

NixOS

Konfiguration, Tipps und Anleitungen rund um NixOS auf syu-pc.

- [Gaming Optimierungen](#)
- [NixOS Spickzettel](#)
- [Secure Boot mit eigenem Kernel unter NixOS \(lanzaboote\)](#)

Gaming Optimierungen

System

AMD Ryzen 9 7950X3D, AMD Radeon RX 7900 (Navi 31), 62 GB RAM, NixOS mit `linuxPackages_zen`.
Alle deklarativen Änderungen liegen in `/etc/nixos/configuration.nix` (Repo `nix-config`).

Kernel und Scheduler

```
boot.kernelPackages = pkgs.linuxPackages_zen;  
  
services.scx = {  
  enable = true;  
  scheduler = "scx_lavd";  
};
```

`linux-zen` bringt Gaming-orientierte Patches gegenüber dem Standard-Kernel mit. `scx_lavd` ist ein `sched_ext` Scheduler, der die asymmetrische Cache-Topologie des 7950X3D (zwei CCDs, eines mit V-Cache) berücksichtigt und Threads entsprechend platziert.

Arbeitsspeicher

```
zramSwap = {  
  enable = true;  
  algorithm = "zstd";  
  memoryPercent = 50;  
};  
  
boot.kernel.sysctl."vm.swappiness" = 120;
```

zram mit `zstd`-Kompression verhindert, dass bei RAM-Druck auf die langsamere Festplatte ausgelagert wird. `swappiness = 120` (Standard: 60) lässt den Kernel eher ins schnelle zram auslagern statt Page-Cache zu verdrängen.

Storage

```
services.udev.extraRules = ''  
    ACTION=="add|change", KERNEL=="nvme[0-9]n[0-9]", ATTR{queue/scheduler}="none"  
    '';
```

Ersetzt den Standard-Scheduler `kyber` durch `none` für alle NVMe-Geräte. NVMe-Queues regeln Reihenfolge und Latenz selbst besser, als es ein Software-Scheduler könnte.

GPU

```
hardware.amdgpu.overdrive = {  
    enable = true;  
    ppfeaturemask = "0xffffffff";  
};  
  
programs.corectrl.enable = true;  
  
programs.gamemode = {  
    enable = true;  
    settings = {  
        gpu = {  
            apply_gpu_optimisations = "accept-responsibility";  
            gpu_device = 1;  
            amd_performance_level = "high";  
        };  
    };  
};
```

`ppfeaturemask` entspermt die Overclocking-/Undervolt-Funktionen der amdgpu, die dann über `corectrl` (GUI) genutzt werden können. GameMode setzt den Performance-Level automatisch auf `high`, sobald eine Spielsession läuft (vorher: `auto`). `gpu_device = 1` entspricht `/sys/class/drm/card1`, der einzigen GPU in diesem System.

Gaming-Tools und Runtime

```

programs.steam = {
    enable = true;
    remotePlay.openFirewall = true;
    dedicatedServer.openFirewall = true;
};

programs.gamescope = {
    enable = true;
    capSysNice = true;
};

```

Dazu `mangohud` in `environment.systemPackages` für Performance-Overlay/Monitoring, sowie GE-Proton als Compatibility-Tool (`GE-Proton11-3` in `compatibilitytools.d`, manuell über ProtonPlus verwaltet statt über Nix).

Übersicht nach Impact

Änderung	Bereich	Impact	Begründung
<code>scx_lavd</code> Scheduler	CPU	Hoch	Cache-Topologie-bewusstes Scheduling auf dem X3D, wirkt direkt auf Frame-Times
<code>linux-zen</code> Kernel	Kernel	Mittel	Gaming-orientierte Patches, niedrigere Latenz als Standard-Kernel
GameMode GPU-Boost (<code>amd_performance_level=high</code>)	GPU	Mittel	Weniger Stutter/Mikroruckler bei plötzlichen Lastspitzen, kaum Effekt auf Average-FPS
GE-Proton	Compatibility	Mittel	Bessere Spielekompatibilität, teils bessere Performance als Standard-Proton
<code>ppfeaturemask</code> Overclocking-Unlock	GPU	Mittel (Enabler)	Ermöglicht OC/Undervolt über corectrl, ohne eigene Kurven kein direkter Effekt
zram (zstd)	Memory	Mittel	Verhindert Swap-Stalls auf Platte bei RAM-Druck
Gamescope (<code>capSysNice</code>)	Display	Mittel	Compositor-Bypass, Wirkung abhängig davon ob pro Spiel genutzt

Änderung	Bereich	Impact	Begründung
corectrl	GPU	Kein direkter Effekt	Reines Steuer-Tool, Wirkung hängt von manuell gesetzten Profilen ab
NVMe I/O-Scheduler none	Storage	Gering	Betrifft nur Ladezeiten/Asset-Streaming, nicht das Rendering
Swappiness 120	Memory	Gering	Nur bei RAM-Druck relevant, bei 62 GB selten der Fall
MangoHud	Monitoring	Kein Performance-Effekt	Reines Anzeige-/Messwerkzeug
Steam RemotePlay/DedicatedServer Firewall	Netzwerk	Kein Performance-Effekt	Feature-Enabler ohne Performance-Wirkung

NixOS Spickzettel

Kurzreferenz für die häufigsten NixOS-Kommandos: System updaten, `nix shell` nutzen, Pakete einmalig ausführen und nach Paketen suchen.

System updaten

System-Konfiguration neu bauen und aktivieren (nach Änderungen an `/etc/nixos/configuration.nix` oder dem Flake):

```
sudo nixos-rebuild switch
```

Nur testen (aktiviert erst beim nächsten Boot als Default, aber sofort für die aktuelle Session):

```
sudo nixos-rebuild test
```

Wenn mit Flakes gearbeitet wird:

```
sudo nixos-rebuild switch --flake /etc/nixos#hostname
```

Channels aktualisieren (klassisches, nicht-Flake-Setup):

```
sudo nix-channel --update
```

Alte Generationen aufräumen (Speicherplatz freigeben):

```
sudo nix-collect-garbage -d
```

`nix shell` nutzen

Startet eine temporäre Shell mit den angegebenen Paketen, ohne sie dauerhaft zu installieren:

```
nix shell nixpkgs#ripgrep nixpkgs#fd
```

Danach sind `rg` und `fd` in der aktuellen Shell verfügbar, verschwinden aber nach dem Verlassen der Shell wieder. Für das klassische (nicht-Flake) Äquivalent:

```
nix-shell -p ripgrep fd
```

Ein Paket einmalig ausführen (ohne Config-Eintrag)

Ideal, um ein Tool kurz auszuprobieren, ohne es in `configuration.nix` einzutragen:

```
nix run nixpkgs#hello
```

Klassisches Äquivalent:

```
nix-shell -p hello --run hello
```

Nach Paketen suchen

Über die Kommandozeile (Flakes/neues CLI):

```
nix search nixpkgs firefox
```

Alternativ über die Weboberfläche: <https://search.nixos.org/packages>

Nach NixOS-Optionen suchen (z. B. für `configuration.nix`):

```
man configuration.nix
```

Weboberfläche für Optionen: <https://search.nixos.org/options>

Need to know

- **Deklarativ statt manuell:** Runtime-Änderungen wie `sysctl foo=bar` von Hand oder ein manuell installiertes Paket überleben keinen Reboot bzw. werden beim nächsten `switch` überschrieben. Alles Dauerhafte gehört in `configuration.nix` / `flake.nix`.
- **switch vs. boot vs. test:** `switch` baut, aktiviert sofort und setzt den Eintrag als Boot-Default. `boot` baut und setzt den Boot-Default, aktiviert aber erst beim nächsten Neustart. `test` aktiviert sofort, ohne den Boot-Default zu ändern (praktisch zum Ausprobieren, ein

Absturz wird beim nächsten Reboot automatisch rückgängig gemacht).

- **Flakes sehen nur, was Git kennt:** Neue Dateien müssen mindestens per `git add` getrackt sein (nicht zwingend committet), sonst ignoriert der Flake-Build sie stillschweigend, mit teils verwirrenden Fehlermeldungen.
- **Home-Manager ist hier als NixOS-Modul eingebunden**, nicht standalone. Ein normaler `nixos-rebuild switch` übernimmt daher auch Änderungen an `home.nix`, ein separates `home-manager switch` ist nicht nötig.

Best Practices

- Vor jedem `switch` die Syntax prüfen, ohne gleich zu bauen:

```
nix-instantiate --parse configuration.nix
```

- Bei riskanteren Änderungen erst nur bauen, ohne zu aktivieren, um Eval-/Build-Fehler ohne Downtime abzufangen:

```
sudo nixos-rebuild build
```

- Config-Repo ist git-versioniert: vor jedem `switch` committen, damit jede funktionierende (oder kaputte) Version nachvollziehbar bleibt und sich per `git revert` zurückrollen lässt.
- Keine Secrets direkt in `configuration.nix` ablegen. Der Wert landet im Klartext im Git-Repo und im Nix-Store, wo er für alle lokalen Nutzer lesbar ist. Für Passwörter/Tokens eignen sich `sops-nix` oder `agenix`.

Nützliche Hinweise

- **Rollback bei kaputtem Rebuild:** `sudo nixos-rebuild switch --rollback`, oder im Boot-Menü eine ältere Generation auswählen.

- **Generationen auflisten:**

```
sudo nix-env --list-generations --profile /nix/var/nix/profiles/system
```

- **Lokale Options-/Manpage-Doku**, passend zur eigenen installierten Version und offline verfügbar: `nixos-help` bzw. `man configuration.nix`.
- **Flake-Inputs aktualisieren:** `nix flake update` aktualisiert alle Inputs in `flake.lock`, `nix flake lock --update-input nixpkgs` nur einen einzelnen.
- **Vorsicht mit `nix-collect-garbage -d`:** löscht alte Generationen inklusive der Möglichkeit, dorthin zurückzurollen. Vorher sicherstellen, dass die aktuelle Generation stabil läuft.

Secure Boot mit eigenem Kernel unter NixOS (lanzaboote)

Ausgangslage

Secure Boot ist ein System-weiter Schalter in der UEFI-Firmware: Ist er aktiv, muss **jede** EFI-Binary in der Bootkette signiert sein — Bootloader, Kernel, Initrd, egal von welchem Betriebssystem. Windows bringt seine eigenen (Microsoft-)Signaturen mit und läuft mit Secure Boot problemlos. Bei den meisten klassischen Linux-Distros signiert man Kernel/Bootloader einmalig mit `sbctl`.

Bei **NixOS** funktioniert das reine `sbctl`-Vorgehen aber nicht dauerhaft: Bei jedem `nixos-rebuild` landet ein neuer Kernel/Initrd im Nix-Store und wird neu nach `/boot` verlinkt — eine manuelle Signatur wäre nach dem nächsten Rebuild sofort wieder ungültig. Der Standardweg für NixOS ist deshalb **lanzaboote**: Es ersetzt `systemd-boot` als Bootmanager und signiert bei jeder Aktivierung automatisch mit.

Warum überhaupt Secure Boot aktivieren, wenn man "nur" Linux nutzt? In diesem Fall: weil zusätzlich **Windows 11** (für Spiele) auf derselben Maschine installiert werden soll, und Windows 11 Secure Boot + TPM 2.0 als Systemvoraussetzung verlangt.

1. Firmware in Setup Mode bringen

Im UEFI-Setup (BIOS) die vorhandenen Secure-Boot-Keys löschen bzw. zurücksetzen ("Clear Secure Boot Keys" / "Setup Mode"). Secure Boot bleibt dabei zunächst **aus** — es wird nur der Zustand hergestellt, in dem sich neue eigene Keys einspielen lassen.

2. lanzaboote als Flake-Input einbinden

In `flake.nix`:

```
inputs = {
  nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";
  lanzaboote.url = "github:nix-community/lanzaboote";
  lanzaboote.inputs.nixpkgs.follows = "nixpkgs";
  # ... weitere Inputs
};

outputs = { self, nixpkgs, lanzaboote, ... }: {
  nixosConfigurations.nixos = nixpkgs.lib.nixosSystem {
    system = "x86_64-linux";
    modules = [
      ./configuration.nix
      lanzaboote.nixosModules.lanzaboote
      # ... weitere Module
    ];
  };
};
```

Hinweis: Den `v0.4.2`-Tag von lanzaboote gibt es zwar, er ist aber teils inkompatibel mit einem aktuellen `nixpkgs-unstable` (die `boot.bootspec.enable`-Option wurde entfernt, Bootspec wird inzwischen immer generiert). Deshalb den Input ohne Versions-Pin auf `master` zeigen lassen — `github:nix-community/lanzaboote` ohne `/v0.4.2`-Suffix.

Danach den Lockfile-Eintrag für den neuen Input erzeugen:

```
nix flake lock --update-input lanzaboote
```

3. Bootloader in configuration.nix umstellen

`systemd-boot` wird von lanzaboote ersetzt, `canTouchEfiVariables` bleibt bestehen. Die Funktionssignatur der Datei braucht zusätzlich `lib` (für `lib.mkForce`):

```
{ config, lib, pkgs, ... }:
{
  boot.loader.systemd-boot.enable = lib.mkForce false;
```

```
boot.loader.efi.canTouchEfiVariables = true;

boot.lanzaboote = {
  enable = true;
  pkiBundle = "/var/lib/sbctl";
};
}
```

Wichtig: `pkiBundle` muss auf `/var/lib/sbctl` zeigen — das ist der Standardpfad, in den `sbctl create-keys` seine Keys schreibt. Ältere Anleitungen (auch eine frühere Version dieser Seite) nennen `/etc/secureboot`; das war der Pfad einer alten sbctl-Version und führt mit aktuellem sbctl zum Fehler `Failed to read public key from .../db.pem: No such file or directory`, weil dort nie etwas erzeugt wird.

Vor dem eigentlichen Rebuild lohnt sich eine reine Evaluation, um Tippfehler/Inkompatibilitäten ohne Build zu erkennen:

```
nix eval .#nixosConfigurations.nixos.config.system.build.toplevel.drvPath
```

4. Keys erzeugen, enrollen, Rebuild

Ab hier sind Root-Rechte nötig:

```
# Eigene Secure-Boot-Keys erzeugen
sudo sbctl create-keys

# WICHTIG: Microsoft-Keys mit-enrollen, sonst bootet Windows danach nicht mehr
sudo sbctl enroll-keys --microsoft

# Rebuild – lanzaboote signiert Kernel/Initrd/Bootloader automatisch mit
sudo nixos-rebuild switch --flake /etc/nixos#<hostname>

# Status prüfen (Secure Boot in der Firmware ist an dieser Stelle noch aus)
sudo sbctl status
bootctl status
```

`sbctl status` sollte alle relevanten Boot-Dateien als "signed" ausweisen.

5. Secure Boot aktivieren

Neu starten, im UEFI-Setup **Secure Boot aktivieren**, speichern. NixOS sollte danach ganz normal weiter booten — nach jedem künftigen `nixos-rebuild` signiert lanzaboote automatisch neu, ein manueller Schritt entfällt.

Windows 11 danach installieren

Da Secure Boot bereits mit den Microsoft-Keys enrolled ist, kann Windows 11 installiert werden, ohne Secure Boot zwischendurch wieder auszuschalten.

Für die Zielplatte gilt: Nicht von Linux aus manuell partitionieren/formatieren. Der Windows-Installer legt sein eigenes Layout an (EFI System Partition, MSR, primäre NTFS-Partition, Recovery-Partition) und formatiert dabei selbst. Es reicht, die alte Partitionstabelle zu entfernen und die Platte im Windows-Setup auszuwählen:

```
sudo umount /pfad/zum/mountpoint # falls die Platte gerade gemountet ist
sudo wipefs -a /dev/nvmeXn1      # Ziel-Gerät! Vorher unbedingt prüfen (lsblk),
                                # dass es wirklich die richtige Platte ist
```

Danach im Windows-11-Setup die Platte auswählen, vorhandene Partitionen löschen, Windows den Rest automatisch erledigen lassen.

Bootreihenfolge im Blick behalten: Manche Windows-Installationen räumen die EFI-Bootvariablen um. Nach der Windows-Installation lohnt sich ein Blick mit `efibootmgr`, ob der NixOS-Eintrag noch vorhanden und als Default gesetzt ist — falls nicht, hilft ein erneutes `sudo nixos-rebuild boot --flake /etc/nixos#<hostname>` bzw. `bootctl install`, um den Eintrag wiederherzustellen.