategies rely on company's needs or will to take risks in conclusion of getting high gain from them.

According to 2016 Global Encryption Trends Study, independently conducted by the Ponemon Institute, the enterprise-wide encryption in 5 years increased from 15 to 38 percent. Also, the ratio of companies with no encryption strategy at all decreased from 37 to 15 percent. More than 50 percent of companies are using extensively deployed encryption technologies to encrypt mostly databases, infrastructure and laptop hard drives [24].

Passwords are the most common authentication method used for accessing computer systems, files, data, and networks. It is important to keep changing them in reasonable time and as a secret to others. Still, no matter the company's security strategy, we keep seeing them on monitors or desktops written on sticky notes, and this is absolutely not secure. In fact, users are the most vulnerable part of securing our systems. To aid their memory, users often include part of a phone number, family name, Social Security number, or even birth date in their passwords [26]. They choose cryptographicaly weak passwords, dictionary words, which are easy to remember but also easy to guess or to break with brute-force attacks in short period of time. According to Splash Data, a supplier of security applications, the most common user selected password in the year 2016 was „123456". They claim that people continue to put themselves at risk for hacking and identity theft by using weak, easily guessable passwords. To create strong password we can follow any trustworthy guide[1] on the Internet. More secure way to create passwords would be to generate cryptographically stronger cipher and use it as a password. The only disadvantage is that it is hard to remember [21].

2.2 Hard drive encryption

It all starts, as mentioned, with desire to keep our data to ourselves and as a secret to others. More often than not, these secrets are stored on our hard drives.

Hard drive encryption is a technology provided by software performing sophisticated mathematical functions or hardware that encrypts the data stored on a hard drive or a disk volume. This technology is used to prevent unauthorized access by unauthorized persons or service to an encrypted data storage without possesion of the appropriate key or password. Encrypting the hard drive means providing another layer of security against hackers and other online threats.

To protect secret data, we usually encrypt this data by using an „encryption key" - see Figure 2.1 Hard drive encryption in a nutshell. Every encryption key should be unpre-

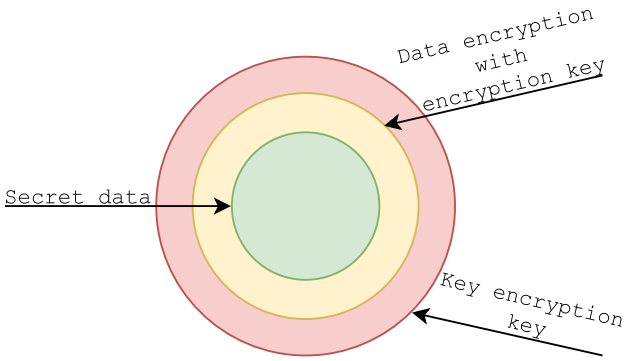


Figure 2.1: Hard drive encryption in a nutshell

dictable and unique set of bits able to „scramble" data in a way that it would be impossible to recover data whithout it. To satisfy this need, encryption keys are generated by specialized algorithms such as AES (Advanced Encryption Standard[2]) for symmetric keys and

[1] https://www.wikihow.com/Create-a-Secure-Password
[2] https://en.wikipedia.org/wiki/Symmetric-key_algorithm#Implementations

RSA (Rivest–Shamir–Adleman[3]) for asymmetric. Changing encryption key implies the whole data decryption with an old key and encryption with new encryption every time the old one is compromised or a change is required. Also this secret data might grow in size, and it is time and resource consuming to decrypt and encrypt the volume or even the whole hard drive. Because of that, the encryption key is then wrapped by the key encryption key.

Key encryption key is mostly generated using the user provided password. This key encryption key is then used to encrypt encryption key which does the actual data encryption. Again, the most unsecure thing in this would be the user provided password which we can easily replace with using only cryptographically stronger key. This principle has at least two advantages. Changing the key encryption key does not affect encrypted data, and key can be changed whenever the user desires to, and redistributed to all users or services that are supposed to access this data.

Hard drive could be encrypted as whole or per partition. Full disk encryption is done in a way that all content on the hard drive except MBR(Master Boot Record) is encrypted. Encrypting MBR would make it impossible to start boot sequence of operating system. Boot sequence would prompt the user for key encryption key in order to load the operating system from encrypted storage.

This could disrupt our daily basis and might be the reason why most of us do not use hard drive encryption, even when we know it will protect our data. There is a way to automate the hard drive unlocking on early boot with a help from key management system. We can get the key encryption key from some remote system, the Escrow server mentioned in section 3.1 Escrow server, or recover it with Tang described in chapter 3.2 Tang server. Before that, let us have a look on most common hard drive encryption implementations.

2.2.1 Bit Locker

BitLocker is a closed source full disk data protection feature that integrates with the operating system Windows Vista and later. It is designed to protect data by providing encryption for entire volumes and addresses the threats of data theft or exposure from lost, stolen, or inappropriately decommissioned computers. By default, it uses the AES encryption algorithm in cipher block chaining (CBC) or XTS mode with a 128-bit or 256-bit key. CBC is not used over the whole disk; it is applied to each individual sector.

BitLocker supports TPM (Trusted Platform Module). The TPM is a hardware component installed in many newer computers by the computer manufacturers. It works with BitLocker to help protect user data and to ensure that a computer has not been tampered with while the system was offline[15].

2.2.2 LibreCrypt

LibreCrypt is a LUKS (Linux Unified Key Setup-on-disk-format) compatible open source "on-the-fly" transparent disk encryption software written mostly in Pascal programming language. This project is based on original FreeOTFE project by Sarah Dean renamed in version 6.2 to LibreCrypt and supports both 32 and 64 bit Windows. LibreCrypt is easy to use even for unexperienced user through its GUI (Graphical User Interface) with support of many languages.

[3]https://en.wikipedia.org/wiki/Public-key_cryptography

LibreCrypt support many ciphers including AES (up to 256 bit), Twofish[4] (up to 256 bit), Blowfish[5] (up to 448 bit), Serpent[6] (up to 256 bit)[**?**]. It can create "virtual disks" on our computer and anything written to these disks is automatically encrypted before being stored on our computer's hard drive[30].

Unfortunatelly this project seems to be abandoned by its developer on github. The source code of LibreCrypt is available on GitHub. We can visit the site or clone source code using git:

```
git clone https://github.com/t-d-k/LibreCrypt.git
```

2.2.3 LUKS

LUKS (Linux Unified Key Setup-on-disk-format) is a platform-independent disk encryption specification. LUKS was created by Clemens Fruhwirth in 2004 and was originally intended for Linux distributions only. It provides a standard on-disk-format for hard disk encryption, which facilitates compatibility among Linux distributions and provides secure management of multiple user passwords.

Referential implementation of LUKS is using a device-mapper crypt target (dm-crypt) subsystem for bulk data encryption. This subsystem is not particularly bound to LUKS and can be used for plain format encryption.

It is important for us to know this specification a little more due to working with Linux distribution and the implemantation of the Tang server.

2.3 Disk encryption with LUKS

[[merge and add motivation related to unlockers and tang]]

With modern versions of cryptsetup (i.e., since 2006), encrypted block devices can be created in two main formats, plain dm-crypt format or the extended LUKS format.

Plain format is just that: It has no metadata on disk. When using any such encrypted device, all the necessary parameters must be passed to cryptsetup from the commandline (otherwise it uses the defaults, which will only succeed if the device was created using default settings). It derives (generate) the master key from the passphrase provided and then uses that to decrypt or encrypt the sectors of the device, with a direct 1:1 mapping between encrypted and decrypted sectors.

In contrast to previous Linux disk encryption solutions, LUKS stores all necessary setup information in the partition header, enabling the user to transport or migrate their data more easily. LUKS uses the dm-crypt subsystem. Device-mapper is a part of the Linux kernel that provides a generic way to create virtual layers of block devices, most commonly LVM (Logical Volume Manager) logical volumes. The device-mapper crypt target (dm-crypt) provides transparent encryption of block devices using the kernel crypto API (Application programming interface) supporting ciphers and digest algorithms via standard kernel modules.

In Fedora and Red Hat Enterprise Linux, userspace interaction with dm-crypt is managed by a tool called cryptsetup, which uses the device-mapper infrastructure to setup and operate on encrypted block devices.

[4] https://en.wikipedia.org/wiki/Twofish
[5] https://en.wikipedia.org/wiki/Blowfish_(cipher)
[6] https://en.wikipedia.org/wiki/Serpent_(cipher)

The advantages of LUKS over plain dm-crypt are the higher usability: automatic configuration of non-default crypto parameters, the ability to add, change, and remove multiple passphrases. Additionally, LUKS offers defenses against low-entropy passphrases like salting and iterated PBKDF2[7] (Password-Based Key Derivation Function 2) passphrase hashing. With LUKS, encryption keys are always generated by the kernel RNG (Random number generator); in contrast to plain dm-crypt where one can choose, e.g. a simple dictionary word and have an encryption key derived from that. One disadvantage of using LUKS over plain is that it is readily obvious there is encrypted data on disk; the other is that damage to the header or key slots usually results in permanent data loss. To mitigate this risk the Backup of the LUKS header would be best option [5].

2.3.1 Creating LUKS volume

Creating just a LUKS volume is quite easy. All we need is a tool called cryptsetup. We can install it on Fedora system using command:

```
# dnf install cryptsetup
```

Note, that this command has to be run by the user with root privileges. After installation succeeds, all we need to do is to choose a partition to encrypt. For demonstration, we will encrypt the */dev/xvdc* patrtition using:

```
# cryptsetup -y -v luksFormat /dev/xvdc
WARNING!
========
This will overwrite data on /dev/xvdc irrevocably.

Are you sure? (Type uppercase yes): YES
Enter LUKS passphrase:
Verify passphrase:
Command successful.
```

But having full disk encryption if the system is already installed might be quite tough. Hard drive partition must contain the LUKS header just before the encrypted data. Lets sum up the easier way first.

In case we have not installed our Linux operation system yet, we could simply select an option in time of installation. Then the installation wizard will most likely asks for pass phrase - the key encryption key. To demonstrate this, screen shots with Fedora 25 system installation can be found on appendix B Pre-istallation enablement of hard drive encryption.

If we have already system installed with lots of data on partition, process will probably last longer and the procedure will be more complex. There is no way we can encrypt the whole system disk with LUKS without unmounting a partition to encrypt. For this purpose, the LUKS In-Place Conversion Tool *luksipc* was developed. Steps to encrypt a disk using *luksipc* are in the appendix C LUKS In-Place Encryption.

2.3.2 LUKS drive structure

A hard drive with a LUKS partition has notable structure see Figure 2.2 LUKS volume structure for a demonstration. The LUKS format uses a metadata header, also marked as *phdr*, and 8 key slot areas at the the beginning of the device. After header, there is a section with bulk data, which are encrypted with the encryption key. Header contains

[7] https://en.wikipedia.org/wiki/PBKDF2

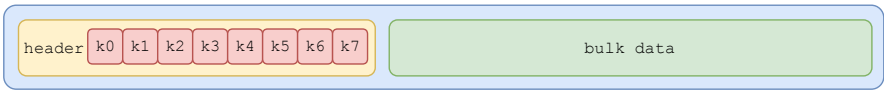


Figure 2.2: LUKS volume structure

information about the used cipher, cipher mode, the key length, a uuid and a master key checksum. The passphrases stored in the key slots, which structure is shown on Figure 2.3 LUKS Key Slot, are used to decrypt a single key encryption key that is stored in the anti-forensic stripes.

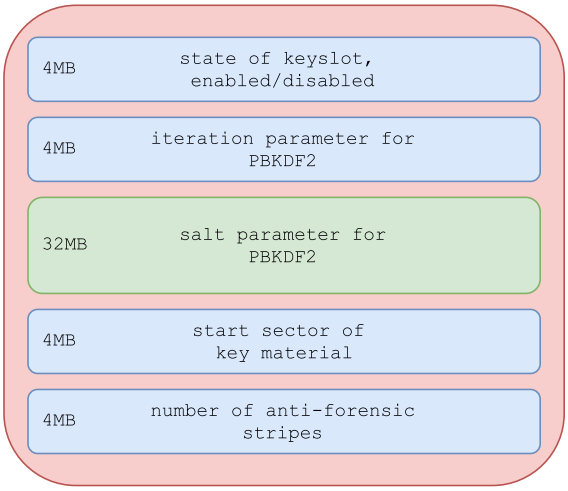


Figure 2.3: LUKS Key Slot

Every key slot is associated with a key material section after the header. When a key slot is active, the key slot stores an encrypted copy of the master key in its key material section. This encrypted copy is locked by a user password or cipher. Supplying this user password unlocks the decryption for the key material, which stores the master key.

2.3.3 Managing LUKS volume

This is a demonstration of how managing LUKS volume could look like in reality, in our case on our „host“ Fedora 27 system. For this demonstration we will use tools available for Fedora 27 such as:

- parted-3.2-28.fc27.x86_64
- cryptsetup-1.7.5-3.fc27.x86_64

These tools are available through Fedora 27 upstream repositories and can be easily downloaded and installed via Fedora package management system the dnf. To list hard drives for our system we will use the parted tool:

```
# parted -l
Model: NVMe Device (nvme)
Disk /dev/nvme0n1: 256GB
Sector size (logical/physical): 512B/512B
```

```
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type      File system  Flags
 1      1049kB  1075MB  1074MB  primary   ext4         boot
 2      1075MB  256GB   255GB   primary}
```

The exhibit shows that on our „host“ system we have installed an NVMe (NVM Express) volume[8] using the msdos, therefore MBR partition table. The NVMe is a specification for accessing SSDs attached through the PCI Express bus. Our SSD drive has two partitions, as we can see, the first is a boot partition and second should be our LUKS partition. To find out if the second partition is LUKS volume, we will use cryptsetup tool with option isLuks:

```
# cryptsetup isLuks /dev/nvme0n1p1 -v
Device /dev/nvme0n1p1 is not a valid LUKS device.
Command failed with code 22: Invalid argument
```

This command has no /std/out nor /std/err output unless the -v option is used. The actual result of the command is returned as an exit code. The output shown reflects that our boot partition is not a valid LUKS volume as expected. This is expected behavior, encrypting the MBR partition would make it impossible for system to boot.

```
# cryptsetup isLuks /dev/nvme0n1p2 -v
Command successful.
```

At this point, we have successfully identified the LUKS volume. To dump the header information of this LUKS volume, use cryptsetup option luksDump:

```
# cryptsetup luksDump /dev/nvme0n1p2
LUKS header information for /dev/nvme0n1p2

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
Payload offset: 4096
MK bits:        512
MK digest:      85 b3 b9 71 b6 b7 51 18 60 39 78 db ac e8 82 97 0c 7b a2 3e
MK salt:        0d 22 53 83 56 0d a0 70 25 c2 bf fe 75 40 71 a9
                75 f1 ae a3 67 e5 b2 a5 14 85 39 1d c6 74 00 a8
MK iterations:  52625
UUID:           267c308e-5d64-4acf-abf2-f6e224e8febf

Key Slot 0: ENABLED
    Iterations:             415583
    Salt:       3e f9 7d 3b b6 08 60 9a eb dc 52 bb 8e 21 eb bf
                b9 4d 80 a4 70 2d 4e 97 8e 47 c1 a3 04 45 74 d4
    Key material offset:    8
    AF stripes:             4000
Key Slot 1: DISABLED
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

[8] https://en.wikipedia.org/wiki/NVM_Express

From the listing we can assume that /dev/nvme0n1p2 has one key encryption key in key slot 0 and has other 7 slots (1-7) disabled. The advantage of LUKS is that if a team or group of up to eight people has to work with a common encrypted volume, each team member may use his own password. New user keys may be added to the encrypted volume and old user keys may be removed. To add a new key, we will use cryptsetup with option luksAddKey as shown:

```
# cryptsetup luksAddKey  /dev/nvme0n1p2
Enter any existing passphrase:
Enter new passphrase for key slot:
Verify passphrase:
```

The cryptsetup will derive the key encryption key from this passphrase and bind it to the first available key slot in LUKS header. The dump of header will now contain information similar to the exhibit below:

```
# cryptsetup luksDump /dev/nvme0n1p2
LUKS header information for /dev/nvme0n1p2

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
Payload offset: 4096
MK bits:        512
MK digest:      85 b3 b9 71 b6 b7 51 18 60 39 78 db ac e8 82 97 0c 7b a2 3e
MK salt:        0d 22 53 83 56 0d a0 70 25 c2 bf fe 75 40 71 a9
                75 f1 ae a3 67 e5 b2 a5 14 85 39 1d c6 74 00 a8
MK iterations:  52625
UUID:           267c308e-5d64-4acf-abf2-f6e224e8febf

Key Slot 0: ENABLED
    Iterations:             415583
    Salt:       3e f9 7d 3b b6 08 60 9a eb dc 52 bb 8e 21 eb bf
                b9 4d 80 a4 70 2d 4e 97 8e 47 c1 a3 04 45 74 d4
    Key material offset:    8
    AF stripes:             4000
Key Slot 1: ENABLED
    Iterations:             1247257
    Salt:       44 48 92 df 29 8a df 81 f6 44 f8 66 c5 c2 32 49
                23 76 8a 37 48 85 33 2a 29 10 d8 cc 8f 45 0a 46
    Key material offset:    1520
    AF stripes:             4000
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

At this point, LUKS volume is accessible via two different passphrases. To remove any key from LUKS header we will use cryptsetup with option luksKillSlot as shown:

```
# cryptsetup luksKillSlot /dev/nvme0n1p2 0
Enter any remaining passphrase:
```

To successfully remove key from key slot 0, we have to enter any remaining passphrase. In our case, we would enter the newly created passphrase which is bound to key slot 1. This will result into having LUKS header in state like shown:

```
# cryptsetup luksDump /dev/nvme0n1p2
LUKS header information for /dev/nvme0n1p2

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
Payload offset: 4096
MK bits:        512
MK digest:      85 b3 b9 71 b6 b7 51 18 60 39 78 db ac e8 82 97 0c 7b a2 3e
MK salt:        0d 22 53 83 56 0d a0 70 25 c2 bf fe 75 40 71 a9
                75 f1 ae a3 67 e5 b2 a5 14 85 39 1d c6 74 00 a8
MK iterations:  52625
UUID:           267c308e-5d64-4acf-abf2-f6e224e8febf

Key Slot 0: DISABLED
Key Slot 1: ENABLED
    Iterations:             1247257
    Salt:       44 48 92 df 29 8a df 81 f6 44 f8 66 c5 c2 32 49
                23 76 8a 37 48 85 33 2a 29 10 d8 cc 8f 45 0a 46
    Key material offset:    1520
    AF stripes:             4000
Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

Password rotation is very important. To change LUKS key slot passphrase use the cryptsetup's option luksChangeKey:

```
#cryptsetup luksChangeKey /dev/nvme0n1p2 -S 1
Enter passphrase to be changed:
Enter new passphrase:
Verify passphrase:
```

Note that passphrase to be changed and the slot number must be related to each other. We could see a change in LUKS header after command succeeded:

```
# cryptsetup luksDump /dev/nvme0n1p2
LUKS header information for /dev/nvme0n1p2

Version:        1
Cipher name:    aes
Cipher mode:    xts-plain64
Hash spec:      sha256
Payload offset: 4096
MK bits:        512
MK digest:      85 b3 b9 71 b6 b7 51 18 60 39 78 db ac e8 82 97 0c 7b a2 3e
MK salt:        0d 22 53 83 56 0d a0 70 25 c2 bf fe 75 40 71 a9
                75 f1 ae a3 67 e5 b2 a5 14 85 39 1d c6 74 00 a8
MK iterations:  52625
UUID:           267c308e-5d64-4acf-abf2-f6e224e8febf

Key Slot 0: DISABLED
Key Slot 1: ENABLED
    Iterations:             1630572
    Salt:       cf 54 a9 77 ed 8b c5 75 ca 65 60 6b 31 cb 29 0f
                4e 54 78 8c b1 9a db 3f 2f 6c aa 84 79 da 81 66
```

```
    Key material offset:    512
    AF stripes:             4000

Key Slot 2: DISABLED
Key Slot 3: DISABLED
Key Slot 4: DISABLED
Key Slot 5: DISABLED
Key Slot 6: DISABLED
Key Slot 7: DISABLED
```

For more usecases and examples how to use cryptsetup see the man page of the tool[9]. Examples shown demonstrate basic key management for LUKS volumes and should be sufficient for basic understanding of cryptsetup/LUKS behavior and usability on the system with an encrypted drive.

2.3.4 LUKS security

In August 2012 Ubuntu Privacy Remix Team did a deep analysis of LUKS/cryptsetup in Ubuntu enviroment. Cryptsetup in version 1.4.1 used by Ubuntu 12.04 LTS has been chosen for they analysis. They wrote the programs luksanalyzer and hashtest for the purpose of the analysis.

At the end, they came into conlusion that cryptsetup with LUKS is a highly secure program for encrypting whole data media or partitions thereupon. The encryption algorithms, and other security mechanism it implements, comply with the current state of the art in cryptography. They found no back door or security-related mistake in the published source code. If we use this program in a secure environment, we may assume with high certainty that no one can get access to the data stored in our volumes as long as they are closed, the passwords are really good and the attacker doesn't apply highly advanced methods below the layer of the operation system, such as BIOS rootkits, hardware keyloggers or video surveillance. A special strong point of Cryptsetup with LUKS is its high power of resistance against dictionary attacks. This resisting power is adapted to the increased speed of hardware when new encrypted volumes are created on new up to date hardware[23].

[9] https://linux.die.net/man/8/cryptsetup

Chapter 3

Atomated decryption

People usually try to automate everything they can in order to avoid repetitive tasks. Even though disc enctyption provides another layer of security to our data, it is not used often. Typing one more passphrase when accessing some removable storage seems to bother. More crucial would be full disk encryption. In order to boot the system, we need to have storage with system decrypted - provide another passphrase every time the computer is turned off and on again. The Tang server aims to solve struggles with the early boot decryption of system volumes. Before Tang, automated decryption was usually handled by a Key Escrow server.

3.1 Key Escrow

The Key Escrow server (also known as a "fair" cryptosystem) is providing escrow service for encryption keys. A client using Key Escrow usually generates a key, encrypts data with it and then stores the key encryption key on a remote server. Unfortunatelly, it is not always as simple as it sounds and there are couple of security concerns.

To transfer the encryption keys we want to store on an Escrow server, we have to encrypt the channel on which we send them. When transmitting keys over an insecure network without an encrypted link, anyone listening to the network traffic could immadiately fetch the key. This should signal security risks, and, of course, we do not want any third party to access our secret data. Usually we encrypt a channel with TLS (Transport Layer Security) or GSSAPI (Generic Security Services Application Program Interface) as shown on a Figure 3.1 Escrow model. Unfortunatelly, this is not enough to have the communication secure.

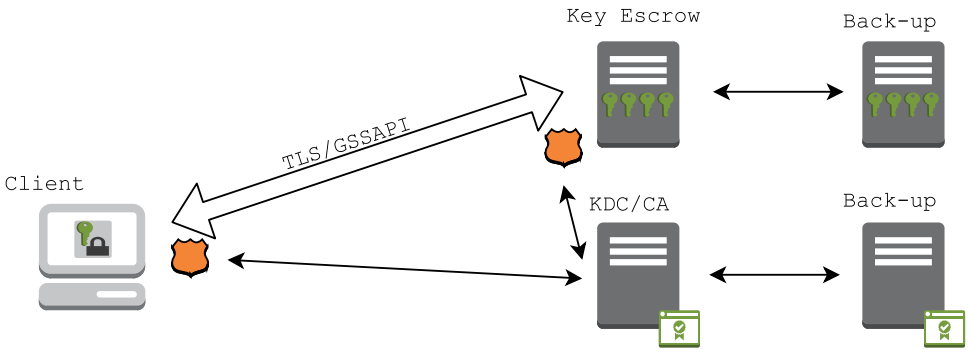


Figure 3.1: Escrow model

We can not just start sending these keys to the escrow server if we do not know whether this server is the one it acts to be. This server has to have its identity verified, and the client has to authenticate to this server too. The increasing amount of keys implicates a need for Certification Authority server (CA) or a Key Distribution Center (KDC) to manage all of them.

With all these keys, and at this point only, the server can verify if the client is permitted to get their key, and the client is able to identify a trusted server. This is a fully stateful process. To sum up, an authorized third party may gain access to keys stored on an Escrow server under certain circumstances only.

The complexity of this system increases the attack surface and for a such complex system it would be unimaginable not to have backups. The Escrow server may store lots of keys from lots of different places, users and services.

With key escrow, a third party gets copies of a cryptographic key. People might not be comfortable with any third party having this ability and that „technical“ problems vex the key escrow solution. Tang's key recovery, on the other hand, lets us just „backup“ and restore cryptographic keys anonymously and without any third party possesing our key.

Key recovery is necessary. Key escrow is not. Let us not confuse these two.

3.2 Tang server

Tang is a very lightweight network service using systemd's socket activation as described in section 6.1 Socket activation. Its purpose is to provide anonymous key recovery to clients all over the accessible network. This key recovery can be used with a help of the Tang's client descibed in subsection 3.2.4 Clevis client to unlock data storages with LUKS encryption. The Tang server is an open source project implemented in C programming language. Let us see how it works.

We can see on Figure 3.2 Tang Model that the model is very similar to the escrow seen on Figure 3.1 Escrow model, but with some things missing. In fact, there is no longer

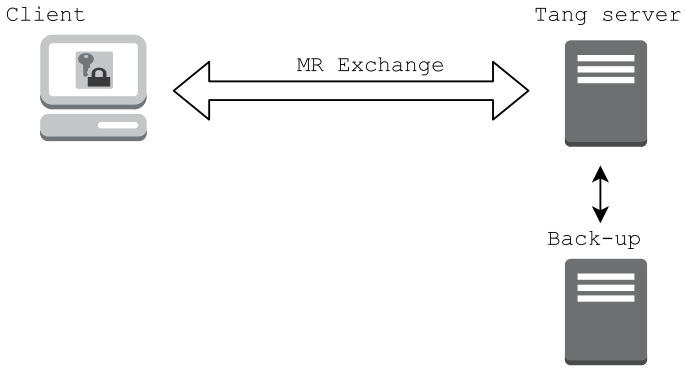


Figure 3.2: Tang model

need for TLS channel to secure comunication between the client and the server, and that is because Tang implements the McCallum-Relyea exchange as described below.

Tang server advertises asymmetric keys on the network and a client is able to get the list of these signing keys by HTTP (Hypertext Transfer Protocol) GET request or even using curl:

```
curl -f http://tang.local/adv
```

All tang comunication is performed using HTTP protocol with the message body in JWK(JSON Web Key) format defined by RFC 7517 [7]. Getting advertised keys is the first step of the McCallum-Relyea exchange protocol summed up in table 3.1 McCallum-Relyea exchange protocol. This protocol has two phases provisioning and recovery.

Provisioning		Recovery	
client's side	server's side	client's side	server's side
	$S\epsilon_{bindingR}[1, p-1]$ $s = g * S$	$E\epsilon_R[1, p-1]$ $x = c + g * E$	
$\leftarrow$ s		x $\rightarrow$	
$C\epsilon_R[1, p-1]$ $e = g * C$		$\leftarrow$ y	$y = z * S$
$K = g * S * C = s * C$ Discard: K, C Retain s, c		$K = y - s * E$	

*capital is private key; g stands for generate

Table 3.1: McCallum-Relyea exchange protocol

Provisioning The server key pair generation is represented by equation , capital is private key; lowercase is public key; g stands for generate.

$$s = g * S \tag{3.1}$$

After the server generates key pair, it is advertising the public part of it. The client then selects one of the Tang server's exchange keys (we will call it sJWK; identified by the use of deriveKey in the sJWK's key_ops attribute). The lowercase „s“ stands for server's public key and JWK is used format of the message. The client then generates a new (random) JWK (cJWK; c stands for client's key pair).

$$c = g * C \tag{3.2}$$

The client performs its half of a standard ECDH exchange producing dJWK which it uses to encrypt the data. Afterwards, it must discard the dJWK and the private key from cJWK. The dJWK is represented as K in equation 3.3.

$$K = s * C \tag{3.3}$$

The client has to store cJWK for later use in the recovery step. Generally speaking, the client may also store other data, such as the URL of the Tang server or the trusted advertisement signing keys called also binding keys.

Recovery Mathematically capital is private key; g stands for generate. When the client wants to access the encrypted data, it must be able to recover encryption key. To recover dJWK after discarding it, the client generates a third ephemeral key (eJWK) as the equation 3.4 represents. This key is used to hide the client's public key and the binding key.

$$e = g * E \tag{3.4}$$

The ephemeral key is generated for each execution of a key establishment process particularly. Using eJWK, the client performs elliptic curve group addition of eJWK and cJWK, producing xJWK represented in the equation 3.5. The client POSTs xJWK to the server.

$$x = c + e \tag{3.5}$$

The server then performs its half of the ECDH key exchange using xJWK and sJWK, producing yJWK reflected in the equation 3.6. The server returns yJWK to the client.

$$y = x * S \tag{3.6}$$

The client then performs half of an ECDH key exchange between eJWK and sJWK, producing zJWK as the equation 3.7 shows.

$$z = s * E \tag{3.7}$$

Subtracing zJWK from yJWK produces dJWK represented as K in the equation 3.8 again.

$$K = y - z \tag{3.8}$$

Finally, the client has calculated the key value, which is better than when the server calculates it. So the Tang server never knew the value of the key and literraly nothing about its clients.

3.2.1 Security

As shown in table 3.2 Comparing Escrow and Tang, Tang compared to Escrow is stateless and doesn't require TLS or authentication. Tang also has limited knowledge. Unlike escrows, where the server has knowledge of every key ever used, Tang never sees a single client key. Tang never gains any identifying information from the client.

	Escrow	Tang
Stateless	No	Yes
SSL/TLS	Required	Optional
X.509	Required	Optional
Authentication	Required	Optional
Anonymous	No	Yes

Table 3.2: Comparing Escrow and Tang

Thanks to McCallum-Relyea exchange protocol summed up in the table 3.1 McCallum-Relyea exchange protocol Tang is resistant to the man in the middle attack. In case of the eavesdroppers, they see the client send xJWK and receive yJWK. Since, these packets are blinded by eJWK, only the party that can unblind these values is the client itself (since only it has eJWK's private key). Thus, the MitM attack fails.

It is of utmost importance that the client protects cJWK from prying eyes and the Tang server must protect the private key for sJWK.

3.2.2 Building Tang

Tang is originally packaged for Fedora OS version 23 and later but we can build it from source of course. It relies on few other software libraries:

- http-parser
- systemd's socket activation
- jose
  - jansson
  - openssl
  - zlib

The steps to build the Tang from sources include downloading the source from poject's GitHub or cloning it. Make sure we have all required dependencies installed and then run:

```
$ autoreconf -if
$ ./configure --prefix=/usr
$ make
# make install
```

Optionally to run tests:

```
$ make check
```

3.2.3 Server enablement

Enabling a Tang server is a two-step process. First step is to enable the Tang services. Start the key update service which is watching the database directory using systemd. After the database of the keys is changed the update service will regenerate the cache used by tang:

```
# systemctl enable tangd-update.path --now
```

Enable service using systemd socket activation:

```
# systemctl enable tangd.socket --now
```

After this, systemd will handle the sockets and the Tang's key rotation procedure.

Secondly, generate a signing key using dependency tool jose and store it in a default directory expected by tang:

```
# jose gen -t '{"alg":"ES256"}' -o /var/db/tang/sig.jwk
```

Do not forget to generate an exchange key:

```
# jose gen -t '{"kty":"EC","crv":"P-256","key_ops":["deriveKey"]}' \
        -o /var/db/tang/exc.jwk
```

Now we are up and running. Keys are stored in the Tang's database directory and update script, which is activated by systemd, generates the cache in a cache directory. The server is ready to send an advertisment on demand.

3.2.4 Clevis client

Clevis has full support for Tang and provides a pluggable key management framework for automated decryption. It can handle automated unlocking of LUKS volumes. Clevis lets us encrypt some data with a simple command:

```
$ clevis encrypt PIN CONFIG < PLAINTEXT > CIPHERTEXT.jwe
```

In clevis terminology, a *PIN* is a plugin which implements automated decryption. We simply pass the name of the supported pin here. Besides Tang *PIN* clevis also supports a *PIN* for performing escrow using HTTP or an SSS *PIN* to provide a way to mix pins together to provide sophisticated unlocking policies by using an algorithm called Shamir Secret Sharing (SSS).

Secondly, *CONFIG* is a JSON object which will be passed directly to the *PIN*. It contains all the necessary configuration to perform encryption and setup automated decryption.

PIN: Tang Clevis has full support for Tang. Here is an example of how to use Clevis with Tang:

```
$ echo hi | clevis encrypt tang '{"url": "http://tang.local"}' > hi.jwe
The advertisement contains the following signing keys:

Apb39FO1vey9FyUe_fEd8lVDABs

Do you wish to trust these keys? [ynYN] y
$ clevis decrypt tang '{"url": "http://tang.local"}' < hi.jwe
hi
```

In this example, we encrypt the message „hi“ using the Tang pin. The only parameter needed in this case is the URL of the Tang server. During the encryption process, the Tang pin requests the key advertisement from the server and asks us to trust the keys. This works similarly to SSH.

Alternatively, we can manually load the advertisment using the adv parameter. This parameter takes either a string referencing the file where the advertisement is stored, or the JSON contents of the advertisment itself. When the advertisment is specified manually like this, Clevis presumes that the advertisement is trusted.

3.2.5 Binding LUKS volumes with clevis

Tang's main goal is to solve early boot decryption of the LUKS volumes and clevis has a solution for it. Clevis can be used to bind a LUKS volume using a pin so that it can be automatically unlocked.

How this works is rather simple. We generate a new, cryptographically strong key. This key is added to LUKS as an additional passphrase. We then encrypt this key using Clevis, and store the output JWE inside the LUKS header using LUKSMeta. Here is an example where we bind `/dev/vda2` using the Tang ping:

```
# clevis bind-luks /dev/sda1 tang '{"url": "http://tang.local"}'
The advertisement is signed with the following keys:
        kWwirxc5PhkFIH0yE28nc-EvjDY

Do you wish to trust the advertisement? [yN] y
Enter existing LUKS password:
```

Upon successful completion of this binding process, the disk can be unlocked using one of the unlockers described below.

Dracut The Dracut unlocker attempts to automatically unlock volumes during early boot. This permits the automated root volume encryption. To unlock our Fedora, rebuild initramfs after installing Clevis using:

```
# dracut -f
```

Upon reboot, we will be prompted to unlock the volume using a password. In the background, Clevis will attempt to unlock the volume automatically. If it succeeds, the password prompt will be cancelled and boot will continue. Install it to fedora using:

```
# dnf install clevis-dracut
```

UDisks2 After installation UDisks2 unlocker runs in a Fedora desktop session. There is no need to manually enable it; just install the Clevis UDisks2 unlocker and restart our desktop session.

```
# dnf install clevis-udisks2
```

The unlocker should be started automatically. This unlocker works almost exactly the same as the Dracut unlocker. If we insert a removable storage device that has been bound with Clevis, we will attempt to unlock it automatically in parallel with a desktop password prompt. If automatic unlocking succeeds, the password prompt will be dissmissed without user intervention.

Chapter 4

Software portability

Ideally, any software would be usable on any operating system, platform, and any processor architecture. Existence of term „porting“, derived from the Latin portāre which means „to carry“, proves that this ideal situation does not occurs that often, and the actual process of „carrying“ software to a system with a different environment is most probably required. Porting is process, which is not documented, required to adapt software designed for specific platform to another platform. Porting is also used to describe procces of converting computer games to become platform independent[31].

Software porting procces might be hard to distinguish with building software. The reason might be that in many cases, building software on the desired platform is enough, when software application does not work „out of the box“. This kind of behavior, an application that works immediately after or even without any special installation and without need for any configuration or modification, is ideal. It often happens when we copy application from one computer to another, not realizing that they have processors with same instructions set and same or very similar operating system therefore environment. But let us be realistic, it depends on many things, such as processor architecture to which was the application written (compiled), quality of design, how the application is meant to be deployed and of course application's source code.

Nowadays, the goal should be to develop software which is portable between preferred computer platforms (Linux, UNIX, Apple, Microsoft). If the software is considered as not portable, it does not have to mean that it is not possible, just that the time and resources spent porting already written software are almost comparable, or even significantly higher than writing software as a whole from scratch. Effort spent porting some software product to work on a desired platform must be little, such as copying already installed files to usb flash drive and run it on another computer. This kind of approach might most probably fail, due to not present dependencies of third party libraries on the destination computer. Despite dominance of the x86 architecture, there is usually a need to recompile software running, not only on different operating systems, to make sure we have all the dependencies present.

Number of significantly different central processor units (CPUs), and operating systems used on the desktop or the server is much smaller than in the past. However, on embeded devices market, there are still much more various architectures available including ARM[1] or MIPS[2].

[1] https://www.arm.com/products/processors
[2] https://www.mips.com/products/classic/

To simplify portability, even on distinguished processors with distant instruction sets, modern compilers translate source code to a machine-independent intermediate code. But still, in the embedded system market, where OpenWrt belongs to, porting remains a significant issue [12].

4.1 OpenWrt system

OpenWrt is perhaps the most widely known Linux distribution for embedded devices especially for wireless routers. It was originally developed in January 2004 for the Linksys WRT54G with buildroot from the uClibc project. Now it supports many more models of routers. OpenWrt is a registered trademark which is held by the Software in the Public Interest (SPI) in the name of the OpenWrt project.

Installing OpenWrt system means replacing our router's built-in firmware with the Linux system which provides a fully writable filesystem with package management. This means that we are not bound to applications provided by the vendor. A router (the embedded device) with this distribution can be used for anything that an embedded Linux system can be used for, from using its SSH Server for SSH Tunneling, to running lightweight server software (e.g. IRC server) on it. In fact, it allows us to customize the device through the use of packages to suit any application. [17]

4.1.1 OpenWrt and LEDE

The LEDE Project ("Linux Embedded Development Environment") is a Linux operating system emerged from the OpenWrt project. Its announcement was sent on 3th May 2016 by Jo-Philipp Wich to both the OpenWrt development list and the new LEDE development list[3]. It describes LEDE as „a reboot of the OpenWrt community" and as „a spin-off of the OpenWrt project" seeking to create an embedded-Linux development community „with a strong focus on transparency, collaboration and decentralisation"[4].

The rationale given for the reboot was that OpenWrt suffered from longstanding issues that could not be fixed from within—namely, regarding internal processes and policies. For instance, the announcement said, the number of developers is at an all-time low, but there is no process for on-boarding new developers and, it seems, no process for granting commit access to new developers.

At the moment, latest release of OpenWrt 15.05.1 (code-named „Chaos Calmer") released in March 2016. LEDE developers continued to work separately on their upstream release and they delivered LEDE „Reboot" with version 17.01.0 on February 22nd 2017.

The remerge proposal vote was passed by LEDE developers in June 2017[5]. After long and sometimes slowly moving discussions about the specifics of the re-merge, with multiple similar proposals but little subsequent action, formally announced on LEDE forum in January 2018[6]. OpenWrt and LEDE projects agreed upon their unification under the OpenWrt name. After merge OpenWrt upstream repository started to show signs of life.

Today (April 2018) the stable LEDE 17.01.4 „Reboot" release of OpenWrt released in October 2017 using Linux kernel version 4.4.92 runs on many routers [18].

[3] https://lwn.net/Articles/686180/
[4] https://www.phoronix.com/scan.php?page=news_item&px=OpenWRT-Forked-As-LEDE
[5] http://lists.infradead.org/pipermail/lede-adm/2017-June/000552.html
[6] https://forum.lede-project.org/t/announcing-the-openwrt-lede-merge/10217

4.1.2 Why use OpenWrt

Custom router firmware may be more stable than our hardware's default firmware from the vendor. Not even that, but probably more secure with regular security updates. Besides OpenWrt, there is another open source Linux based firmware available such as Tomato or DD-WRT.

In the past OpenWrt has supported only CLI (command line interface) configuration, therefore, it was best match for software developers, network admins, or advanced users. A user not acquainted with Linux or even not comfortable with CLI may find the OpenWrt platform hard to use and may turn to other available solutions.

The Tomato firmware is the best match for unexperienced users providing rich GUI and many other features, specifically live „visual" traffic monitoring, allowing easy visibility on inbound/outbound traffic in real-time. Big disadvantage of the Tomato is that the list of supported devices is quite poor.

DD-WRT on the other hand, is compatible with more routers than any other third party firmware. Compared to Tomato, DD-WRT is reported to have more bugs and less intuitive GUI. The DD-WRT was known as the most feature rich firmware until OpenWrt came along.

OpenWrt turns our router in a fully capable GNU/Linux computer, not just a network „magic" box. Recently, the OpenWrt platform has come a long way in making itself more accessible to all user levels. It has an Luci[7] Web UI(user interface) now. Users that are less experienced with Linux can easily set up their network using Luci. OpenWrt is capable of running lightweight services like an IRC bouncer or samba/ftp file sharing (some routers have USB ports able to power HDD) or even run software build on our own[25].

The goal of this thesis would be to port a lightweight Tang daemon described in the section 3.2 Tang to OpenWrt system. Tang server will help us unlock our encrypted volumes while on safe home or office network without a need for extra PC running it but having it on our tiny OpenWrt „server". With Tang, we do not have to care about typing passphrases over and over to unlock LUKS drives in a safe environment.

4.2 OpenWrt's toolchain

Embeded devices are not meant for building on them because they do not have enough memory nor computation resources as ordinary personal computers do(see Table 4.1). Building on such device would be time consuming and may result in overheating, which could cause the hardware to fail. For this particular reason, package building is done with cross-compiler. Compilation is done by set of tools called toolchain and it consists of:

- compiler
- linker
- a C standard library

For porting to „target" system (OpenWrt) this toolchain has to be generated on „host" system. The toolchain can be achieved in many different ways. The easiest way is undoubtedly to find a .rpm (.deb or any distribution specific available) package and have it installed on our „host" system. If a binary package with desired toolchain is not available

[7] https://github.com/openwrt/luci

for our system or is not available at all, there might be a need to compile a custom toolchain from scratch.[8]

In case of OpenWrt, we have an available set of Makefiles and patches called buildroot which is capable of generating toolchain.

4.2.1 Compiler

In a nutshell, compilers translate the high level programming language source code into lower level language such as machine understandable assembler instructions. To do so, the compiler is performing many operations, starting with preprocessing.

Preprocessing supports macro substitution and conditional compilation. Typically, the preprocessing phase occurs before syntactic or semantic analysis. However, some languages such as Scheme[9] support macro substitutions based on syntactic forms.

Lexical analysis, also known as lexing or tokenization, breaks the source code text into a sequence of small pieces called lexical tokens. Lexical analyzer does the scanning (cutting the input text into tokens and assign them a category) and the evaluating (converting tokens into a processed value). Common token categories may include:

- keywords (bound to programming language)
- identifiers (can not be keyword)
- separators
- operators
- literals
- comments

The set of token categories varies in different programming languages. The lexeme syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it.

Syntax analysis typically builds a parse tree structure according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.

Semantic analysis checks the parse tree and does type checking, object binding and definite assignment. The output of semantic analysis is the symbol table or it may result in rejecting incorrect programs or issuing warnings.

Code optimization, such as dead code elimination, inline expansion, constant propagation, loop transformation and even automatic parallelization, is done after analysis and it is independent on the CPU architecture.

Code generation is CPU dependent and does the transformation of intermediate language into the target machine language. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and generating machine instructions along with their associated addressing modes.

Cross-compiler is a programming tool capable of creating an executable file that is supposed to run on a „target“ architecture, in a similar or completely different environment, while working on a different „host“ architecture. It can also create object files used by linker.

[8] tools like crosstool-ng (https://github.com/crosstool-ng/crosstool-ng) may help

[9] https://www.scheme.com/tspl4/

The reason for using cross-compiling might be to separate the build environment from the target environment as well. OpenWrt toolchain uses gcc compiler, and it is one of the most important parts of toolchain[28].

4.2.2 Linker

With a linker, we are able to link compiler's object files into a single executable, library or object file. A linker can do static linking and dynamic linking.

Static linking is the result of the linker copying all library routines used in the program into the executable image. This may require more disk space and memory than dynamic linking, but is more portable, since it does not require the presence of the library on the system where it runs. Static linking also prevents „DLL Hell", since each program includes exactly the versions of library routines that it requires, with no conflict with other programs. In addition, a program using just a few routines from a library does not require the entire library to be installed.

Dynamic linking is the process of postponing the resolving of some undefined symbols until a program is run. That means that the executable code still contains undefined symbols, plus a list of objects or libraries that will provide definitions for these. Loading the program will load these objects/libraries as well, and perform a final linking. Dynamic linking needs no linker.

The advantage of dynamic linking is that often-used libraries (for example the standard system libraries) need to be stored in only one location, not duplicated in every single binary. If a bug in a library function is corrected by replacing the library, all programs using it dynamically will benefit from the correction after restarting them. Programs that included this function by static linking would have to be re-linked first.

The disadvantage known on the Windows platform as „DLL Hell", an incompatible updated library will break executables that depended on the behavior of the previous version of the library if the newer version is not properly backward compatible[28].

4.2.3 C standard library

Most common C standard libraries are: GNU Libc, uClibc musl-libc, or dietlibc. They provide macros, type definitions and functions for tasks such as input/output processing (<stdio.h>), memory management (<stdlib.h), string handling (<string.h>), mathematical computations (<math.h>) and many more[10]. The OpenWrt's cross-compilation toolchain uses musl-libc [29].

4.3 OpenWrt's buildroot

OpenWrt's buildroot is a build system capable of generating toolchain, and also a root filesystem (also called sysroot), an environment tightly bound to the target. The build system can be configured for any device that is supported by OpenWrt.

The root filesystem in general is a mere copy of the file system of target's platform. In many cases, just having the folders /usr and /lib would be sufficient, therefore we do not need to copy nor create the entire target file system on our host.

[10] https://en.wikipedia.org/wiki/C_standard_library#Header_files

It is a good idea to store all these things, the toolchain and the root filesystem in a single place. With using OpenWrt's buildroot we will have this covered. Be tidy and pedantic, because cross-compiling can easily become a painful mess![3]

4.3.1 Buildroot prerequisites

Let us demonstrate what are minimum requirements of space and size of RAM (Random Access Memory) for building packages for Openwrt using its buildroot. For generating an installable OpenWrt firmware image file with a size of e.g. 8MB, we will need at least:

- ca. 200 MB for OpenWrt build system
- ca. 300 MB for OpenWrt build system with OpenWrt Feeds
- ca. 2.1 GB for source packages downloaded during build from the Feeds
- ca. 3-4 GB to build (i.e. cross-compile) OpenWrt and generate the firmware file
- ca. 1-4 GB of RAM to build Openwrt.(build x86's img need 4GB RAM)

These are specifications of embedded device TL-WR842Nv3, a regular wireless router manufactured by TP-LINK, which was used to test all packages related to Tang server porting effort:

Model	TL-WR842N(EU)
Version	v3
Architecture:	MIPS 24Kc
Manufacturer:	Qualcomm Atheros
Bootloader:	U-Boot
System-On-Chip:	Qualcomm Atheros QCA9531-BL3A
CPU Speed:	650 MHz
Flash chip:	Winbond 25Q128CS16
Flash size:	16 MiB
RAM chip:	Zentel A3R12E40CBF-8E
RAM size:	64 MiB
Wireless:	Qualcomm Atheros QCA9531
Antenae(s):	2 non-removable
Ethernet:	4 LAN, 1 WAN 10/100
USB:	1 x 2.0
Serial:	?

Table 4.1: TL-WR842Nv3 specifications

Comparison of available storage space on wireless router to the actual sum of space required only for buildroot to work correctly, which is about 6.4 GB, should demonstrate why the building for embedded devices is done with cross-compiling. Another reasons, not to just compare internal storage which might be extended[11] are device's minimalistic RAM size and lower computation capability of the CPU.

[11] https://wiki.openwrt.org/doc/howto/extroot

4.4 Preparing the host environment

To start with cross-compilation on the host system, we need to set up an environment for it. As the OpenWrt buildroot is a set of scripts, it has runtime dependencies. We need to install these dependencies first. To install buildroot dependencies on Fedora 27 system run:

```
# dnf install binutils bzip2 gcc gcc-c++ \
gawk gettext git-core flex ncurses-devel ncurses-compat-libs \
zlib-devel zlib-static make patch perl-ExtUtils-MakeMaker \
perl-Thread-Queue  glibc glibc-devel glibc-static quilt sed \
sdcc intltool sharutils bison wget unzip openssl-devel
```

Some feeds (the OpenWrt packages) might be not available over git but only via other versioning tools like svn (subversion) or mercurial. If we want to obtain their source-code, we need to install svn and mercurial as well:

```
# dnf install subversion mercurial
```

Please note that these commands have to be run by a user with root privileges.

4.4.1 Getting buildroot

OpenWrt system has available SDK(Software development kit) buildroots for every released version of OpenWrt system. It is good to consider using OpenWrt's SDK in order to build the software application to only one specific release of the target system. For example when we are using a „stable“ release of OpenWrt 15.05.1 with codename „Chaos Calmer“ on TL-WR842N(EU) device, we should probably end up in „Supplementary Files“ section of the OpenWrt archives [12] looking for the SDK. While porting an upstream projets (such as Tang) for latest target release a bleeding edge buildroot would be the best solution.

If we have a host platform which aims to be on the bleeding edge of open-source technologies, such as OS Fedora, we will probably encounter issues with dependencies. These issues are appearing because an older version of the dependencies are required for this old SDK to work and might not be available for our host platform anymore. In this case, we would have to install or even build an older version of required packages.

We will work with bleeding edge (trunk version) buildroot and build for the latest LEDE 17.01.4 release. To clone upstream buildroot run command:

```
$ git clone https://github.com/openwrt/openwrt.git
```

4.5 Working with buildroot

Working with Openwrt's buildroot requires some basic knowledge of the git version control system[13] and processes of upstream project development using GitHub[14]. A short description of how to fork and set up repository can be found in appendix D Setting up the repository. All work related to this thesis has been done using GitHub account Tiboris[15] therefore following command to get our fork of buildroot would be:

```
$ git clone https://github.com/Tiboris/openwrt.git ~/buildroot-openwrt
```

[12] https://archive.openwrt.org/chaos_calmer/15.05.1/ar71xx/generic/
[13] https://git-scm.com/docs/gittutorial
[14] https://guides.github.com/introduction/flow/
[15] https://github.com/Tiboris/

After we have the environment ready to be worked on (all required dependencies installed and fork of our buildroot downloaded on host system), let us just follow these simple 4 rules to not break our environment in any way.

1. Do everything in buildroot as non-root user!
2. Issue all OpenWrt build system commands in the <buildroot> directory, e.g. ∼/buildroot-openwrt/
3. Do not build in a directory that has spaces in its full path.
4. Change ownership of the directory where we downloaded the OpenWrt to other than root user

Having these in mind will definitely make our work easier.

4.5.1 Setting feeds

In OpenWrt, a „feed“ is a collection of packages which share a common location. Feeds may reside on a remote server, in a version control system, on the local filesystem, or in any other location addressable by a single name (path/URL) over a protocol with a supported feed method. It is the most important step to do before starting cross-compilation. As the listing 4.1 Content of feeds.conf.default shows, a list of usable feeds is configured from the feeds.conf file or feeds.conf.default when feeds.conf does not exist.

```
src-git packages https://git.openwrt.org/feed/packages.git
src-git luci https://git.openwrt.org/project/luci.git
src-git routing https://git.openwrt.org/feed/routing.git
src-git telephony https://git.openwrt.org/feed/telephony.git
#src-git video https://github.com/openwrt/video.git
#src-git targets https://github.com/openwrt/targets.git
#src-git management https://github.com/openwrt-management/packages.git
#src-git oldpackages http://git.openwrt.org/packages.git
#src-link custom /usr/src/openwrt/custom-feed
```

Listing 4.1: Content of feeds.conf.default

The new custom OpenWrt packages are located in packages feed (see the first line of the 4.1 Content of feeds.conf.default listing). To work with our own fork of the packages, we can simply change the line to point to our fork repository. Feeds can point to a special branch of our choice:

```
src-git packages https://github.com/Tiboris/packages-OpenWrt.git;new_packages
```

or commit hash in repository:

```
src-git packages https://github.com/Tiboris/packages-OpenWrt.git^dbdfc99
```

Changing this feed to our custom feed will reduce effort spent on getting all new dependencies to the buildroot's feeds/packages directory.

The feeds are „managed“ with the script available in builroot's scripts directory, *feeds*. To download the feeds, run this script with command *update* and option *-a*. Remember to invoke it from the buildroot directory(∼/buildroot-openwrt).

```
$ ./scripts/feeds update -a
```

To make any feed available for the build, we shall „install“ it using the same feeds script:

```
$ ./scripts/feeds install <PACKAGENAME>
```

or we can use option *-a* instead of the *<PACKAGENAME>* and make all of the feeds available for the build.

If for some unknown reason the issued update of packages does not seems to show all the updates, it might be helpful to clean the buildroot's *tmp/* directory[20].

```
$ rm -rf tmp/
```

4.5.2 The menuconfig

For OpenWrt, it has been the intention from the beginning, with the development of menuconfig, to create a simple, yet powerful, environment for the configuration of individual builds. Working with menuconfig is very intuitive and the tool is more or less self-explanatory. Even the most specialized configuration requirements can be met by using it. Depending on the particular target platform, package requirements and kernel module needs, the standard configuration process will include modifying:

- Target system
- Package selection
- Build system settings
- Kernel modules

Start the menuconfig interface shown on Figure 4.1 menuconfig by issuing the following command:

```
$ make menuconfig
```

The make menuconfig will collect package info first. Afterwards, the following window will appear in our terminal and there we can start configuring the image using menuconfig:

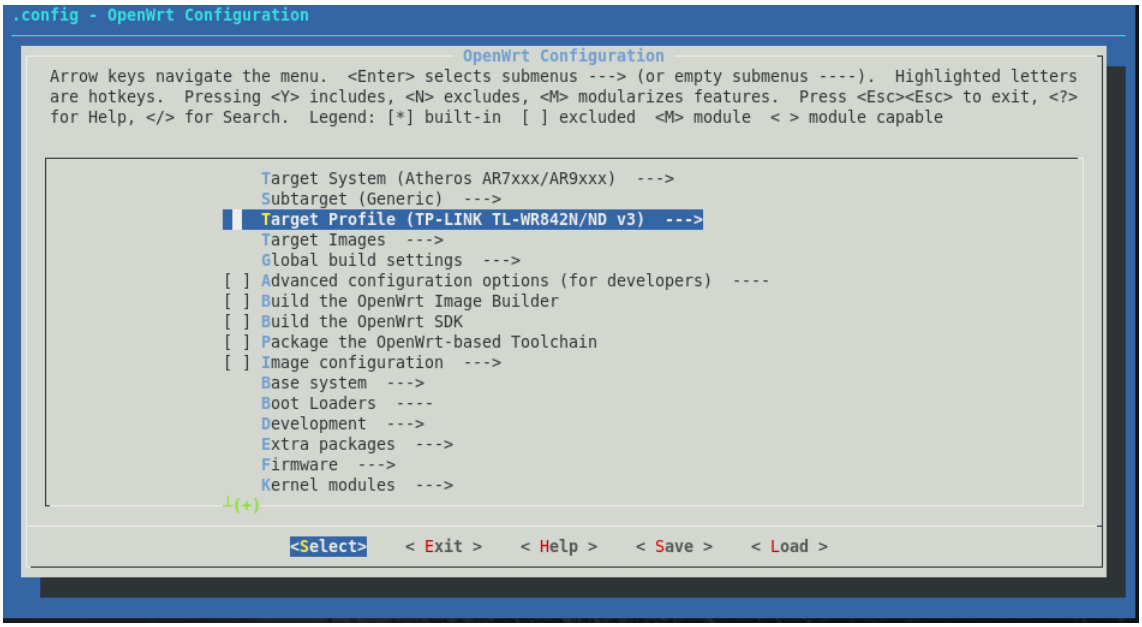


Figure 4.1: menuconfig

For configuring the image, we have three options: y, m, n which are represented as follows:

- pressing y sets the <*> built-in label
  This package will be compiled and included in the firmware image file.
- pressing m sets the <M> package label
  This package will be compiled, but not included in the firmware image file. (E.g. to be installed with opkg after flashing the firmware image file to the device.)
- pressing n sets the < > excluded label
  The source code will not be processed.

When we save our configuration, the file .config will be created in the buildroot directory according to desired configuration.

Target system is selected from the extensive list of supported platforms, with the numerous target profiles – ranging from specific devices to generic profiles, all depending on the particular device at hand. In our case, we should browse the „Target Profile“ selection and find our targeted device (TP-LINK TL-WR842N/ND v3).

Package selection has the option of either 'selecting all package', which might be unpractical in certain situations, or relying on the default set of packages will be adequate or make an individual selection. It is here worth mentioning that some package combinations might break the build process, so it can take some experimentation before the expected result is reached. Added to this, the OpenWrt developers are themselves only maintaining a smaller set of packages – which include all default packages – but, the feeds-script makes it very simple to handle a locally maintained set of packages and integrate them in the build-process.

The final step before the process of compiling the intended image would be to exit the menuconfig tool – this also includes the option to save a specific configuration or load an already existing, and pre-configured, version. Exit the UI, and choose to save the settings[27].

4.5.3 Building single Packages

When developing or packaging software for OpenWrt, it is convenient to be able to build only the package in question (e.g. with package cups):

```
$ make package/cups/compile V=s
```

For a rebuild run:

```
$ make package/cups/{clean,compile,install} V=s
```

It doesn't matter what feed the package is located in, this same syntax works for any installed package.

If for some reason the build fails, the easiest way to spot the error is to do:

```
$ make V=s 2>&1 | tee build.log | grep -i error
```

Complete guide can be found on OpenWrt Wiki[16].

[16] https://wiki.openwrt.org/doc/howto/build

Chapter 5

Porting the dependencies

Programs don't run in a vacuum but they interface with the outside world. The view of this outside world differs from environment to environment. Things like hostnames, system resources, and local conventions could be different.

When we start porting a code to a specific target platform, in our case OpenWrt, it is likely that we will face the first problem: satisfying missing dependencies. This problem is easy to solve in principle, but can easily mess things up to a level we wouldn't imagine.

If the code depends on some library that is NOT in the root filesystem, there's no way out but to find them somewhere, somehow. Dependencies can be satisfied in two ways: with static libraries or with shared libraries. If we are lucky, we could find a binary package providing what we need (i.e. the library files and the header files), but most often we will have to cross-compile the source code on our own. Either ways, we end up with one or more binary files and a bunch of header files. Where to put them? Well, that depends. There are a few different situations that can happen, but basically everything reduces to having dependencies in buildroot's root filesystem or in a different folder.

Having dependencies in a different folder could be an interesting solution to keep the libraries that we cross-compiled on our own separated from the other libraries (for example, the system libraries). We can do that if we want but if we do, we must remember to provide to compiler and linker programs with the paths where header files and binary files can be found. With static libraries, this information are only needed at compile and linking time, but if we are using shared libraries, this won't suffice. We must also specify where these libraries can be found at run time.

If we are satisfying the dependencies with shared libraries (.so files) having dependencies in root filesystem is probably the most common solution (and maybe, the best solution). Remember that when everything will be up and running, these libraries must be installed somewhere in the file system of the target platform. So there is a natural answer to the question above: install them in the target's root filesystem, for example in /usr/lib (the binary shared files) and /usr/include (the header files) or in any other path that allow the loader to find those libraries when the program executes. Do not forget to install them in the file system of the actual target machine, in the same places, in order to make everything work as expected. Please note that static libraries ('.a' files) does not need to be installed in the target file system since their code is embedded in the executable file when we cross-compile a program. In case of OpenWrt we will use this approach[3].

5.1 Find the depencies

To port Tang to OpenWrt system we have have all its dependencies available and installed in buildroot. First thing we should do is some digging and find out if dependencies for our software, we are about to port, are available for target's platform. All packages available for OpenWrt can be found in its github repositories. Let us remember Tang's dependencies first:

- http-parser
- systemd's socket activation
- José
  - jansson
  - openssl
  - zlib

After some digging we will find out that the OpenWrt system has already packages openssl, zlib, http-parser, and jansson. Packages openssl and zlib are in versions already sufficient for Tang to get things work.

Package http-parser was little bit tricky. At first try there is a chance that we will not find this package in OpenWrt's repository. The reason might be that we will try to find a exactly „http-parser“ string in OpenWrt's GitHub repository openwrt/openwrt only. First of all, we must not forget that feeds of many packages are living in repository packages nested under openwrt organization. Secondly after searching repository openwrt/packages for http-parser we will find out that it is named as libhttp-parser and in version 2.2.3. This, let us say naming convention, is really common in Linux and could be predictable as the http-parser is in fact only a library. One way or another compared to fedora packages it needs to be updated to latest version.

Jansson package was only available in version 2.7 which is too outdated for Tang's dependency José because it require at least jansson version 2.10. We shall not forget that there is a need for updating jansson package for OpenWrt platform as well so we will do updates first.

Packages José, Tang and systemd are not listed in OpenWrt's packages. Porting of the systemd would be huge effort but tang's requirements are minimal and we should be able to work with xinetd's socket activation. Finally, as José and Tang are not listed in packages they will require to add them into package feeds to port Tang itself.

We will use fork of openwrt/packages repository placed outside of the buildroot package:

```
$ git clone git@github.com:Tiboris/packages-OpenWrt.git
```

The following packages were built using OpenWrt buildroot with latest upstream commit hash `030a23001b74ede5fa2e6070a8fb04f3feccfbbd`. The OpenWrt buildroot has to have a packages feeds set to point to repository with available OpenWrt packages. We used repository with latest upstream commit hash `580053888235713dd95b96b37169926bffedce0b`.

5.2 Update outdated packages

Finding that some of dependencies are already available on our desired target platform will definitelly make us satisfied. We would agree that starting with something already built for OpenWrt is the best thing to do when we are approaching unknown platform. In following subsections we will use all things we know from section 4.5 Working with buildroot. Let us start with missing update of José's dependency, package jansson.

5.2.1 Update jansson

Jansson is a C library for encoding, decoding and manipulating JSON data. The latest release of the jansson is v2.11 released on 11th of February 2018[13].

However at the begining of the Tang porting efort started the latest release was v2.10. To update mentioned OpenWrt available jansson package version v2.7 to 2.10 change to package's fork directory and create a new branch for changes:

```
$ cd ~/packages-OpenWrt
$ git checkout -b jansson-update
```

In order to tell the OpenWrt buildroot how to build a program we need to create a special Makefile in the appropriate directory. The appendix E OpenWrt package's Makefile ilustrate its content. We shall now find the jansson package Makefile in the repository using for example:

```
$ git grep jansson | grep PKG_NAME
libs/jansson/Makefile:PKG_NAME:=jansson
```

The *PKG_NAME* variable is one that identifies package for OpenWrt buildroot[19]. List of all available variables can be found on the OpenWrt wiki page[1]. We can assume that the jansson library related files are located in *libs/jansson* directory of the packages repository. Now we can now update Makefile.

We will open it with our preferred editor and find variable *PKG_VERSION*. Edit the old version number (in our case 2.7) to new version number (2.10). This edit will result into changing the *PKG_SOURCE* variable in Makefile. Variables *PKG_SOURCE* and *PKG_SOURCE_URL* are used to identify location of the archive with sources for specified version from where the sources would be downloaded. After the downaload the OpenWrt buildroot checks the file integrity. The *PKG_HASH* and *PKG_MD5SUM* variables serve this purpose. As the new version of archive with sources will be downloaded we need to change *PKG_HASH*/*PKG_MD5SUM* variable as well. For some reason upstream developers required the use of *PKG_HASH* variable and the bz2 archive for the jansson package. In general it would be sufficient to change only version and the appropriate *PKG_HASH*/*PKG_MD5SUM* variable. We changed the filename extension for *PKG_SOURCE* variable from „.tar.gz“ to „.tar.bz2“. Now download the archive containing sources and get the archive hash using sha256sum:

```
$ wget http://www.digip.org/jansson/releases/jansson-2.10.tar.bz2 -P /tmp
$ sha256sum /tmp/jansson-2.10.tar.bz2 | cut -d " " -f1
241125a55f739cd713808c4e0089986b8c3da746da8b384952912ad659fa2f5a
```

Last but not least commit the changes and push the changes into our fork.

```
$ git commit -a
$ git push --set-upstream origin jansson-update
```

[1] https://wiki.openwrt.org/doc/devel/packages

Before submitting the pull request we should try to build updated package first. Following step is not neccessary because update of jansson package has been already merged upstream. But to do so, let us remember that we have special branch set up for buildroot feeds. In general to test updated or new packages we will use this branch to merge our newly created branch into it using:

```
$ git checkout new_packages
$ git merge jansson-update
```

After successfull merge we have updated jansson package available in our custom feeds. Trigger update with feeds script and make jansson available in menuconfig with:

```
$ ./scripts/feeds update packages
$ ./scripts/feeds install jansson
```

To finally build package we shall run the command:

```
$ make package/jansson/{clean,compile} V=s
```

After successfull build the most important part would be to contribute changes to the upstream. Update of jansson is already done through merged pull-request[2]. With a knowledge of buildroot and the contribution guidelines effort spent on such updates may be quite minimal.

5.2.2 Update http-parser

Tang uses this parser library for both parsing HTTP requests and HTTP responses. Sources can be found on its GitHub page[3].

The lastest release available at the time was version v2.7.1 untill 9th February's 2018 release of v2.8.0 which update will be demonstrated below. Fedora is still using version v2.7.1 with Tang server so compared to last available version on OpenWrt which was v2.3.0 an update is needed. Hoping for the best we first thy to update the libhttp-parser to version v2.7.1 to match Fedora version similar way as with jansson. Only updating version of the package may suffice but the case of libhttp-parser as dependency was special as you will notice in subsection 6.2.1 Obstacles of delivering Tang. To upgrade this package we will do same as with jansson package. First make sure that we are still in packages repository and create branch for the change:

```
$ git checkout master
$ git checkout -b libhttp-parser-update
```

Let us locate the http-parser Makefile:

```
$ git grep http-parser | grep PKG_NAME
libs/libhttp-parser/Makefile:PKG_NAME:=libhttp-parser
```

Find out the source url to download the archive containing and get the archive hash using sha256sum:

```
$ wget https://github.com/nodejs/http-parser/archive/v2.8.0.tar.xz -P /tmp
$ sha256sum /tmp/v2.8.0.tar.gz | cut -d " " -f1
83acea397da4cdb9192c27abbd53a9eb8e5a9e1bcea2873b499f7ccc0d68f518
```

Please note the file extension in old makefile and donwload same filetype for upgrade.

[2] https://github.com/openwrt/packages/pull/4289
[3] https://github.com/nodejs/http-parser

Before commiting the changes we also realized that owner of the repository changed from „joyent“ to „nodejs“ so we addressed these changes as well by editing propper sections. We shall now edit correspondent variables, commit the changes and push them into our fork.

```
$ git commit -a
$ git push --set-upstream origin libhttp-parser-update
```

Again following step to merge the libhttp-parser-update branch with feeds branch is not neccessary because the submitted pull-request containing these changes has been already merged to the upstream:

```
$ git checkout new_packages
$ git merge libhttp-parser-update
```

After package is available in our feeds trigger update with feeds script and make new version of libhttp-parser package available in menuconfig:

```
$ ./scripts/feeds update packages
$ ./scripts/feeds install libhttp-parser
```

Finally build an updated libhttp-parser running:

```
$ make package/libhttp-parser/{clean,compile} V=s
```

The update of the libhttp-parser can be found upstream in merged pull-request[4]. Unfortunatelly as we will see in subsection 6.2.1 Obstacles of delivering Tang, the successfull build of the updated package may not be enough. Especially when built package is also a dependency for other packages.

5.3 New package José

After updating of jansson and libhttp-parser we are kind of familiar with the OpenWrt's Makefiles. Unfortunatelly José package is not packaged for OpenWrt. Now comes the time to write makefile on our own.

José is a C-language implementation of the Javascript Object Signing and Encryption standards. Specifically, José aims towards implementing the following standards:

- RFC 7520 - Examples of JSON Object Signing and Encryption (JOSE) [16]
- RFC 7515 - JSON Web Signature (JWS) [8]
- RFC 7516 - JSON Web Encryption (JWE) [10]
- RFC 7517 - JSON Web Key (JWK) [7]
- RFC 7518 - JSON Web Algorithms (JWA) [6]
- RFC 7519 - JSON Web Token (JWT) [9]
- RFC 7638 - JSON Web Key (JWK) Thumbprint [11]

JOSE (Javascript Object Signing and Encryption) is a framework intended to provide a method to securely transfer claims (such as authorization information) between parties. Tang uses JWKs in comunication between client and server. Both POST request and reply bodies are JWK objects[14].

[4] https://github.com/openwrt/packages/pull/5446

5.3.1 Create José

First, let us create feature branch and directory for new package:

```
$ git checkout master
$ git checkout -b libhttp-parser-update
$ mkdir -p utils/jose
```

At first it does not matter whether new Makefile will be placed whether libs or utils for the build purposes. We can simply change it as upstream developers would require.

To start with such a work it is good to have some kind of the template. For the José we used the jansson's Makefile as a template. Place this „template" Makefile into utils/jose directory and start with editing having the OpenWrt wiki page[5] opened on the side.

Let us go through it from the top to the bottom. This is first non comment line in the file:

```
include $(TOPDIR)/rules.mk
```

Without this include our Makefile would not work so we will leave it as it is. The next are the package name, version and release variables. This have to be the first thing to edit:

```
PKG_NAME:=jose
PKG_VERSION:=10
PKG_RELEASE:=1
```

We will skip the licence variable for José because of the its upstream.

As we defined the version of the package which we desire we should visit the project José pages and browse for the release archive. José's upstream releases lives on GitHub[6]. Visit the site and copy link location of the tar.bz2 archive for José release 10. Now download the archive and as we did when updating packages run sha256sum:

```
$ wget -P /tmp \
https://github.com/latchset/jose/releases/download/v10/jose-10.tar.bz2
$ sha256sum /tmp/jose-10.tar.bz2 | cut -d " " -f1
5c9cdcfb535c4d9f781393d7530521c72b1dd81caa9934cab6dd752cc7efcd72
```

This manual step is reflected in the Makefile as shown:

```
PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2

PKG_SOURCE_URL:=\
https://github.com/latchset/$(PKG_NAME)/releases/download/v$(PKG_VERSION)/

PKG_HASH:=\
5c9cdcfb535c4d9f781393d7530521c72b1dd81caa9934cab6dd752cc7efcd72
```

The *PKG_SOURCE* variable now contain the value of the source archive name. The *PKG_SOURCE_URL* provides the wshole path to archive stored on GitHub server and the *PKG_HASH* is used to verify the file integrity.

```
PKG_INSTALL:=1
PKG_BUILD_PARALLEL:=1

PKG_FIXUP:=autoreconf
include $(INCLUDE_DIR)/package.mk
```

[5] https://wiki.openwrt.org/doc/devel/packages
[6] https://github.com/latchset/jose/releases

Setting *PKG_INSTALL* to „1“ will call the package's original „make install“ to the directory defined with *PKG_INSTALL_DIR* variable. The *PKG_FIXUP* performs the important `autoreconf -f -i` for project using autotools. And again without including makefile package.mk the buildroot would not now how to continue build. Upstream may require to divide it into two packages the library and the tool. It is a good practice to do with every similar package:

```
define Package/libjose
  SECTION:=libs
  TITLE:=Provides a full crypto stack...
  DEPENDS:=+zlib +jansson +libopenssl
  URL:=https://github.com/latchset/jose
  MAINTAINER:=Tibor Dudlak <tibor.dudlak@gmail.com>
endef

define Package/$jose
  SECTION:=utils
  TITLE:=Provides a full crypto stack...
  DEPENDS:=+libjose +zlib +jansson +libopenssl
  URL:=https://github.com/latchset/jose
  MAINTAINER:=Tibor Dudlak <tibor.dudlak\@gmail.com>
endef
```

To add a nice description for both packages we should add some defines to Makefile with propper text:

```
define Package/jose/description
    ... description text ...
endef

define Package/libjose/description
    ... description text ...
endef
```

The *Build/Configure* section can be skipped if the source doesn't use configure or has a normal config script, otherwise we can put our own commands here or use *$(call Build/Configure/Default,)* to pass in additional arguments after the comma for a standard configure script.

```
define Build/Configure
    $(call Build/Configure/Default)
endef
```

Every library package should have section *Build/InstallDev*. This section is important for linker and buildsystem especially to work correctly when library would be used as dependency (static libs, header files) for other tools or packages. This section has no use on the target device.

```
define Build/InstallDev
    $(INSTALL_DIR)  $(1)/usr/lib
    $(INSTALL_DIR)  $(1)/usr/include
    $(INSTALL_DIR)  $(1)/usr/include/$(PKG_NAME)
    $(INSTALL_DIR)  $(1)/usr/lib/pkgconfig
    $(CP)   $(PKG_INSTALL_DIR)/usr/lib/lib$(PKG_NAME).so*   $(1)/usr/lib
    $(CP)   $(PKG_INSTALL_DIR)/usr/include/$(PKG_NAME)/*.h  \
            $(1)/usr/include/$(PKG_NAME)
    $(CP)   $(PKG_BUILD_DIR)/*.pc   $(1)/usr/lib/pkgconfig
endef
```

A set of commands to copy files into the ipkg which is represented by the *$(1)* directory. As source we can use relative paths which will install from the unpacked and compiled source, or *$(PKG_INSTALL_DIR)* which is where the files in the Build/Install (not used here) step end up.

```
define Package/libjose/install
    $(INSTALL_DIR)  $(1)/usr/lib
    $(CP)   $(PKG_INSTALL_DIR)/usr/lib/lib$(PKG_NAME).so*   $(1)/usr/lib/
endef

define Package/jose/install
    $(INSTALL_DIR)  $(1)/usr/bin
    $(INSTALL_BIN)  $(PKG_INSTALL_DIR)/usr/bin/$(PKG_NAME)  $(1)/usr/bin/
endef
```

At the bottom of the file is where the real magic happens, *BuildPackage* is a macro setup by the earlier include statements. BuildPackage only takes one argument directly – the name of the package to be built. All other information is taken from the define blocks.

```
$(eval $(call BuildPackage,libjose))
$(eval $(call BuildPackage,jose))
```

At this point our first Makefile for José is ready. Let us merge this changes to feeds and try to build our freshly created package.

```
$ git commit -a
$ git push --set-upstream origin add-jose
$ git checkout new_packages
$ git merge add-jose
```

In buildroot run:

```
$ ./scripts/feeds update packages
$ ./scripts/feeds install jose
$ make menuconfig
$ make package/jose/{clean,compile}
```

Note that feeds script with an install option will install also missing dependecies of José to be available in buildroot. The new packages will be available in the *Extra packages* section of the menuconfig as you can see on Figure 5.1 Extra packages. Do not worry if the José's

Figure 5.1: Extra Packages

build takes time, it has openssl library as dependency and to compile this one it takes some time. The pull request for adding the Jose package for OpenWrt can be found in its repository[7].

[7] https://github.com/openwrt/packages/pull/4334

Chapter 6

Porting Tang

After successfull cross-compilation of jose we have all the dependencies „ready“ and packaged except the systemd. systemd is only one of the many implementations (inetd, launchd, ucspi-tcp, xinetd) of a super-server providing socket activation.

6.1 Socket activation

Socket activation is a technology provided by a super-server (also called a service dispatcher daemon). A super-server starts other servers when needed as well, normally with access to them checked by a TCP wrapper. It uses very few resources when in idle state.

A service designed for the socket activation would behave as bare CLI application with input read from stdin (standard input) and output written to stdout (standard output). Tang is exactly this kind of an application and because of that we need to configure socket activation[**?**].

6.1.1 xinetd

xinetd listens for incoming requests over a network and launches the appropriate service for that request. Requests are made using port numbers as identifiers and xinetd usually launches another daemon to handle the request. This is reflected on Figure 6.1 xinetd socket activation below. xinetd features access control mechanisms such as TCP Wrapper

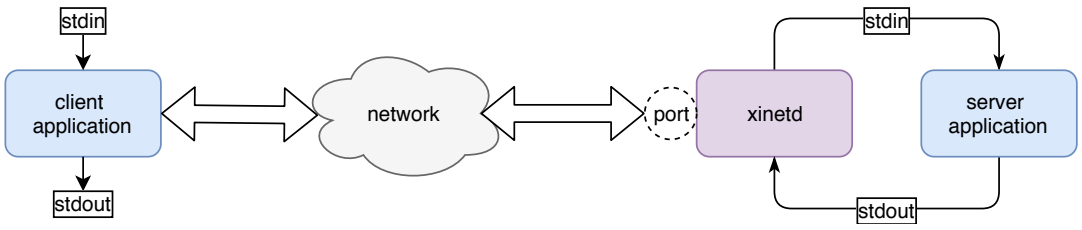


Figure 6.1: xinetd socket activation

ACLs (access control lists), extensive logging capabilities, and the ability to make services available based on time. It can place limits on the number of servers that the system can spawn. xinetd is listening on behalf of the services. Whenever a connection would come in an instance of the respective service will be spawned with using stdin and stdout of the service application[2].

6.2 Package the Tang

Similarly to José we need to create a new package for OpenWrt. Let us create a branch and directory for the Tang:

```
$ git checkout master
$ git chechout -b add-tang
$ mkdir -p utils/tang/
```

The Tang project is owned by same owner on GitHub as José. We should visit the project releases page[1] and get the Tang version v6. Then add following lines to the Makefile similarly as with José's Makefile:

```
include $(TOPDIR)/rules.mk

PKG_NAME:=tang
PKG_VERSION:=6
PKG_RELEASE:=1

PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2
PKG_SOURCE_URL:=\
https://github.com/latchset/$(PKG_NAME)/releases/download/v$(PKG_VERSION)/

PKG_HASH:=1df78b48a52d2ca05656555cfe52bd4427c884f5a54a2c5e37a7b39da9e155e3


PKG_INSTALL:=1
PKG_BUILD_PARALLEL:=1

PKG_FIXUP:=autoreconf

include $(INCLUDE_DIR)/package.mk
```

Do not forget to add the package description which should have section dependencies filled. The libhttp-parser dependency used for parsing HTTP requests. José the library and tool for the Javascript Object Signing and Encryption. xinetd is a runtime dependency, the actual build proccess of the Tang does not require its libraries but to have its socket activation available in runtime. The bash dependency is there for a reason that Tang's tangd-update and tangd-keygen executables are bash scripts. These scripts are complex and are using data structures that are not available for OpenWrt's default shell - ash.

```
define Package/tang
  SECTION:=utils
  TITLE:=tang v$(PKG_VERSION) - daemon for binding data to a third party
  DEPENDS:=+libhttp-parser +xinetd +jose +bash
  URL:=https://github.com/latchset/tang
endef
```

The Tang package will be present in utils section of the Let us add a brief description to our new package using description define:

```
define Package/tang/description
    Tang is a small daemon for binding data to the presence of a third party
endef
```

The buildroot should know where to install the Tang's binaries. Let us define a install section and use standard tangd binary location as on Fedora OS:

[1] https://github.com/latchset/tang/releases/

```
define Package/tang/install
    $(INSTALL_DIR)  $(1)/usr/libexec
    $(INSTALL_BIN)  \
            $(PKG_INSTALL_DIR)/usr/lib/$(PKG_NAME)d*    $(1)/usr/libexec/
endef
```

Let us not forget the last line which allows the actual „magic“ to happen:

```
$(eval $(call BuildPackage,$(PKG_NAME)))
```

We can now merge these changes to feeds and try to build our freshly created Tang package:

```
$ git commit -a
$ git push --set-upstream origin add-tang
$ git checkout new_packages
$ git merge add-tang
```

The new packagee tang will be available in the *Extra packages* section of the menuconfig after updating feeds:

```
$ ./scripts/feeds update packages
$ ./scripts/feeds install jose
$ make menuconfig
$ make package/jose/{clean,compile}
```

After first try to build the tang package we will encounter the systemd dependency errror:

```
configure: error: Package requirements (systemd) were not met:

No package 'systemd' found

Consider adjusting the PKG_CONFIG_PATH environment variable if you
installed software in a non-standard prefix.
```

We did not defined it for Makefile but the cross-compilation of the package will try to configure and compile downloaded sources. Compiler will try to find the systemd dependency as it is defined in the „file“ in Tang repository. We shall remove this builtime dependency.

To do so we will remove a requirement for systemd from the Tang's configure.ac and Makefile.am file. These patches are too extensive to be demonstrated. *Makefile_am.patch* and *configure_ac.patch* can be found in submitted pull-request files[2] on GitHub.

To have sources patched before compilation we have to crate a directory for them in the package's feeds repository we forked on branch containing commit adding the Tang and copy them to created directory:

```
$ cd packages-OpenWrt
$ git checkout add-tang
$ mkdir -p utils/tang/patches
```

Patches included in this directory are automatically applied on the sources downloaded from the mirror in the build time.

To have these changes in our feeds in buildroot commit them and push to add-tang branch. After these changes pushed and merged with *new_packages* branch a rebuild of the tang package we will succeed. Now we have Tang package ready to be installed on our device.

We are avoiding troubles with using the libhttp-parser in version 2.8.0. These self issued problems are described in following subsection 6.2.1 Obstacles of delivering Tang. The most important part after successfull build would be to configure it right way.

[2] https://github.com/openwrt/packages/pull/5447/files

6.2.1 Obstacles of delivering Tang package

The upstream world is not always ideal. As this porting effort started OpenWrt upstream was in bad shape and almost dead for reasons we described in subsection **??** OpenWrt and LEDE. The latest release of the OpenWrt with codename „Chaos Calmer“ did not calm developers. The active part of them decided to focus on LEDE project and submitting a pull-request to the OpenWrt was painfull. Working with outdated buildroot (an SDK) mixed with upstream aiming host system such as Fedora as well. After OpenWrt waters calmed down we decided to use upstream version of buildroot and it solved many issues with outdated dependecies[3] that came into way.

Outdated buildroot caused many issues with dependecies but as we installed older version of them the build could be triggered. The most annoying thing was to run build over and over and collect linker and compiler errors and adding additional flags into Makefile such as:

```
+CFLAGS += -fhonour-copts
+TARGET_CFLAGS += $(FPIC) -std=gnu99
+TARGET_LDFLAGS += -Wl,-rpath-link=$(1)/usr/lib
```

Same flags and running build over an over were happening with José and with the Tang. In case of Tang there was one additional TARGET_CFLAGS option:

```
-D_GNU_SOURCE
```

Without this one Tang was issuing a compilation errors with implicit declaration of the functions:

```
dprintf()
vdprintf()
```

libhttp-parser dependency has been in its latest version 2.7.1 when the effort srarted. With this version we have spent the most of the time due to not paying attention and watching only a release numbers. The Fedora OS has packaged http-parser in same version so we assumed that it will be enough to update the version of the package already available in OpenWrt. Unfortunatelly it was sufficient until a first attempt to build the Tang package.

We have to realize that not only a OpenWrt packages can have patches packaged but also our host system distribution packages. Building the Tang package with libhttp-parser updated to version 2.7.1 failed on dependency and has thrown an error:

```
checking for http_parser.h... no
configure: error: http-parser required!
```

After days spent looking for cause we found out that Fedora's package http-parser contains one patch[4] to add functionality required by the Tang to the http-parser. Actual checking of the header was in configure.ac file of the Tang package:

```
AC_CHECK_HEADER([http_parser.h], [],
        [AC_MSG_ERROR([http-parser required!])], [
#include <http_parser.h>
#ifndef HTTP_STATUS_MAP
#error HTTP_STATUS_MAP not defined!
#endif
])
```

[3] https://github.com/openwrt/openwrt/pull/537/files
[4] https://github.com/nodejs/http-parser/pull/337

The fix of this issue was obvious. Add the very same patch to the OpenWrt's package libhttp-parser. As comments on appendix F List of pull-requests related to libhttp-update tells us there was a little problem with it. In the end the release 2.8.0 brought us a salvation and bump to this version solved a dependency error for a *HTTP_STATUS_MAP* macro.

On the other hand it brought us another issue. But watching at this following issue we were delighted and provided an easy fix for it rightaway:

```
Package tang is missing dependencies for the following libraries:
libhttp_parser.so.2.8
```

Adding a symbolic link to the libhttp-parser's install sections of the Makefile will suffice.

```
ln -s libhttp_parser.so.$(PKG_VERSION) libhttp_parser.so.2.8
```

Chapter 7

Tang on OpenWrt

Having the installation capable packages in our buildroot is only the half of the work done. We need to install them on the actual device and setup the enviroment especially for the Tang package to work correctly. The installation of the packages on OpenWrt is done with the opkg package manager.

The opkg (Open Package Management System) is a lightweight package manager used to download and install OpenWrt packages. These packages could be stored somewhere on device's filesystem or the package manager will download them from local package repositories or ones located on the Internet mirrors. Users already familiar with GNU/Linux package managers like dnf, yum, apt/apt-get, pacman, emerge etc. will definitely recognize the similarities. It also has similarities with NSLU2's Optware, also made for embedded devices.

Opkg attempts to resolve dependencies with packages in the available repositories/mirrors. If the opkg fails to find the dependency, it will report an error, and abort the installation of selected package.

Missing dependencies with third-party packages are probably available from the source of the package.

Remember that we removed systemd related lines from sources. Now we do not have automatic updates of the Tang's cache and keys are not generated when tang has empty database.

7.1 Install the packages

The packages that have been built in our buildroot are not available in any online mirror yet. To make it available for openwrt we should create a pull-request and hope for the comunity to accept it. Before we do so, we have to make sure, that the built packages are working with installing them and testing their functionality.

To get our newly built packages to the target device running the OpenWrt we can upload them using scp to device's filesystem. After successfull build the packages are present in the bin/packages/mips_24kc/packages/ directory:

```
scp bin/packages/mips_24kc/packages/*.ipk root@192.168.0.1:/root/custom-packages/
```

This command will upload every package built before in builroot to the device's /root/custom-packages/ directory. The buildroot should contain at least these files:

```
$ ls bin/packages/mips_24kc/packages/
bash_4.4.12-1_mips_24kc.ipk
```

```
jansson_2.10-1_mips_24kc.ipk
jose_10-1_mips_24kc.ipk
libhttp-parser_2.8.0-1_mips_24kc.ipk
libjose_10-1_mips_24kc.ipk
Packages
Packages.gz
Packages.manifest
Packages.sig
tang_6-1_mips_24kc.ipk
xinetd_2.3.15-5_mips_24kc.ipk
```

After the successfull upload connect to the device and install packages. We recommend installing only newly built or updated packages. opkg will resolve known dependencies from the mirrors and will install them. Make sure that you install them in right order. Due to fact that opkg not is not capable to resolve these new packages. Install the packages using opkg:

```
opkg install /root/custom-packages/jansson_2.10-1_mips_24kc.ipk
opkg install /root/custom-packages/jose_10-1_mips_24kc.ipk
opkg install /root/custom-packages/libhttp-parser_2.8.0-1_mips_24kc.ipk
opkg install /root/custom-packages/tang_6-1_mips_24kc.ipk
```

The opkg tool will resolve other dependencies and install them as well. After packages are installed we may now proceed to the enviroment setup.

7.2 Setting up the Tang keys

The Tang server is packaged with the cripts which help us to genarate keys and cache for the Tang daemon. Only problem with them was that OpenWrt has no realpath available therefore the `tangd-update` script did not work on OpenWrt.

```
bash: realpath: command not found
```

Simply replacing it with `readlink -f` solved an issue and script with such change is capable of generating cache. Creating a diff file for this change and adding it into *utils/tang/patches* should be in order.

The OpenWrt Makefile can define section „Package/$(PKG_NAME)/postinst“. This section usually contain a short shell script to tweak the package after installation to make package work out of the box. We can reflect behavior described above to this short sript and add it to the Tang's Makefile:

```
define Package/tang/postinst
#!/bin/sh
if [ -z "$${IPKG_INSTROOT}" ]; then
    mkdir -p /usr/share/tang/db && mkdir -p /usr/share/tang/cache
    KEYS=$(find /usr/share/tang/db/ -name "*.jw*" -maxdepth 1 | wc -l)
    if [ "${KEYS}" = "0" ]; then # if db is empty generate new key pair
        /usr/libexec/tangd-keygen /usr/share/tang/db/
    elif [ "${KEYS}" = "1" ]; then # having 1 key should not happen
        (>&2 echo "Please check the Tang's keys in /usr/share/tang/db \
and regenate cache using /usr/libexec/tangd-update script.")
    else
        /usr/libexec/tangd-update /usr/share/tang/db/ /usr/share/tang/cache/
    fi
    (cat /etc/services | grep -E "tangd.*8888\/tcp") > /dev/null \
        || echo -e "tangd\t\t8888/tcp" >> /etc/services
fi
endef
```

7.3 Configure Tang for xinetd

We need xinetd's socket activation for Tang to work. And to do so we will need a configuration file for the tangd service for xinetd daemon:

```
service tangd
{
    port            = 8888
    socket_type     = stream
    wait            = no
    user            = root
    server          = /usr/libexec/tangd
    server_args     = /usr/share/tang/cache
    log_on_success  += USERID
    log_on_failure  += USERID
    disable         = no
}
```

Listing 7.1: Configuration of Tang service for xinetd

This is configuration to run /usr/libexec/tangd anfter a request comes to port 8888. The server will be spawned with one argument, the cache directory. Please note that we used other directory compared to Fedora. We need to have Tang keys stored on persistent data storage. The /var/ location is only a symling to the /tmp directory on OpenWrt device. The developers on IRC channel proposed to use the persistent /usr/share/ directory for that purpose.

The last step before starting the Tang service on OpenWrt would be to setup /etc/services:

```
# echo -e "tangd\t\t8888/tcp" >> /etc/services
```

In such case of the accidental removal of this file we have to copy the backup of the file from devices ROM and edit it again.

```
# cp /rom/etc/services /etc/services
```

Now we shall restart the xinetd daemon using the xinetd script located in /etc/init.d/ directory of the device:

```
#/etc/init.d/xinetd stop
#/etc/init.d/xinetd start
```

To test that service is running we can use the telnet to the device on port defined in the xinetd configuration. The server should advertise its public key. It can be retrieved with simple GET request for /adv contenr from the server. Try telnet to device write „GET /adv HTTP/1.1“ and confirm this with two newlines. The ouput similar to following should appear:

```
$ telnet 192.168.0.1 8888
Trying 192.168.0.1...
Connected to 192.168.0.1.
Escape character is '^]'.
GET /adv HTTP/1.1

<unknown> GET /adv => 200 (src/tangd.c:85)
HTTP/1.1 200 OK
Content-Type: application/jose+json
Content-Length: 956
```

```
{"payload":"eyJrZXlzIjpbeyJhbGciOiJFQ01SIiwiY3J2IjoiUC01MjEiLCJrZXlfb3BzIjpb
ImRlcml2ZUtleSJdLCJrdHkiOiJFQyIsIngiOiJBR3V2amxUZmpYaDBraWFEa19Tak1vMGhYUm1R
dzFZNkVkNE9yN3Fza2J2c1h6QUlvRTl5ZnRwR2xRVng1OVlxZ1gtR3hQSE8tdzVLVXFmanRGQkVV
ZVByIiwieSI6IkFiNE9NNTBhQ1Y4NkdZVW1PdHBua1VTanNXNUFleENXZG5PZFEyakl0Z1RGNXNq
MG1TSFZCYXhsd2w1N3ZTUEdrSExiRl96SFlUVzlvVzNoTFJyeXRRNmYifSx7ImFsZyI6IkVTNTEy
IiwiY3J2IjoiUC01MjEiLCJrZXlfb3BzIjpbInZlcmlmeSJdLCJrdHkiOiJFQyIsIngiOiJBQVpS
ZmNlNUhLaFYyck1OQzhqLW5iVW1pdlh4NDRjcU1qX2Jvbk5OdnNTcXdBRjhFcGoyTFM0cFpfdUNR
VDJGVDRUSHJnX1Y4VHBKci1PQW41Z05CaUZNIiwieSI6IkFTMjJSdVJZZXVOU1ZtSkdfSmcwSW1n
by1LREd2QVFPZV9Vdk9fT3RZS3lGeUVoSWI1U0FLd3cwSEF0QjJFX2FTMG5pcWV3UUlod1QyanR5
eklCaWdkQU0ifV19","protected":"eyJhbGciOiJFUzUxMiIsImN0eSI6Imp3ay1zZXQranNvb
iJ9","signature":"ADpGOjYeB45MznQOxA6Pw9MXTMYQ649UkRSi_RsP8KPKosl-eA7GmOIiBM
FOoPnCNX-cGjhyDBbQuvESUCJ_3txjANf-srntxFAX5p72Eip-kROGrlCdLrVLnlg36itQUBFx7S
UyB_7I6CX6gdpWyJ-wtro8f2Snu6wwMGl-8V3ylWYr"}
```

The content of the first line from server response should be suspicious to us. According to the RFC 2616 the HTTP header should not contain such line[4].

```
<unknown> GET /adv => 200 (src/tangd.c:85)
```

After some investigation done in the Tang sources we found out that the line contain some debug information from the Tang daemon which were written to the /dev/stderr. From this we can assume that xinetd is sending also /dev/stderr to the created socker. To bypass this beavior we should write som wrapper script which will redirect the /dev/stderr ouptut somewhere else. We decided that this output may be helpful for debugging purpose and forwarded it to the some sort of log file in the /var/log/ directory of the device using this script:

```
#!/bin/bash
echo "==================================" >> /var/log/tangd.log
echo `date`: >> /var/log/tangd.log
/usr/libexec/tangd $1 2>> /var/log/tangd.log
```

Let us name this wrapper script the tangdw.

To get Tang to work we need to put invocation of this script in place where tangd binary was configured to. Move the script to the /usr/libexec directory where the tangd binary is to have it in one place and edit the line of xinetd configuration to run wrapper:

```
    server          = /usr/libexec/tangdw
```

The last thing would be to change the sript's permissions on device to allow execution.

We did a lot changes to the configuration. The Tang for the OpenWrt platform needs a xinetd configuration which for the ease of use could be packaged with it. The same with the wrapper script in order to have the service working correctly we also need this file. There is a way to add new files for each package and package them for the OpenWrt's opkg to install. To do so we will crate a directory for the files in the package's feeds repository we forked on branch containing commit adding the Tang and copy files there.

```
$ cd packages-OpenWrt
$ git checkout add-tang
$ mkdir -p utils/tang/files
```

As the new files are copied there we need to also edit a *Package/tang/install* section to install these files in proper directories (config file into dedicated /etc/xinetd.d/; wrapper into /usr/libexec/) as shown:

```
define Package/tang/install
    $(INSTALL_DIR)  $(1)/usr/libexec
    $(INSTALL_DIR)  $(1)/etc/xinetd.d/
    $(INSTALL_BIN)  \
```

```
            $(PKG_INSTALL_DIR)/usr/libexec/$(PKG_NAME)d*    $(1)/usr/libexec/
    $(INSTALL_BIN)  ./files/tangdw  $(1)/usr/libexec/
    $(CP)           ./files/tangdx  $(1)/etc/xinetd.d/
endef
```

After all changes done last thing would be to ammend commit on a branch, update the feeds in buildroot, rebuild the Tang package, upload and reinstall it on our device running the OpenWrt. Due to many changes to the Makefile and the Tang branch in packages see the submitted pull-request[1] where all changes are.

7.4 Contribute you work!

Note that every update and new package we did has its reference to an upstream pull-request. Have in mind that everyone can profit from our work. Do not be afraid to contribute to upstream. It is not always easy but following contribution guide for specific project is a big help. The result or even journey of the upstream contribution will definitelly get us some experience. Be tidy and pedantic, try to write easy to understand commits and files/pathches that have purpose and are well formated. It is the best way to have comunity on your side!

7.5 Tang's limitations

Let us sum up the limitations that Tang has on OpenWrt platform due to replaced systemd dependency with less capable xinetd implementation of super-server. We also found one platform independent limitation of the Tang's solution to full disk description on early boot.

Key exchange and generation of cache must be manual but can be automated with inotifytools to watch directory for change and run update script automatically. But this mean that the embeded device with such configuration will have another proccess always running. We decided to not implement automated update of the cachce in favor to computation resources being available to other services.

xinetd forwarding the stderr output to the socket as well is a little problem. To have tang working correctly we wrote a wrapper running the shell script redirecting the stderr into log file. Running some shell script takes some time compared to only running the binary file but for xinetd it is neccessary.

Early boot decryption using Wi-Fi network is a platform independent chicken-egg problem caused by information about wireless network being encrypted on system volume. The information needed to connect to wireless network needs to be decrypted but in order to decrypt them automatically we need to connect to network exposed to Tang server. It could be solved using a TPM to store information about such network but it is a security vulnerability and must be considered with a caution.

[1] https://github.com/openwrt/packages/pull/5447

Chapter 8

Conclusion

The Tang 3.2 server is a very lightweight program. It provides secure and anonymous data binding using McCallum-Relyea exchange algorythm.

As every server purpose is to serve its clients, it needs to have client application. In case of Tang we have Clevis. Clevis is a client software with full support for Tang. It has minimal dependencies and it is possible to use with HTTP, Escrow, and it implements Shamir Secret Sharing. Clevis has GNOME integration so it is not only a command line tool. Clevis also supports removable devices unocking using UDisks2 or even early boot integration with dracut. This was inspiration for this thesis with a goal to achieve it with OpenWrt embedded device running Tang server only.

To port Tang to OpenWrt system it was necesarry to port all its dependencies first.

After struggling with older version, package http-parser known as libhttp-parser in OpenWrt feeds, is now updated to latest upstream version 2.8.0 with STATUS_MAP patch. The systemd would be huge effort but tang's requirements are minimal and we were able to work with xinetd's socket activation. With correct configuration of xinetd and removing dependency for systemd Tang server is running on OpenWrt with some platform specific changes and limitations.

Bibliography

[1] Bauer, J.: *LUKS In-Place Conversion Tool*. [Online] Accessed 3 May 2017. Retrieved from: http://www.johannes-bauer.com/linux/luksipc/

[2] Braun, R.: *xinetd*. [Online] Accessed 1 May 2017. Retrieved from: http://web.archive.org/web/20051227095035/http://www.xinetd.org:80/

[3] Dini, F.: *cross compile tutorial*. [Online] Accessed 27 April 2018. Retrieved from: http://www.fabriziodini.eu/posts/cross_compile_tutorial/

[4] Fielding, R.; Gettys, J.; Mogul, J.; et al.: Hypertext Transfer Protocol – HTTP/1.1. Technical Report 2616. June 1999. obsoleted by RFCs 7230, 7231, 7232, 7233, 7234, 7235, updated by RFCs 2817, 5785, 6266, 6585. Retrieved from: http://www.ietf.org/rfc/rfc2616.txt

[5] Fruhwirth, C.: LUKS On-Disk Format Specification Version 1.1. *Changes*. vol. 1. 2005: pp. 22–01,. [Online] Accessed 22 April 2018. Retrieved from: http://tomb.dyne.org/Luks_on_disk_format.pdf

[6] Jones, M.: JSON Web Algorithms (JWA). Technical Report 7518. May 2015. Retrieved from: http://www.ietf.org/rfc/rfc7518.txt

[7] Jones, M.: JSON Web Key (JWK). Technical Report 7517. May 2015. Retrieved from: http://www.ietf.org/rfc/rfc7517.txt

[8] Jones, M.; Bradley, J.; Sakimura, N.: JSON Web Signature (JWS). Technical Report 7515. May 2015. Retrieved from: http://www.ietf.org/rfc/rfc7515.txt

[9] Jones, M.; Bradley, J.; Sakimura, N.: JSON Web Token (JWT). Technical Report 7519. May 2015. Retrieved from: http://www.ietf.org/rfc/rfc7519.txt

[10] Jones, M.; Hildebrand, J.: JSON Web Encryption (JWE). Technical Report 7516. May 2015. Retrieved from: http://www.ietf.org/rfc/rfc7516.txt

[11] Jones, M.; Sakimura, N.: JSON Web Key (JWK) Thumbprint. Technical Report 7638. September 2015. Retrieved from: http://www.ietf.org/rfc/rfc7638.txt

[12] Lehey, G.: *Porting UNIX Software: From Download to Debug*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.. 1995. ISBN 1-56592-126-7.

[13] Lehtinen, P.: *jansson*. [Online] Accessed 1 May 2017.
Retrieved from: https://github.com/akheron/jansson

[14] McCallum, N.: *jose*. [Online] Accessed 1 May 2017.
Retrieved from: https://github.com/latchset/jose

[15] Microsoft: *BitLocker*. [Online] Accessed 21 April 2018.
Retrieved from: https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview

[16] Miller, M.: Examples of Protecting Content Using JSON Object Signing and Encryption (JOSE). Technical Report 7520. May 2015.
Retrieved from: http://www.ietf.org/rfc/rfc7520.txt

[17] OpenWrt: *About OpenWrt*. [Online] Accessed 27 April 2018.
Retrieved from: https://openwrt.org/

[18] OpenWrt: *LEDE 17.01.1 - First Service Release - April 2017*. [Online] Accessed 27 April 2018.
Retrieved from: https://openwrt.org/releases/17.01/notes-17.01.1

[19] OpenWrt, W.: *Creating packages*. [Online] Accessed 28 April 2018.
Retrieved from: https://wiki.openwrt.org/doc/devel/packages

[20] OpenWrt, W.: *OpenWrt Feeds*. [Online] Accessed 28 April 2018.
Retrieved from: https://wiki.openwrt.org/doc/devel/feeds

[21] SplashData: *Worst Passwords of 2016*. [Online] Accessed 15 May 2017.
Retrieved from:
https://www.teamsid.com/worst-passwords-2016/?nabe=4587092537769984:2,6610887771422720:1&utm_referrer=https%3A%2F%2Fwww.google.cz%2F

[22] Steinbeck John, e. b. S. S.; Benson., J. J.: *Of men and their making*. London: Allen Lane The Penguin Press. 2002. ISBN 07-139-9622-6.

[23] Team, U. P. R.: Security Analysis of Cryptsetup/LUKS. 2012. [Online] Accessed 22 April 2018.
Retrieved from:
https://www.privacy-cd.org/analysis/cryptsetup_1.4.1-luks-analysis-en.pdf

[24] Thales: *Global Encryption Trends Study*. [Online] Accessed 1 May 2017.
Retrieved from:
http://images.go.thales-esecurity.com/Web/ThalesEsecurity/{5f704501-1e4f-41a8-91ee-490c2bb492ae}_Global_Encryption_Trends_Study_eng_ar.pdf

[25] vpnpick: *DD-WRT vs. Tomato vs. Open WRT?* [Online] Accessed 26 April 2018.
Retrieved from: https://vpnpick.com/dd-wrt-vs-tomato-vs-open-wrt/

[26] Wakefield, R. L.: Network Security and Password Policies. *The CPA Journal*. vol. 74, no. 7. 07 2004: pp. 6–6,8. copyright - Copyright New York State Society of Certified Public Accountants Jul 2004; Last updated - 2011-07-20; SubjectsTermNotLitGenreText - United States; US.
Retrieved from:
https://search.proquest.com/docview/212314970?accountid=17115

[27] Wiki, O.: *How to build*. [Online] Accessed 26 April 2018.
Retrieved from: https://wiki.openwrt.org/doc/howto/build

[28] Wikipedia: *Compiler*. [Online] Accessed 27 April 2018.
Retrieved from: https://en.wikipedia.org/wiki/Compiler

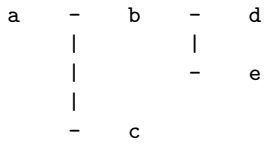
[29] Wikipedia: *C standard library*. [Online] Accessed 27 April 2018.
Retrieved from: https://en.wikipedia.org/wiki/C_standard_library

[30] Wikipedia: *Free On The Fly Encryption*. [Online] Accessed 3 May 2017.
Retrieved from: https://en.wikipedia.org/wiki/FreeOTFE

[31] Wikipedia: *Porting*. [Online] Accessed 1 May 2017.
Retrieved from: https://en.wikipedia.org/wiki/Porting

Appendix A

Compact disk content



Appendix B

Pre-installation enablement of hard drive encryption

B.1 Fedora 25 - disc encryption option selecting

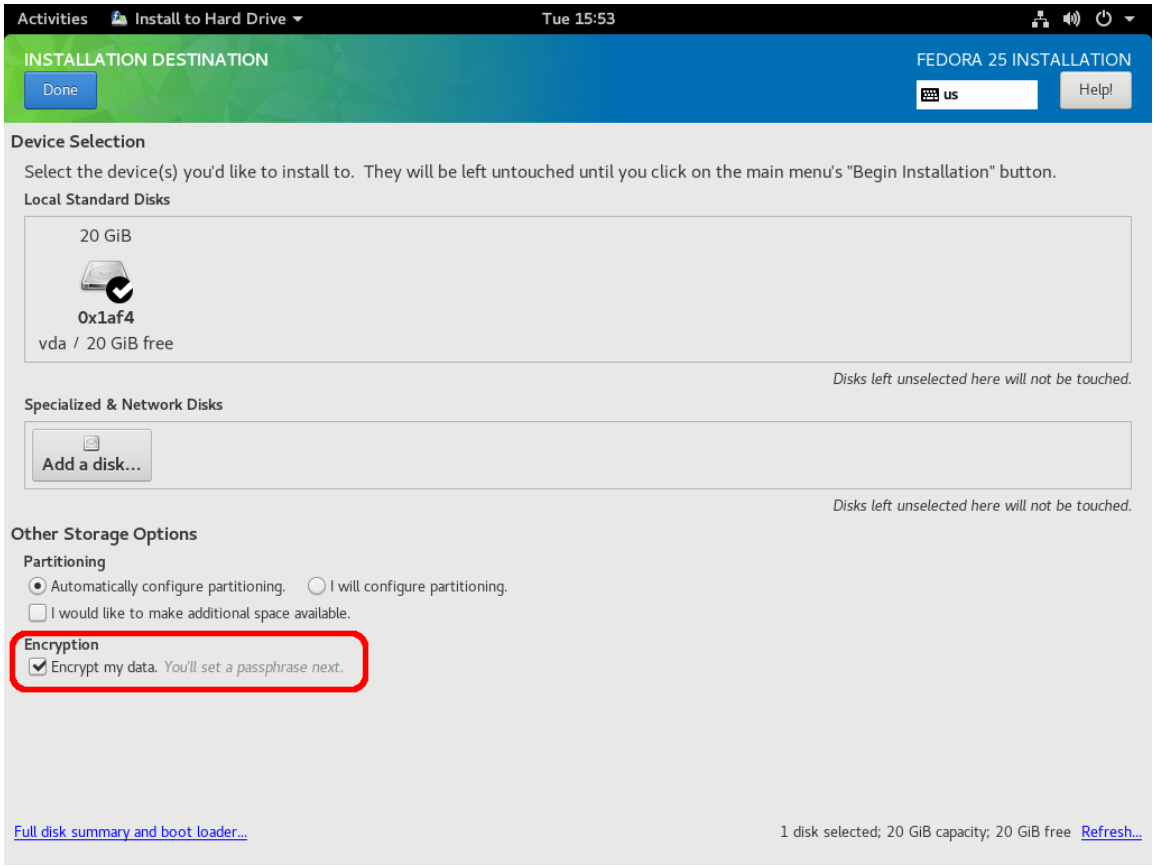


Figure B.1: Checking option

B.2 Fedora 25 - determination key encryption key

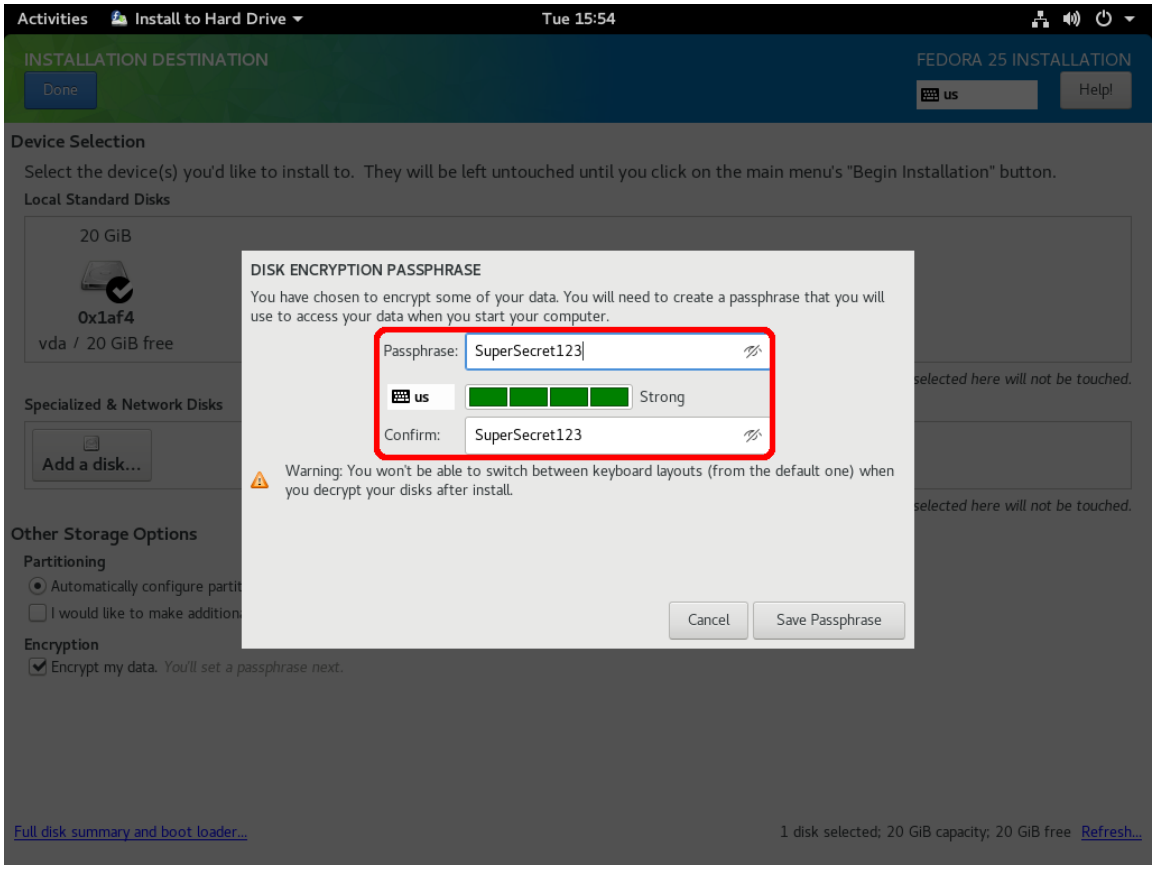


Figure B.2: Determining key

Appendix C

LUKS In-Place Encryption

It takes 4 steps to perform an in place encryption with *luksipc* [1]:

1. Unmounting the filesystem
2. Resizing the filesystem to shrink about 10 megabytes (2048 kB is the current LUKS header size – but do not trust this value, it has changed in the past!)
3. Performing luksipc
4. Adding custom keys to the LUKS keyring

C.1 Step 1 - unmounting

There should not be any problems unmounting partition, unless you want to encrypt / - the root partition, which in our case (to lock whole disk) will be necessary. To do so we need to restart our computer and boot any other or live distribution capable of completing these next steps.

```
# umount /dev/vda2
```

C.2 Step 2 - resizing

There are plenty tools for re-sizing, essentially for partitioning as whole (fdisk, e2fsck, etc.). Demonstrating how this is done for ext 2, 3, 4 here:

```
# e2fsck /dev/vda2
# resize2fs /dev/vda2 -s -10M
```

Delete and recreate shrinked partition with fdisk:

```
# fdisk /dev/vda
Welcome to fdisk (util-linux 2.23.2).

Changes will remain in memory only, until you decide to write them. Be careful before us

Command (m for help):
```

Check the partition number with typing the „p“:

```
Command (m for help): p
Disk /dev/vda: 407.6 GiB, 437629485056 bytes, 854745088 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disklabel type: dos
Disk identifier: 0x5c873cba
Partition 2 does not start on physical sector boundary.

Device         Boot         Start      End          Blocks       Id    System
/dev/vda1       *           2048       1026047      512000       83    Linux
/dev/vda2                   1026048    1640447      307200       8e    Linux LVM}
```

C.3 Step 3 - encrypting

After this, luksipc comes into play. It performs an in-place encryption of the data and prepends the partition with a LUKS header. Firt we have to download luksipc or install it with package manager.

```
$ wget https://github.com/johndoe31415/luksipc/archive/master.zip
$ unzip master.zip
$ cd luksipc-master/
$ make
```

Now run it with parameters like:

```
# ./luksipc -d /dev/vda2
```

luksipc will have created a key file /root/initial_keyfile.bin that you can use to gain access to the newly created LUKS device:

```
# cryptsetup luksOpen --key-file /root/initial\_keyfile.bin /dev/vda2 fedoradrive
```

C.4 Step 4 - adding key

DO NOT FORGET to add key to LUKS volume:

```
# cryptsetup luksAddKey --key-file /root/initial\_keyfile.bin /dev/vda2
```

Appendix D

Setting up the repository

To save our work progress and be able to contribute to the upstream repository we should have a own fork of it. A fork is a copy of a repository that we can manage. It lets us make changes to a project without affecting the original (upstream) repository. We can fetch updates from the upstream or submit changes to the original repository with pull requests. These pull request are generated from the „devel“ branch that we should have in our fork.

To fork OpenWrt's buildroot we should have our GitHub account set up[1], visit the OpenWrt's project upstream repository[2], and click on „Fork“ button in the top-right corner of the page. Before cloning our fork it is recomended to set ut git enviroment[3] on host machine and upload the SSH keys[4] to our account to minimize pushing effort to our fork. All work related to this thesis has been done using GitHub account Tiboris[5] therefore output of following commands are bound to it.

```
$ git clone https://github.com/Tiboris/openwrt.git ~/buildroot-openwrt
```

Please note that this command will clone the *openwrt* repository of the GitHub user *Tiboris* to the *~/buildroot-openwrt* directory on our host.

To catch up with changes made upstream we should consider to configure a remote that points to it. Configured remote allows us to sync changes made in the original repository with the fork and vice versa. Make sure you are in the buildroot directory and add remote[6] called upstream:

```
$ cd ~/buildroot-openwrt
$ git git remote add upstream \
    https://github.com/openwrt/openwrt.git
```

To verify that remote is present run:

```
$ git remote -v
origin  git@github.com:Tiboris/openwrt.git (fetch)
origin  git@github.com:Tiboris/openwrt.git (push)
upstream    https://github.com/openwrt/openwrt.git (fetch)
upstream    https://github.com/openwrt/openwrt.git (push)
```

[1] https://github.com/join
[2] https://github.com/openwrt/openwrt
[3] https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup
[4] https://help.github.com/articles/adding-a-new-ssh-key-to-your-github-account/
[5] https://github.com/Tiboris/
[6] https://help.github.com/articles/configuring-a-remote-for-a-fork/

With the remote conficured correctly we can now sync with upstream repository. To download latest upstream changes use the „fetch“ option and the „merge“ option to aply them to our fork's branch[7].

```
$ git fetch upstream master
remote: Counting objects: 4376, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4376 (delta 1776), reused 1775 (delta 1775),
 pack-reused 2599
Receiving objects: 100% (4376/4376), 1.27 MiB
 | 743.00 KiB/s, done.
Resolving deltas: 100% (2932/2932), completed
 with 809 local objects.
From https://github.com/openwrt/openwrt
 * branch                  master     -> FETCH_HEAD
   36fb0697e2..d089a5d773  master     -> upstream/master
$ git merge upstream/master
```

The detailed description can be found on original GitHub pages[8].

[7] https://help.github.com/articles/syncing-a-fork/
[8] https://help.github.com/articles/working-with-forks/

Appendix E

OpenWrt package's Makefile

This is an example of what OpenWrt Makefile for package could look like. The following makefile is slightly edited makefile from the tutorial[1] on the web.

```
#############################################
# OpenWrt Makefile for helloworld program
#
#
# Most of the variables used here are defined in
# the include directives below. We just need to
# specify a basic description of the package,
# where to build our program, where to find
# the source files, and where to install the
# compiled program on the router.
#
# Be very careful of spacing in this file.
# Indents should be tabs, not spaces, and
# there should be no trailing whitespace in
# lines that are not commented.
#
#############################################

include $(TOPDIR)/rules.mk

# Name and release number of this package
PKG_NAME:=helloworld
PKG_RELEASE:=1

# This specifies the directory where we're going to build the program.
# The root build directory, $(BUILD_DIR), is by default the build_mipsel
# directory in your OpenWrt SDK directory
PKG_BUILD_DIR := $(BUILD_DIR)/$(PKG_NAME)

include $(INCLUDE_DIR)/package.mk

# Specify package information for this program.
# The variables defined here should be self explanatory.
# If you are running Kamikaze, delete the DESCRIPTION
# variable below and uncomment the Kamikaze define
# directive for the description below
define Package/helloworld
	SECTION:=utils
```

[1] https://www.gargoyle-router.com/old-openwrt-coding.html

```
        CATEGORY:=Utilities
        TITLE:=Helloworld -- prints a snarky message
endef

# Uncomment portion below for Kamikaze and later delete DESCRIPTION variable above
define Package/helloworld/description
        If you can't figure out what this program does, you're probably
        brain-dead and need immediate medical attention.
endef

# Specify what needs to be done to prepare for building the package.
# In our case, we need to copy the source files to the build directory.
# This is NOT the default. The default uses the PKG_SOURCE_URL and the
# PKG_SOURCE which is not defined here to download the source from the web.
# In order to just build a simple program that we have just written, it is
# much easier to do it this way.
define Build/Prepare
        mkdir -p $(PKG_BUILD_DIR)
        $(CP) ./src/* $(PKG_BUILD_DIR)/
endef

# We do not need to define Build/Configure or Build/Compile directives
# The defaults are appropriate for compiling a simple program such as this one

# Specify where and how to install the program. Since we only have one file,
# the helloworld executable, install it by copying it to the /bin directory on
# the router. The $(1) variable represents the root directory on the router running
# OpenWrt. The $(INSTALL_DIR) variable contains a command to prepare the install
# directory if it does not already exist. Likewise $(INSTALL_BIN) contains the
# command to copy the binary file from its current location (in our case the build
# directory) to the install directory.
define Package/helloworld/install
        $(INSTALL_DIR) $(1)/bin
        $(INSTALL_BIN) $(PKG_BUILD_DIR)/helloworld $(1)/bin/
endef

# This line executes the necessary commands to compile our program.
# The above define directives specify all the information needed, but this
# line calls BuildPackage which in turn actually uses this information to
# build a package.
$(eval $(call BuildPackage,helloworld))
```

Listing E.1: Makefile for Helloworld.

Appendix F

List of pull-requests

Here is a list of all pull requests realated to the Tang porting effort.

zlib Not relevant: https://github.com/openwrt/packages/pull/4290/files

jansson Merged: https://github.com/openwrt/packages/pull/4289/files

libhttp-parser Not relevant: https://github.com/openwrt/packages/pull/4304/files
Closed: https://github.com/openwrt/packages/pull/4335/files
Merged: https://github.com/openwrt/packages/pull/5446/files

jose Open: https://github.com/openwrt/packages/pull/4334/files

tang Open https://github.com/openwrt/packages/pull/5447/files