Winston White

Dr. Arslan Khan

CSE 597: Securing Embedded Systems

12 December 2025

nixWRehosT: Rehosting NixOS Firmware

1 Introduction

NixOS is an open-source, declarative operating system that is designed around the principle of complete reproducibility. The custom *Nix* package manager is used to centrally manage the entire system from software configuration to service daemons through a single file, or larger Git repository. This *configuration.nix* replaces imperative commands to install software with tools like *apt*, *yum*, or *dnf*. The expectation is that a user can transfer a NixOS repository anywhere and install an identical system down to exact dependency versions. NixOS can be consider Linux-based, as it still takes advantages of the Linux kernel. However, the NixOS designers achieve this unique reproducibility by significantly breaking user space guarantees.

NixOS does not follow the *Filesystem Hierarchy Standard (FHS)* and instead utilizes a distinctive *symlink-based* approach. Standard directories such as *usr* and *var* do not exist on a standard NixOS installation. Instead, each directory is a *symbolic link (symlink)* that references the *Nix Store*. The *Nix Store* is an immutable root directory where the logic from a NixOS configuration is internally resolved. Each dependency for a given *system upgrade* is written to its own isolated, immutable sub-directory named with a version hash. The thought process for this architecture is that it minimizes dependency conflicts while ensuring simple rollbacks in the case of a broken build. Static paths to certain standard directories like *lib* and *bin* are also internally patched in applications to point towards *Nix Store* locations.

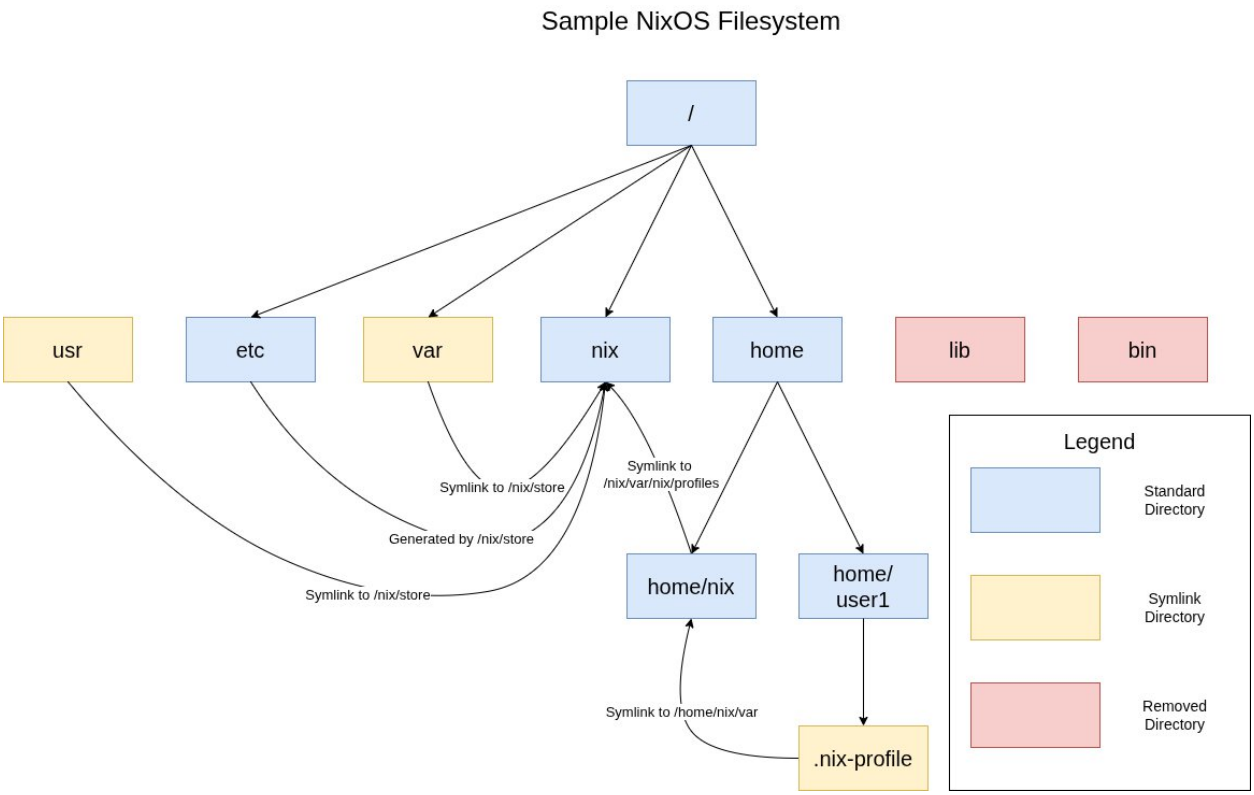

Figure 1: Sample Visualization of the NixOS Filesystem Structure

Despite the learning curve that it introduces, NixOS is becoming increasingly widespread in industry and open-source communities. This growth includes embedded systems and firmware vendors. Firmware images that run NixOS offer state-of-the-art reproducibility on low-power systems. Yet, this wave in enthusiasm for embedded NixOS has not propagated into academia at this time. As NixOS-based firmware becomes more popular, there will need to be an dedicated effort towards an open-source rehosting framework. In this paper, we propose a novel rehosting approach for *firmware images with NixOS as the core operating system*. This approach *nixWRehosT* intended to address the nuances of embedded NixOS implementations while expanding from prior Linux-based rehosting frameworks. As an open-source project, *nixWRehosT* and its artifacts are publicly available on GitHub at:

- https://github.com/winstonwumbo/cse-597-nixWRehosT

2 Literature Review

NixOS, in the context of embedded systems, has been a relatively new application within both industry and academia. Although the concept of NixOS originally began in 2003, these efforts worked towards standard desktop environments on *x86* and *ARM* architectures. *reUpNix: Reconfigurable and Updateable Embedded Systems*, the oldest research paper on NixOS implications for an embedded system, was only published in 2023. [5] Work on *NixWRT*, an open-source project to adapt NixOS for router firmware, began quite recently as well in 2018. [9], [10] Overall, we were only able to identify 5 total research papers that are relevant to embedded NixOS during the semester. This youth and lack of ongoing research indicates that there is still significant room for improvement in support of embedded NixOS projects.

Nix as a Declarative Solution for Embedded Security Challenges and System Administration Problems, a recent master's thesis from 2025, approaches embedded NixOS from the perspective of a higher-level security analyst. It investigates how NixOS might improve the response outcomes for defensive *blue teams* and offensive *red teams* that monitor embedded devices. As a result, this thesis is not a code-base assessment and does not evaluate NixOS-based firmware from the perspective that we are interested in. [1]

2.1 NixOS in Real-Time and Automotive Software

In *Design and Implementation of an Accessible Real-Time Simulation Framework Using Low-Cost Hardware and Open-Source Software*, Naqvi et al. detail the creation of an affordable aerospace simulation framework extended from NixOS. The system is a collaboration between Technische Universität Clausthal and the German Aerospace Center, Institute of Flight Systems. It implements a closed-source fork of NixOS for real-time operating systems (RTOS), which is named *NixOS-RT*. This is combined with a *Raspberry Pi* and *FastAPI* to produce high quality

analytics. Naqvi et al. demonstrated effective simulation in comparison with contemporary FPGA systems. Nonetheless, this publication is not related to the implications of NixOS for lower level firmware, and its artifacts have not been made publicly available. [2]

Uppsala University has produced two master's theses on the topic of NixOS for automotive systems. Kamini discusses method to partially adapt the reproducibility of NixOS into his custom *Micronix* package manager for *QNX 8*, a popular RTOS for automobiles. Rather than completely porting NixOS, this is a lightweight native application to demonstrate how Nix concepts can be feasibly imitated in QNX. Akula, on the other hand, designed a quantitative evaluation to compare NixOS performance with the conventional *apt* package manager on both Linux and QNX. This thesis demonstrates how NixOS can be consistently faster in aggregate, while its symlink-based nature still has timing implications for QNX. Each thesis is again not focused on embedded firmware, rather pursuing the reimplementation of *Nix-inspired* systems for another operating system. Neither thesis has open-source code for the work performed. [3], [4]

2.2 NixOS in Firmware Development

reUpNix: Reconfigurable and Updateable Embedded Systems proposes a methodology to apply NixOS in an updateable and reconfigurable build system for embedded firmware. Gollenstede et al. address several shortcomings with NixOS for creating firmware images, including data duplication for system size. The publication was able to achieve an 86% reduction in the base image size. It also discusses other improvements such as an A/B bootloader partition, allowing an image to switch between generations of NixOS updates. This work is the closest in concept to *nixWRehosT* but from the opposite perspective. *reUpNix* creates standard Linux

firmware images using NixOS as the build system, while *nixWRehosT* intends to be a platform for rehosting firmware that internally runs NixOS itself. [5]

2.3 Current Firmware Rehosting Solutions

Greenhouse is a modern framework that focuses on single-service rehosting of Linux-based firmware binaries in user-space emulation. Rather than booting a full firmware image, Greenhouse extracts and emulates individual services, significantly reducing the complexity of peripheral modeling and kernel dependencies. This approach enables scalable dynamic analysis of network-facing services but sacrifices full-system fidelity, making it less suitable for validating interactions between services, init systems, and the underlying operating system. While Greenhouse is effective for conventional Linux firmware, it assumes a FHS-compliant layout and does not account for many of the alternative models used by NixOS. [6]

Penguin extends prior rehosting work like *Greenhouse* by introducing a target-centric methodology. Instead of adapting firmware to fit an emulator, Penguin incrementally adapts the emulation environment to the firmware. This includes automated kernel selection, filesystem repair, and syscall compatibility adjustments by tweaking *QEMU* configuration. Similar to Greenhouse, Penguin presumes traditional Linux user space guarantees. This limits Penguin's applicability to NixOS-based firmware without significant modification. [7]

Peripheral modeling frameworks such as P2IM address a different but complementary challenge in firmware rehosting. P2IM focuses on automatically modeling hardware peripherals to emulate firmware in the absence of real hardware. This approach is particularly effective for bare-metal or RTOS-based firmware, where hardware interactions are tightly coupled to execution flow. However, P2IM does not address higher-level operating system concerns such as

init systems or filesystem reconstruction. Consequently, it does not directly support Linux nor NixOS-based firmware images. [8]

3 Design of NixWRehosT

nixWRehosT is developed as an automated system for rehosting NixOS-based firmware images. It is comprised of five main components: the Extractor, Runner, Checker, Patcher, and Viewer. For an easily accessible environment, all components are housed within a *Vagrant* virtual machine. This ensures that all users are able to reproduce the experiment without complex system configuration. Docker is not used because NixOS and its related utilities rely on the *Systemd* initialization service, which is discouraged for running inside a container.

The intended target firmware for *nixWRehosT* is sample set of open-source firmware projects provided by the GitHub community. *NixWRT*, the primary focus of this platform, is a popular firmware image for home routers that reimplements *OpenWRT*. *NixWRT* is designed for *MIPS* architecture and will not natively run on an *x86* device. *Jetpack-NixOS* provides firmware for the Nvidia Jetson and is funded by Anduril Industries. Anduril is a defense contractor with a long history in embedded systems and NixOS. The final set of images planned for *nixWRehosT* are from the *NixOS-Baloo* family, which are designed as a proof-of-concept (PoC) for NixOS on Chromebook and Android motherboards. [8]-[12]

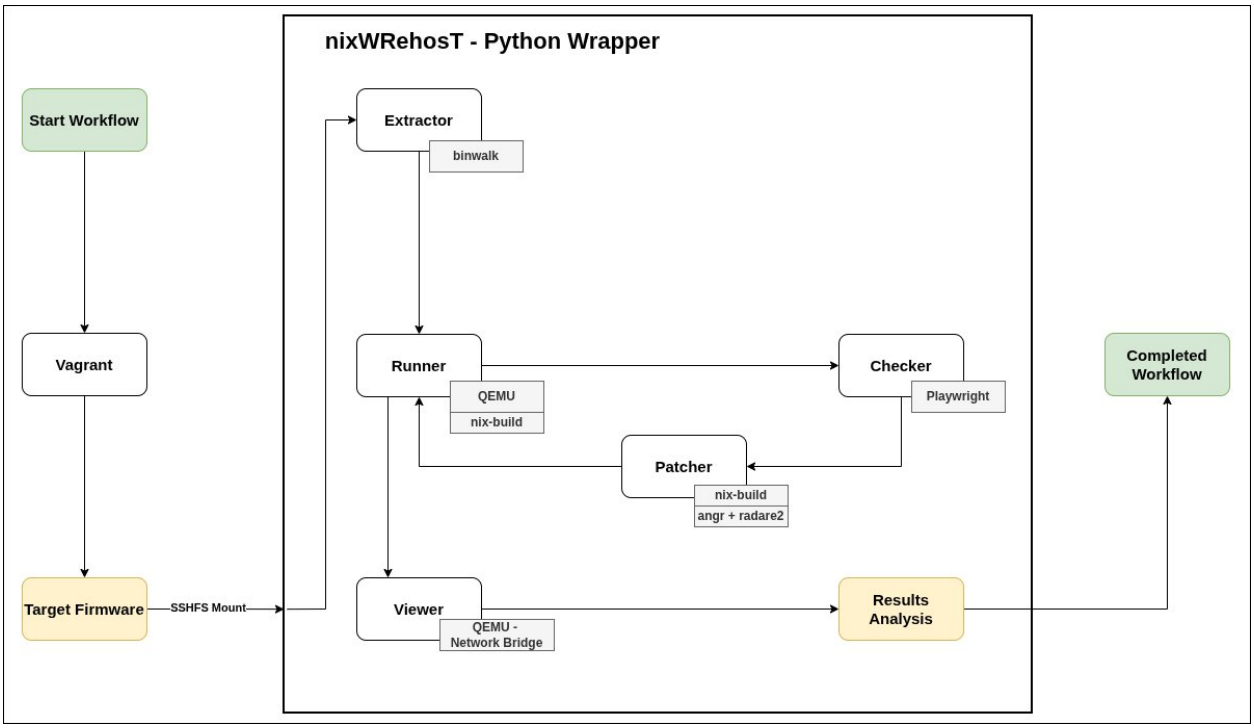

Figure 2: nixWRehosT Proposed Architecture

Rather than manually activating each component, a Python wrapper script is provided to improve the user experience. The overall architecture is inspired by existing rehosting frameworks such as Greenhouse and Penguin, while implementing additional components to handle the nuances of NixOS on user space. [6], [7]

nixwrehost.py > ...

```python
import subprocess
from pathlib import Path

firmware = input("Firmware image to rehost: ")
subprocess.run([
    "binwalk",
    "-eM",
    "--preserve-symlinks",
    f"{firmware}"
    ])

if not Path("result/run.sh").is_file():
    subprocess.run([
        "nix-build",
        "-I", "rehost-config=./qemu/nixwrt-build.nix",
        "--arg", "device", "import ./qemu",
        "--argstr", "image", f"_{firmware}.extracted/",
        "-A", "outputs.default"
    ])
```

```python

print("Starting nixWRehosT Runner...")
bg_proc = subprocess.Popen(
    [
        "bash", "result/run.sh"
    ],
    stdin=subprocess.DEVNULL,
    stdout=subprocess.DEVNULL,
    stderr=subprocess.DEVNULL
)
print("Started nixWRehosT Runner: PID ", bg_proc.pid)

print("Starting nixWRehosT Viewer (terminal will be taken over)...")
subprocess.run([
    "nix-shell", "-p", "qemu", "--run",
        " ".join([
            "qemu-system-x86_64",
            "-echr", "16",
            "-m", "1024",
            "-cdrom", "systemrescue-12.02-amd64.iso",
            "-netdev", "socket,mcast=230.0.0.1:1235,localaddr=127.0.0.1,id=lan",
            "-device", "virtio-net,disable-legacy=on,disable-modern=off,netdev=lan,mac=ba:ad:3d:ea:21:01",
            "-display", "none",
            "-serial", "mon:stdio"
        ])
    ],
    stdin=None,    # attach to your actual terminal
    stdout=None,   # default, terminal output
    stderr=None    # defaul)
)

print("Exited nixWRehosT Viewer...")
```

Figure 3: nixWRehosT Wrapper Script: Written in Python to run Bash subprocesses

3.1 Extractor

Similar to existing solutions in rehosting, *nixWRehosT* uses the *Binwalk* application for extracting the initial filesystem. Binwalk is executed with the *–e* argument to write an extracted filesystem to the same directory, then the *–M (Mashotrya)* and *—preserve-symlinks* arguments to attempt to preserve symlinks on the root of the filesystem. *SquashFS* is the most common filesystem type among the sample images of NixOS firmware. This provides a starting point for the *nix-build* utility to rebuild the *Nix Store* during the *Runner* component.

```
Last login: Fri Dec 12 17:28:11 2025 from 192.168.121.1
vagrant@ubuntu2204:~$ cd nixwrehost/
vagrant@ubuntu2204:~/nixwrehost$ python3 nixwrehost.py
Firmware image to rehost: images/nixwrt.bin
```

Figure 4: CLI prompt to enter path towards NixOS-based firmware binary

3.2 Runner

The *Runner* is the primary component of the *nixWRehosT* platform. This is the component which actually rehosts the firmware, allowing validation of fidelity and other aspects in the process. First, we run the *nix-build* utility pointed towards our extracted root filesystem. An instance of the *Nix* package manager is installed within the base virtual machine. Using arguments declared in the *nixwrt-build.nix* and *qemu.nix* configuration files, the package manager is able to rebuild broken *Nix Store* paths that were not maintained by Binwalk. The *nix-build* utility will read our Nix code from these files and perform operations originally intended for migrating desktops from one device to the next. It had not been commonly applied to firmware, but successfully worked for our purposes in this circumstance.

Afterwards, a nested instance of QEMU is run for user mode emulation of the NixOS-based firmware image. This QEMU *runner* is managed by the *Nix* package manager as well to provide a cohesive, automated system. The *qemu-deps* submodule on GitHub provides additional Nix code to support this. Available samples of firmware are for *MIPS* motherboards, so the QEMU *runner* is customized with options to enhance emulation fidelity specifically for *MIPS*.

3.3 Viewer

The *Viewer* component is another nested instance of QEMU within the main *Vagrant* virtual machine. This component is provided to allow interactive testing of network capabilities of the rehosted firmware images. The *NixWRT* samples are intended as router firmware. This simplifies validation of intended states. We utilize the *SystemRescue* live image to easily access a serial console for initiating commands along an internal *QEMU bridge network* between the two nested instances. It is downloaded using *curl* when the *Vagrant* virtual machine is first initialized.

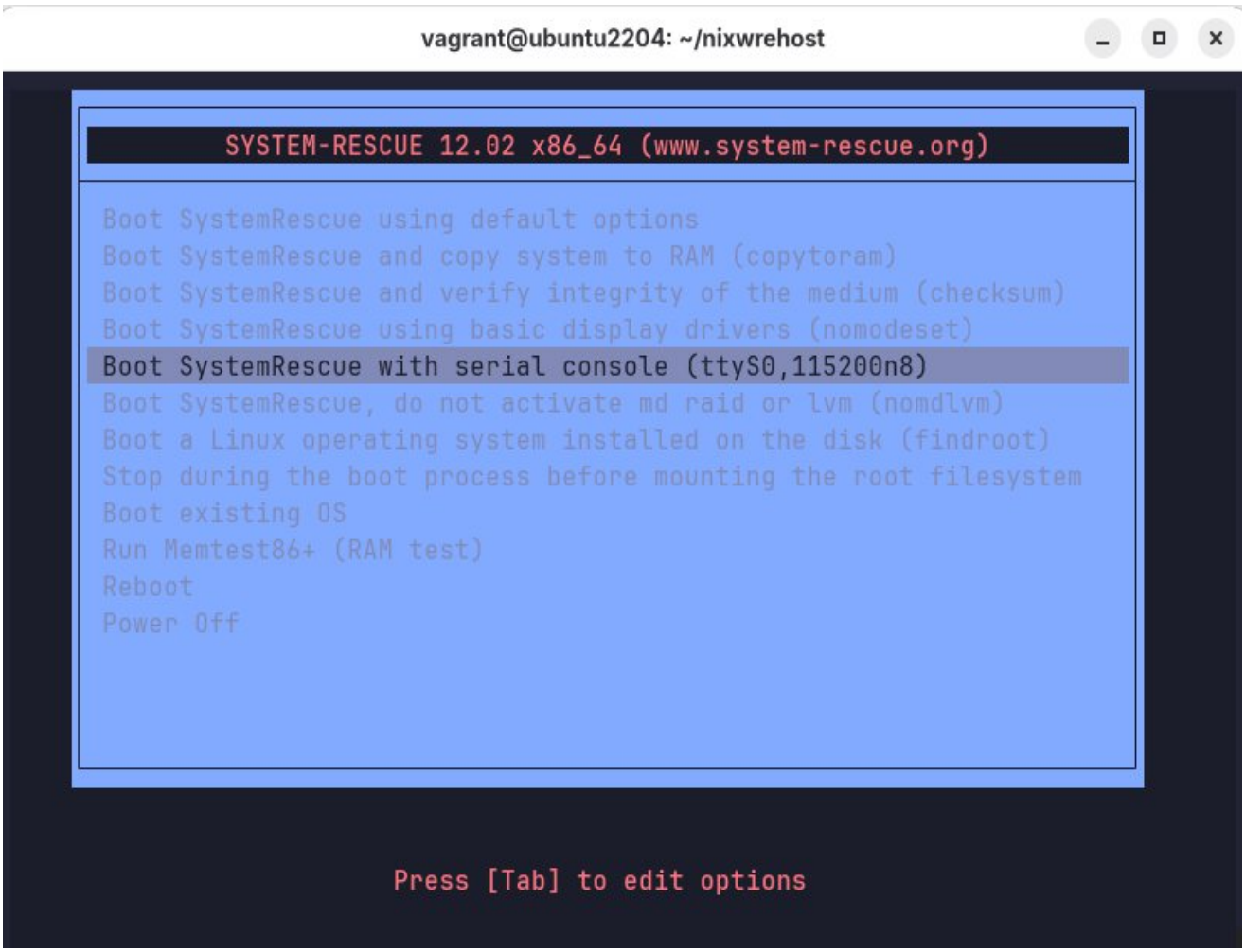

Figure 5: Select Boot SystemRescue with serial console

Once the *Viewer* component is finished booting *SystemRescue*, simple console commands can be initiated on the *10.3.0.x* network segment. *NixWRT* images will be assigned to *10.3.0.1* by default. Initiating a *ping* command should result in a successful back-and-forth response over the *ICMP* protocol.

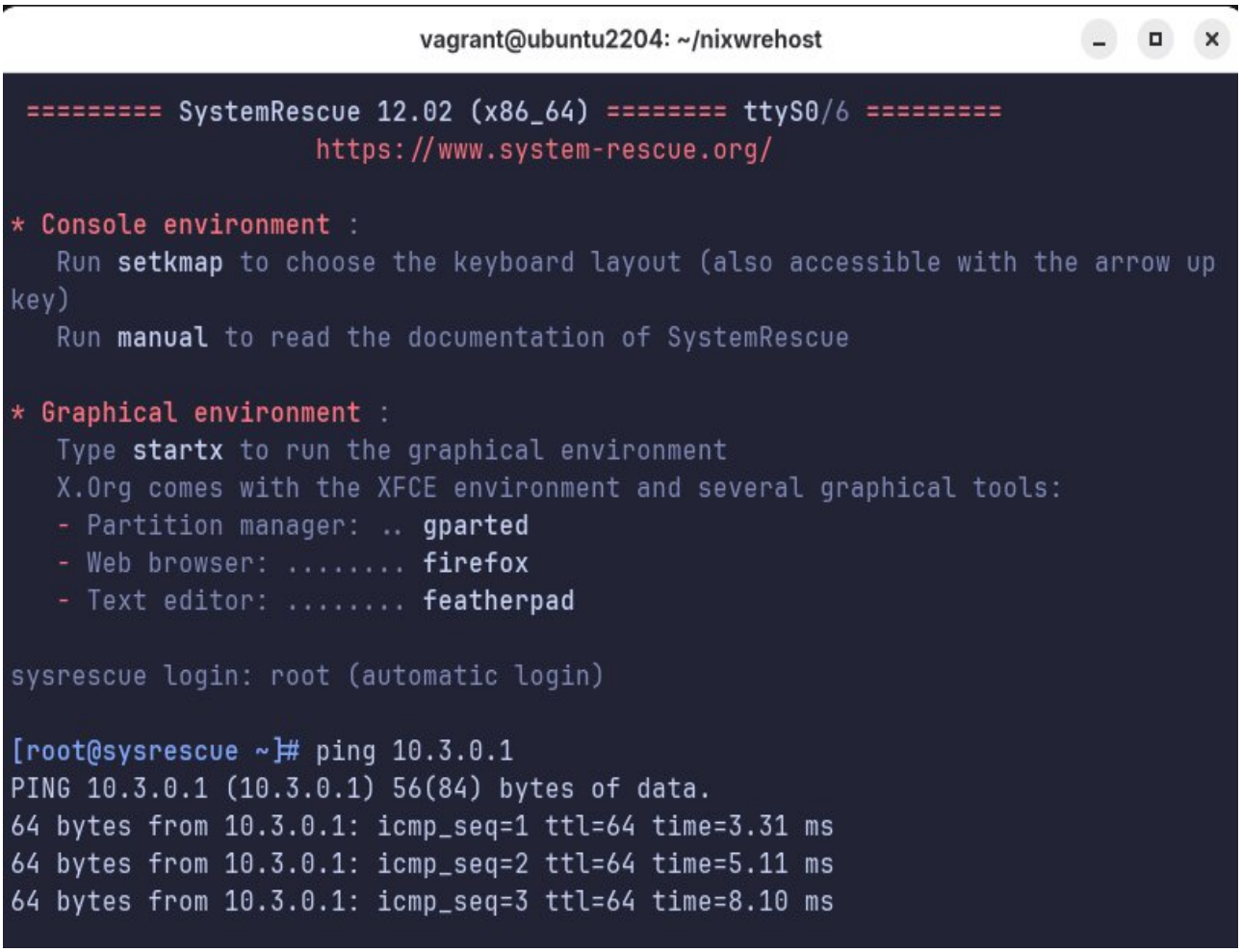

Figure 6: Enter ping command within SystemRescue serial console

We can further stress test the fidelity of emulation for the network stack on the *NixWRT* firmware image. Remote connection to the *NixWRT* instance should be supported through a *Secure Shell (SSH)* session. For the demonstration version, the default credentials for this *NixWRT* image will be a username *root* and password *secret*. Once a successful connection is established with *NixWRT*, users can explore the filesystem through BusyBox utilities. The repaired *Nix Store* paths will be visible in the */nix/store* directory.

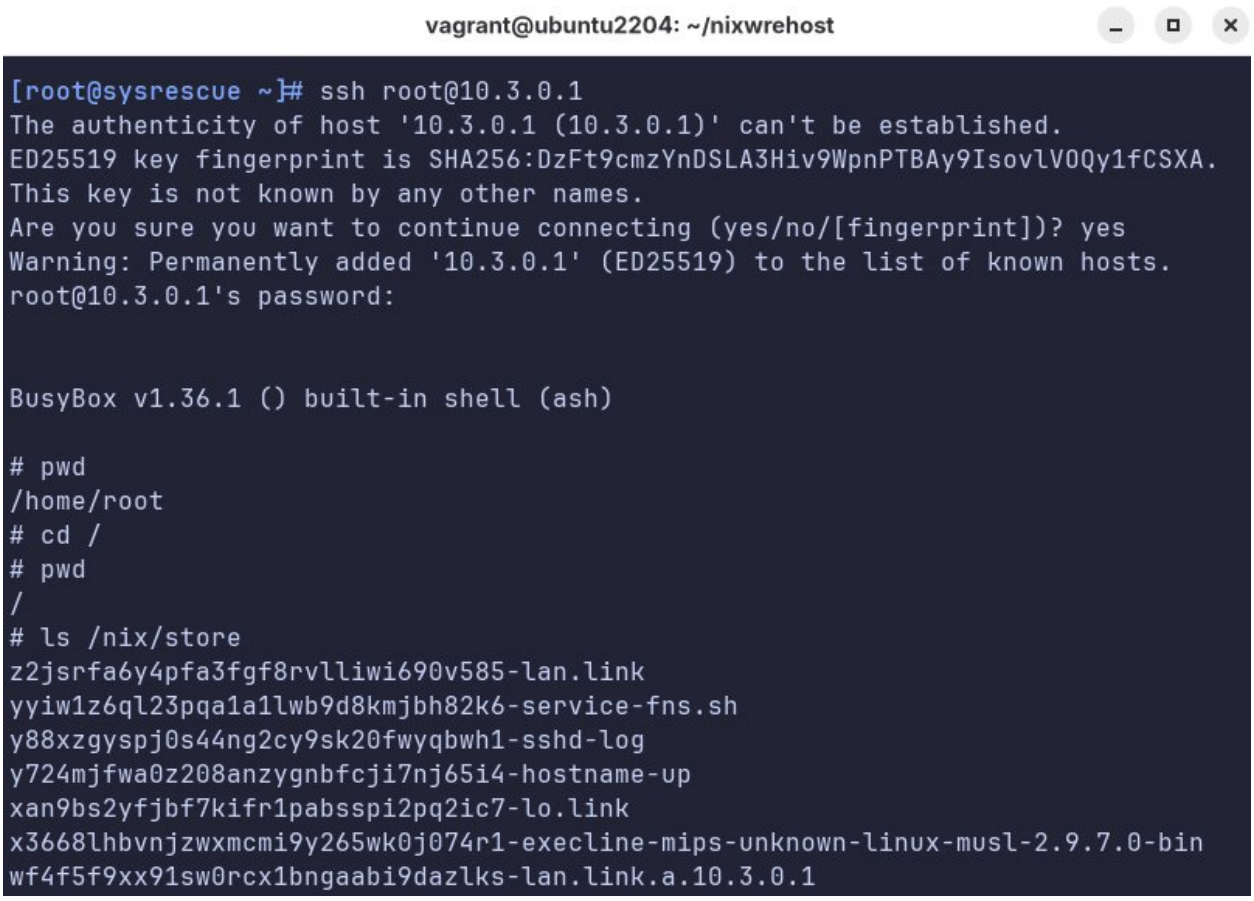

Figure 7: Sample of a rehosted NixWRT image

4 Discussion

4.1 Open Challenges

The most immediate obstacle this semester was compiling viable NixOS-based firmware images for unit testing, surprisingly. Although many projects were discovered, such as *NixWRT*, *NixOS-Baloo*, and *Jetpack-NixOS*, many of these community projects turned out to be unmaintained. Some ran older versions of NixOS and need packages that have been removed from the central repository. In a similar situation, we discovered third-party dependencies that had been removed from platforms like *Rust Cargo* and *Python Pip*. This suggests a wave of excitement for embedded NixOS in the open-source community, which may have cooled down later. An additional issue was discovered with Anduril, creator of *Jetpack-NixOS*. Although the

company heavily relies on NixOS for embedded systems, this project is their only open-source firmware image and is poorly documented.

Much of the early semester was spent trying to debug these issues across multiple images, rather than dedicating focus on rehosting a single image. This *tunnel vision* on obtaining multiple viable images may have been poor within the scope of the semester project. In the second half of the semester, I settled down to focus on rehosting the *NixWRT* images in particular. This highlights time that could be better spent improving the *Ongoing Work* goals. [8]-[12]

4.2 Ongoing Work

Currently *nixWRehosT* is only partially implemented despite its successful demonstration. While the *nix-build* patches are high enough fidelity to emulate services like *SSH*, more complex *NixWRT* services like the graphical *OpenWRT* web interface still fail. The *Extractor, Runner,* and *Viewer* components are completed, but a fully automated workflow with the *Checker* and *Patcher* components will be an ongoing effort into the next semester. As a result, *nixWRehosT* only runs with the initial patches provided to *QEMU* by the *nix-build* configuration and *Binwalk*. It cannot self-sufficiently detect bugs and update itself at the moment.

The prospective design for these components would include popular automation frameworks, similar to *Greenhouse* and *Penguin*. The *Playwright* automation framework is the planned choice for the *Checker* to test the fidelity of the network stack, including the Web interface. It is a more modern alternative to *Selenium* and directly integrates with browsers using a *WebSocket* rather than relying on heavy browser drivers. The subsequent *Patcher* component would start with modifying the *QEMU* configuration through *nix*-build, then implement the industry standards of *angr* and *radare2* for deeper binary analysis if needed. [6], [7]

References

[1] E. Korte, “Nix as a Declarative Solution for Embedded Security Challenges and System Administration Problems,” MS thesis, University of Turku, 2025. [Online]. Available: https://www.utupub.fi/bitstream/handle/10024/180653/Korte_Eino_Thesis.pdf

[2] M. A. Naqvi, S. Gupta, U. Durak, and S. Hartmann, *Design and Implementation of an Accessible Real-Time Simulation Framework Using Low-Cost Hardware and Open-Source Software*, vol. 48. 2025, p. 21. doi: 10.11128/arep.48.a4800.

[3] A. J. Kamini, “An Evaluation of Nix Principles for Automotive Software Package Management,” MS thesis, Uppsala University, 2025. [Online]. Available: https://uu.diva-portal.org/smash/get/diva2:2002562/FULLTEXT01.pdf

[4] A. S. Akula, “Evaluating Nix for Automotive Embedded Targets,” MS thesis, Uppsala University, 2025. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-568601

[5] N. Gollenstede, U. Kulau, and C. Dietrich, “reUpNix: Reconfigurable and Updateable Embedded Systems,” *LCTES 2023: Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 40–51, Jun. 2023, doi: 10.1145/3589610.3596273.

[6] H. J. Tay *et al.*, “Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation,” *Proceedings of the 32nd USENIX Conference on Security Symposium*, Art. no. 324, Aug. 2023, doi: 10.5555/3620237.3620561.

[7] A. Fasano *et al.*, “Target-Centric Firmware Rehosting with Penguin,” *Network and Distributed System Security (NDSS) Symposium 2025*, Jan. 2025, doi: 10.14722/bar.2025.23010.

[8] B. Feng, A. Mera, and L. Lu, “P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling (extended version),” *arXiv.org*, Sep. 13, 2019. https://arxiv.org/abs/1909.06472.

[9] D. Barlow, “NixWRT: Build images for embedded MIPS SoCs using NixPkgs (experimental),” *GitHub*, Jul. 2018. https://github.com/vkleen/nixwrt.

[10] D. Barlow and NixCon2018, “NixWRT: Purely Functional Firmware Images for IoT,” *TIB AV-Portal*. Oct. 2018. [Online]. Available: https://av.tib.eu/media/39604.

[11] A. Gautier, “NixOS-Firmware: Baloo,” *GitHub*, Dec. 2021. https://github.com/baloo/nixos-firmware.

[12] Anduril Industries, “Jetpack-NixOS: NixOS module for NVIDIA Jetson devices,” *GitHub*, Sep. 2022. https://github.com/anduril/jetpack-nixos.