Partitioning

Different partioning schemes and their setup

Understanding Linux file systems

Linux supports a number of different file systems with different sets of features and intended use-cases.

Ext4: The All-rounder

Ext4 is the latest iteration of the "Extended file system" and the default on most Linux distributions. It supports journaling, which means the file system keeps a list of files that are to be written to the disk and once the file has been written, it is removed from the journal. This improves file system integrity in case of a power loss. It also features delayed allocation, which aims to improve flash memory life. Ext4 also actively prevents file fragmentation when writing data.

Btrfs: The new kid on the block

Btrfs is a new type of Linux file system that is designed differently from Ext4 in some respects.

Btrfs is a copy-on-write (CoW for short) file system, which means that copies of files are only "virtual" and do not occupy any additional storage space, and a copy only becomes "real" once it has been changed. Writes do not overwrite data in place; instead, a modified copy of the block is written to a new location, and metadata is updated to point at the new location.

Btrfs organizes its data in subvolumes, which can be mounted like partitions. Unlike partitions, subvolumes do not have a fixed size. Instead, subvolumes are merely an organizational unit on the same Btrfs partition, the size of which depends on the contents stored in them. Any number of subvolumes can be created for different mount points, e.g. `/` and `/home`. This allows, amongst other things, for multiple operating systems to be installed to the same disk on the same computer without interfering with each other.

Another feature of Btrfs is its ability to create snapshots of the file system. The state of a subvolume can be recorded in a snapshot, e.g. before a critical system update, in order to revert to a previous state of the file system if necessary. Thanks to CoW, snapshots require very little storage space compared to full-fledged backups (although they are no replacement for them!) and can be mounted and booted from like regular subvolumes. This makes it possible to "rewind" the state of the file system with comparatively little effort. Tools such as `snapper` or `timeshift` can simplify and automate the process of creating during system updates and restoring from snapshots from the commandline.

Btrfs also implements transparent compression of data blocks. Written data is automatically stored in compressed form if the appropriate mount options are set. There are a number of different compression algorithms to choose from, including lz4, gzip and Zstandard. This can also increase the life span of flash based storage devices, as less data is written to the disk and not as much wear-leveling is taking place.

Btrfs comes with RAID management for RAID 0, 1 and 10 built into the file system itself and makes an additional software or firmware RAID superfluous for these configurations. In addition, the integrated RAID functionality offers the advantage that it is aware of used and free data blocks in mirrored setups, which can considerably speed up the reconstruction of a RAID, as only the used blocks are reconstructed. Using the built-in RAID functionality in Btrfs also allows for more storage devices to be added to the RAID later on.

XFS: large data made easy

XFS is a high-performance file system particularly proficient at parallel I/O due to its allocation group based design. This makes it ideal for when you're dealing with bandwidth intensive tasks, i.e. multiple processes accessing the file system simultaneously. Like ext4 it contains a journal for file system consistency.

XFS keeps an overview over the free space on the file system, allowing it to quickly determine free blocks large enough for new data in order to prevent file fragmentation.

Singular file system

The simplest, most basic partitioning scheme in any Linux operating system consists of 3 partitions:

Type	File System	Description
EFI System Partition	vfat	Stores boot loaders and bootable OS images in `.efi` format
Root File System	ext4, btrfs, XFS, or other	Stores the Linux OS files (kernel, system libraries, applications, user data)
Swap	Swap partition or file	Stores swapped memory pages from RAM during high memory pressure

This guide assumes the following:

- There is only 1 disk that needs partitioning
- `/dev/nvme0n1` is the primary disk

Preparing the disk

Determine the disks that are installed on your system. This can easily be done with `fdisk`:

```
fdisk -l
```

It outputs a list of disk devices with one or more entries similar to this:

```
Disk /dev/nvme0n1: 232.89 GiB, 250059350016 bytes, 488397168 sectors
Disk model: Samsung SSD 840
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

The line starting the device file with `/dev/` is the relevant one. Start partitioning the disk with `cfdisk`
:

WARNING: Make sure you are modifying the correct device, else you *will* lose data!

```
cfdisk /dev/nvme0n1
```

If the disk has no partition table yet, `cfdisk` will ask you to specify one. The default partition table format for UEFI systems is `gpt`. Create a layout with at least 3 partitions:

Size	FS Type
1G	EFI System
(RAM size)	Linux Swap
(remaining)	Linux root (x86-64)

NOTE: Specifying the correct file system type allows some software to automatically detect and assign appropriate mount points to partitions. See Discoverable Partitions Specification for more details.

You can verfiy that the partitions have been created by running `fdisk -l` again:

```
Disk /dev/nvme0n1: 232.89 GiB, 250059350016 bytes, 488397168 sectors
Disk model: Samsung SSD 840
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX

Device              Start       End   Sectors   Size Type
/dev/nvme0n1p1       2048   2099199   2097152     1G EFI System
/dev/nvme0n1p2    2099200  35653631  33554432    16G Linux swap
/dev/nvme0n1p3   35653632 488396799 452743168 215.9G Linux root (x86-64)
```

This time `fdisk` will also list the partitions present on the disk.

NOTE: You might notice a pattern with how Linux structures its block devices. Partitions also count as "devices" which you can interact with. Each partition has an incrementing counter attached to its name to specify its order in the partition layout.

Formatting partitions

Format the partition with the appropriate `mkfs` subcommand for the file system you want to use, e.g. ext4:

```
mkfs.ext4 /dev/nvme0n1p3        # ext4 root file system
mkfs.fat -F 32 /dev/nvme0n1p1   # EFI System Partition
mkswap /dev/nvme0n1p2           # Swap space
```

Next mount the file systems:

ATTENTION: Depending on which file system you chose earlier for your root file system, additional mount parameters might be beneficial or necessary, e.g. `btrfs` requires specifying the subvolume you want to mount using the option `subvol=NAME`. Refer to the file system's manual to determine relevant mount parameters.

```
mount /dev/nvme0n1p3 -o noatime /mnt
mount /dev/nvme0n1p1 --mkdir /mnt/efi
swapon /dev/nvme0n1p2
```

Singular file system (LUKS, encrypted)

LUKS (Linux Unified Key Setup) is the standard for Linux hard disk encryption. By providing a standard on-disk-format, it does not only facilitate compatibility among distributions, but also provides secure management of multiple user passwords. LUKS stores all necessary setup information in the partition header, enabling to transport or migrate data seamlessly.

Management of LUKS encrypted devices is done via the `cryptsetup` utility.

> **NOTE:** Why should you encrypt your data? Encryption ensures that no one but the rightful owner has access to the data. Encryption is therefore not only used to hide sensitive data from prying eyes, it also serves to protect your privacy. Encryption should be considered especially for portable devices such as laptops. In the event of loss or theft, encryption ensures that personal data and secrets (passwords, key files, etc.) do not fall into the wrong hands and are less likely and not as easily be abused.

The simplest, most basic encrypted partitioning scheme in a Linux operating system consists of 3 partitions:

Type	File System	Description
EFI System Partition	vfat	Stores boot loaders and bootable OS images in `.efi` format
Root File System	LUKS2	Stores the Linux OS files (kernel, system libraries, applications, user data)
Swap	Plain	Stores swapped memory pages from RAM during high memory pressure

This guide assumes the following:

- There is only 1 disk that needs partitioning
- `/dev/nvme0n1` is the primary disk

Preparing the disk

Determine the disks that are installed on your system. This can easily be done with `fdisk`:

```
fdisk -l
```

It outputs a list of disk devices with one or more entries similar to this:

```
Disk /dev/nvme0n1: 232.89 GiB, 250059350016 bytes, 488397168 sectors
Disk model: Samsung SSD 840
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

The line starting the device file with `/dev/` is the relevant one. Start partitioning the disk with `cfdisk`
:

> **WARNING:** Make sure you are modifying the correct device, else you *will* lose data!

```
cfdisk /dev/nvme0n1
```

If the disk has no partition table yet, `cfdisk` will ask you to specify one. The default partition table format for UEFI systems is `gpt`. Create a layout with at least 3 partitions:

Size	FS Type
1G	EFI System
(RAM size)	Linux Swap
(remaining)	Linux root (x86-64)

> **NOTE:** Specifying the correct file system type allows some software to automatically detect and assign appropriate mount points to partitions. See Discoverable Partitions Specification for more details.

You can verfiy that the partitions have been created by running `fdisk -l` again:

```
Disk /dev/nvme0n1: 232.89 GiB, 250059350016 bytes, 488397168 sectors
Disk model: Samsung SSD 840
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
```

```
Disklabel type: gpt
Disk identifier: XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX

Device              Start       End   Sectors   Size Type
/dev/nvme0n1p1       2048   2099199   2097152     1G EFI System
/dev/nvme0n1p2    2099200  35653631  33554432    16G Linux swap
/dev/nvme0n1p3   35653632 488396799 452743168 215.9G Linux root (x86-64)
```

This time `fdisk` will also list the partitions present on the disk.

> **NOTE:** You might notice a pattern with how Linux structures its block devices. Partitions also count as "devices" which you can interact with. Each partition has an incrementing counter attached to its name to specify its order in the partition layout.

Formatting partitions

Before writing a file system to the disk a LUKS container needs to be created with the `cryptsetup` utility:

> **WARNING:** Do **NOT** forget your passphrase! In case of loss you won't be able to access the data inside the container anymore!

```
cryptsetup luksFormat /dev/nvme0n1p3
```

Open the newly created LUKS container and supply the passphrase you just set:

> **NOTE:** `cryptroot` is used as an example here. It is the "mapper name" under which the opened LUKS container will be available at, in this example: `/dev/mapper/cryptroot`. You may use whatever name you like.

```
cryptsetup open /dev/nvme0n1p3 cryptroot
```

Formatting and mounting partitions

Create file systems for the ESP and the root file system:

```
mkfs.fat -F 32 /dev/nvme0n1p1
mkfs.ext4 /dev/mapper/cryptroot
```

Mount the file systems:

```
mount /dev/mapper/cryptroot -o noatime /mnt
mount --mkdir /dev/nvme0n1p1 /mnt/efi
```

NOTE: For an additional layer of security and privacy, swap space is going to be set up to be re-encrypted with a random passphrase on every boot in a later step. This way contents that have been swapped out of RAM and onto disk become inacessible after the machine has been powered off.

Encrypt non-root devices (LUKS)

If you have more than one hard disk that you need to encrypt (e.g. SSD as main disk, HDD as data disk) there are a few things to keep in mind to ensure continued smooth operation without any loss of convenience.

The layout is as follows:

Type	File System	Description
Home File System	LUKS2	Stores user home directories and personal files

Preparing the disk

Determine the disks that are installed on your system. This can easily be done with `fdisk`:

```
fdisk -l
```

Start partitioning the disk with `cfdisk`:

> **WARNING:** Make sure you are modifying the correct device, else you *will* lose data!

```
cfdisk /dev/sda
```

If the disk has no partition table yet, `cfdisk` will ask you to specify one. The default partition table format for UEFI systems is `gpt`. Create a layout with at least 3 partitions:

Size	FS Type
(disk size)	Linux home

> **NOTE:** Specifying the correct file system type allows some software to automatically detect and assign appropriate mount points to partitions. See Discoverable Partitions Specification

> for more details.

Formatting partitions

Before writing a file system to the disk a LUKS container needs to be created with the `cryptsetup` utility:

> **WARNING:** Do **NOT** forget your passphrase! In case of loss you won't be able to access the data inside the container anymore!

> **NOTE:** Using `/dev/sda` as an example of a SATA HDD that is intended to be mounted at `/home`.

```
cryptsetup luksFormat /dev/sda1
```

Open the newly created LUKS container and supply the passphrase you just set:

> **NOTE:** `crypthome` is used as an example here. It is the "mapper name" under which the opened LUKS container will be available at, in this example: `/dev/mapper/crypthome`. You may use whatever name you like.

```
cryptsetup open /dev/sda1 crypthome
```

Formatting and mounting partitions

Create a file system for the home file system:

```
mkfs.ext4 /dev/mapper/crypthome
```

Mount the file systems:

```
mount --mkdir /dev/mapper/crypthome -o noatime /mnt/home
```

LVM + dm-cache (unencrypted)

LVM dm-cache is a feature of the Linux device mapper, which uses a fast storage device to boost data read/write speeds of a slower one. It achieves this by transparently copying blocks of frequently accessed data to the faster storage device in the background. On subsequent reads/writes the faster storage device is queried first. If the requested data blocks are not on there, it automatically falls back on the slower source storage device.

This makes it possible to combine the benefits of SSD speeds with the low cost and high storage capacity of HDDs, when comparable pure SSD-based storage with the same capacity is too expensive or otherwise unavailable.

> **NOTE:** This partition scheme is tailored towards a desktop computer setup with enough RAM and no SWAP (and therefore no hibernate/suspend-to-disk support).

> **CAUTION:** This setup does **NOT** utilize LUKS disk encryption.

This guide assumes the following:

- `/dev/nvme0n1` is the primary disk (cache device)
- `/dev/sda` is the secondary disk (origin device)

Nomenclature

Term	Description
Physical Volume (PV)	On-disk partitioning format to be combined in a VG to a common storage pool
Volume Group (VG)	Grouping of one or more PVs to provide a combined storage pool from which storage can be requested in the form of LVs.
Logical Volume (LV)	Logical partition format which can be accessed like a block device to hold file systems and data.

Preparing the cache device

First the available disks need to be determined. This can easily be achieved with `fdisk`:

```
fdisk -l
```

To start the actual partitioning process start `cfdisk` and point it to the **disk** you wish to partition:

> **WARNING:** Make sure to select your actually desired device!

```
cfdisk /dev/nvme0n1
```

Partition the disk in the following way:

FS Type	Size	Mount Point	Comment
vfat	1G	/boot	EFI System
LVM	(remaining)		Linux LVM

Preparing the origin device

Partition the disk by starting `cfdisk` and pointing it to the **disk** for the origin device:

> **WARNING:** Make sure to select your actually desired device!

```
cfdisk /dev/sda
```

Partition the disk in the following way:

FS Type	Size	Mount Point	Comment
LVM	(all)		Linux LVM

Creating physical volumes, volume group and logical volumes

To create physical volumes as the basis for the LVM setup, use `pvcreate` and point it to the **partitions** you created in the two previous steps:

```
pvcreate /dev/nvme0n1p2   # SSD
pvcreate /dev/sda1        # HDD
```

Continue by creating a volume group with `vgcreate` that spans both physical volumes you just created:

> **NOTE:** `vg0` is used as an example here. Use whatever you like.

```
vgcreate vg0 /dev/nvme0n1p2 /dev/sda1
```

Next, create logical volumes inside the volume group with `lvcreate`, using 100% of the available space on the HDD and specifying the cache pool on the SSD:

```
lvcreate -l 100%FREE -n lv_root vg0 /dev/sda1
lvcreate --type cache-pool -n lv_cache -l 100%FREE vg0 /dev/nvme0n1p2
```

Finally, link the cache pool to the origin device with `lvconvert`:

```
lvconvert --type cache --cachepool vg0/lv_cache vg0/lv_root
```

Formatting devices

Format the partitions with the appropriate `mkfs` subcommand:

```
mkfs.fat -F 32 /dev/nvme0n1p1        # EFI System Partition
mkfs.btrfs /dev/mapper/vg0-lv_root   # Btrfs root file system
```

Mount the root Btrfs file system:

```
mount /dev/mapper/vg0-lv_root /mnt
```

Next, create the subvolumes with the `btrfs` user space tools:

```
btrfs subvolume create /mnt/@
btrfs subvolume create /mnt/@home
```

Unmount the root file system again:

```
umount -R /mnt
```

Mount the `@` subvolume at `/mnt`:

```
mount /dev/mapper/vg0-lv_root -o noatime,compress-force=zstd,space_cache=v2,subvol=@ /mnt
```

Create directories for subsequent mount points:

```
mkdir -p /mnt/{boot,home}
```

Mount the remaining file systems:

```
mount /dev/nvme0n1p1 /mnt/boot
mount /dev/mapper/vg0-lv_root -o noatime,compress-force=zstd,space_cache=v2,subvol=@home /mnt/home
```

LVM on LUKS (encrypted, Laptop)

LUKS (Linux Unified Key Setup) is the standard for Linux hard disk encryption. By providing a standard on-disk-format, it does not only facilitate compatibility among distributions, but also provides secure management of multiple user passwords. LUKS stores all necessary setup information in the partition header, enabling to transport or migrate data seamlessly.

Management of LUKS encrypted devices is done via the `cryptsetup` utility.

Nomenclature

Term	Description
Physical Volume (PV)	On-disk partitioning format to be combined in a VG to a common storage pool
Volume Group (VG)	Grouping of one or more PVs to provide a combined storage pool from which storage can be requested in the form of LVs.
Logical Volume (LV)	Logical partition format which can be accessed like a block device to hold file systems and data.

Partitioning Setup

NOTE: This partitioning scheme does **NOT** include an LVM cache device.

While it is technically possible to add an LVM cache device to this setup, it is not advised to do so, ***as this will leak plain text contents of the unlocked LUKS container into the cache,*** which can be read in a hex editor by opening the raw device file directly — entirely defeating the purpose of encrypting the disk!

A LUKS on LVM setup is recommended instead.

LVM on LUKS has the benefit of being able to encrypt an entire drive (useful for laptops with encrypted swap for resume) while only needing to provide a single passphrase to unlock it entirely for simplicity.

However, since the LVM container resides inside the LUKS container it cannot span multiple disks, as it is confined by the boundaries by the parent LUKS container.

This guide assumes the following:

- This is used on a laptop computer with resume capabilities (Swap partition)
- There is only one drive: `/dev/nvme0n1`
- The root file system will be btrfs, with subvolumes for `/` and `/home`
- To tighten security, this setup assumes a unified kernel image and booting via EFISTUB, with the *ESP* mounted at `/efi`. Extra steps will be necessary to make the machine bootable.

Preparing the drive

1. List available disks

   ```
   fdisk -l
   ```

2. Start partitionaing tool for primary disk (`cfdisk` *is a little easier to use as it has a nice TUI)*

   > **WARNING:** Make sure to select your actually desired device!

   ```
   cfdisk /dev/nvme0n1
   ```

3. Partition with the following scheme

FS Type	Size	Mount Point	Comment
vfat	1G	`/efi`	EFI System
LUKS	(remaining)		Linux file system

Creating the LUKS container

1. Create the LUKS container and enter a passphrase

   > **WARNING:** Do **NOT** forget your passphrase! In case of loss you won't be able to access the data inside the container anymore!

   ```
   cryptsetup luksFormat /dev/nvme0n1p2
   ```

2. Open the newly created LUKS container

   > **NOTE:** `cryptlvm` is used as an example here. Use whatever you like.

   ```
   cryptsetup open /dev/nvme0n1p2 cryptlvm
   ```

Creating LVM inside the LUKS container

1. Create an LVM physical volume inside LUKS container

   ```
   pvcreate /dev/mapper/cryptlvm
   ```

2. Create the volume group:

   ```
   vgcreate vg0 /dev/mapper/cryptlvm
   ```

3. Create the logical volumes

   > **NOTE:** When using resume, make `lv_swap` as large as RAM. In this example the machine has **16 GB** of RAM.

   ```
   lvcreate -L 16G -n lv_swap vg0       # Swap as big as RAM (16 GB)
   lvcreate -l 100%FREE -n lv_root vg0  # Root file system
   ```

Formatting devices

1. Create partitions

   ```
   mkfs.fat -F 32 /dev/nvme0n1p1        # EFI System Partition
   mkfs.btrfs /dev/mapper/vg0-lv_root   # Btrfs root volume
   mkswap /dev/mapper/vg0-lv_swap       # Swap space
   ```

2. Create Btrfs subvolumes

   ```
   # First, mount the root file system
   mount /dev/mapper/vg0-lv_root /mnt

   # Create subvolumes
   btrfs subvolume create /mnt/@
   btrfs subvolume create /mnt/@home
   ```

3. Mount partitions

```
# Unmount the root file system
umount -R /mnt

# Mount the @ subvolume
mount /dev/mapper/vg0-lv_root -o noatime,compress-force=zstd,space_cache=v2,subvol=@ /mnt

# Create mountpoints
mkdir -p /mnt/{efi,home}

# Mount the remaining partitions/subvolumes
mount /dev/nvme0n1p1 /mnt/efi
mount /dev/mapper/vg0-lv_root -o noatime,compress-force=zstd,space_cache=v2,subvol=@home
/mnt/home

# Activate swap
swapon /dev/mapper/vg0-lv_swap
```

LUKS on LVM (encrypted, cached, Desktop)

LUKS (Linux Unified Key Setup) is the standard for Linux hard disk encryption. By providing a standard on-disk-format, it does not only facilitate compatibility among distributions, but also provides secure management of multiple user passwords. LUKS stores all necessary setup information in the partition header, enabling to transport or migrate data seamlessly.

Management of LUKS encrypted devices is done via the `cryptsetup` utility.

Nomenclature

Term	Description
Physical Volume (PV)	On-disk partitioning format to be combined in a VG to a common storage pool
Volume Group (VG)	Grouping of one or more PVs to provide a combined storage pool from which storage can be requested in the form of LVs.
Logical Volume (LV)	Logical partition format which can be accessed like a block device to hold file systems and data.
Cache device	Fast storage used for caching reads/writes to slow storage
Origin device	Slow primary storage holding the actual data

Partitioning Setup

LUKS on LVM has the benefit of a LUKS container being able to span multiple disks, thanks to the machanisms of the underlying LVM. This, however, comes with the downside that if you want to have multiple volumes (e.g. for your root volume and a separate home volume or encrypted SWAP) you will have to take extra steps to unlock these volumes during the boot process.

> **NOTE:** If you want to utilize LVM cache this is the desired partioning scheme to use, as the encrypted LUKS container will reside inside an LVM LV and the LVM caching mechanism will cache the LV instead of the unlocked LUKS container, thus not leaking any secrets into the

> cache.

This guide assumes the following:

- This is used on a desktop computer without the need to resume (no SWAP partition)
- There are multiple drives: `/dev/nvme0n1` (SSD) and `/dev/sda` (HDD)
- The HDD will be cached by the SSD
- The root file system will be btrfs, with subvolumes for `/` and `/home`
- To tighten security, this setup assumes a unified kernel image and booting via EFISTUB, with the *ESP* mounted at `/efi`. Extra steps will be necessary to make the machine bootable.

Preparing partition layout

Start by listing available disks:

```
fdisk -l
```

Create a partition layout with `cfdisk` by pointing it to the first disk, e.g. `/dev/nvme0n1`:

> **ATTENTION:** `cfdisk` expects a device file, not a partition.

```
cfdisk /dev/nvme0n1
```

If `cfdisk` asks you about the partition table scheme to use, select `gpt`.

Create the following partition layout:

FS Type	Size	Mount Point	Comment
vfat	1G	/efi	EFI System
LVM	(remaining)		Linux LVM

Start `cfdisk` for the second disk, e.g. `/dev/sda`:

```
cfdisk /dev/sda
```

Create the following partition layout:

FS Type	Size	Mount Point	Comment
LVM	(all)		Linux LVM

Setting up LVM

Start by creating LVM PVs on the partitions we just laid out:

```
pvcreate /dev/nvme0n1p2   # SSD
pvcreate /dev/sda1        # HDD
```

Next, create a VG spanning both PVs:

> **NOTE:** `vg0` is used as an example here. Name your VG whatever you like.

```
vgcreate vg0 /dev/nvme0n1p2 /dev/sda1
```

Create an LV inside `vg0`, using 100% of the available space on the PV at `/dev/sda1` and label it `lv_root`:

```
lvcreate -l 100%FREE -n lv_root vg0 /dev/sda1
```

Create an LV inside `vg0`, using 100% of the available space on the PV at `/dev/nvme0n1p2` and label it `lv_cache`:

```
lvcreate -l 100%FREE -n lv_cache --type cache-pool vg0 /dev/nvme0n1p2
```

Finally, link both LVs together so that the LV on the HDD is being cached by the pool on the SSD:

```
lvconvert --type cache --cachepool vg0/lv_cache vg0/lv_root
```

Creating the LUKS container

Create the LUKS container inside the LV of the origin device:

> **WARNING:** Do **NOT** forget your passphrase! In case of loss you won't be able to access the data inside the container anymore!

```
cryptsetup luksFormat /dev/mapper/vg0-lv_root
```

Open the newly created LUKS container and supply the passphrase you just set:

> **NOTE:** `cryptroot` is used as an example here. Use whatever you like.

```
cryptsetup open /dev/mapper/vg0-lv_root cryptroot
```

Formatting and mounting partitions

Create file systems for the ESP and the root file system:

```
mkfs.fat -F 32 /dev/nvme0n1p1
mkfs.btrfs /dev/mapper/cryptroot
```

Mount the root btrfs file system and create the subvolumes:

```
mount /dev/mapper/cryptroot /mnt

btrfs subvolume create /mnt/@
btrfs subvolume create /mnt/@home
```

Unmount the root btrfs file system:

```
umount -R /mnt
```

Mount the `@` subvolume:

```
mount /dev/mapper/cryptroot -o noatime,compress-force=zstd,space_cache=v2,subvol=@ /mnt
```

Create mount points for `/efi` and `/home`:

```
mkdir -p /mnt/{efi,home}
```

Mount the remaining partitions and subvolumes:

```
mount /dev/nvme0n1p1 /mnt/efi
mount /dev/mapper/cryptroot -o noatime,compress-force=zstd,space_cache=v2,subvol=@home /mnt/home
```