Docker Introspection of Rehosted Firmware

By Diego Contreras
Advisors: William Oliver, Zackary Estrada

Introduction and Motivation

Firmware rehosting is the process of taking the firmware of an embedded system, then simulating it in a virtual environment so that its behaviour can be analyzed as if it were running on the original system. As seen in Fig 1., a rehosted router firmware would believe that despite being simulated on a different computer, it believes it is still on a router. The IGLOO project at MIT Lincoln Laboratory uses the Penguin rehosting framework to achieve this on multiple Linux-based platforms by using the fw2tar tool to get bootable virtual machines from firmware images. [1]

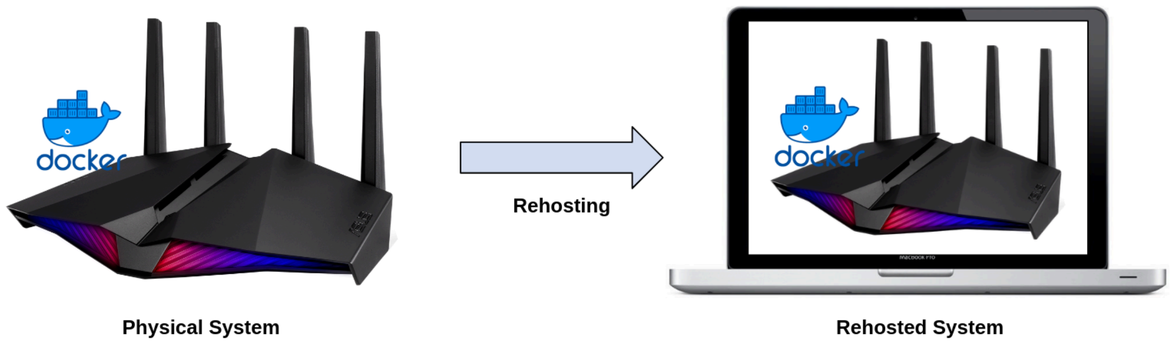


Fig 1. Modified rehosting diagram from [2]

Many modern embedded systems have adopted Docker to isolate services, simplify updates, and make their software modular. Both BalenaOS and Home Assistant are examples of embedded operating systems specifically designed to run docker containers on embedded systems, whether it be for the industry or for personal use.

As IGLOO beings to look into these types of containerized firmware, we would like to answer questions such as:

- Did the containers start in their rehosted image?
- Is the rehosted environment healthy enough to be considered trustworthy?
- What information can be extracted by docker introspection?

Therefore, the goal of this UROP was to design and test a Docker introspection mechanism for Penguin, and to evaluate how well IGLOO can support containerized firmware.

This report describes the design of a Python-based Penguin plugin, two case studies with BalenaOS and Home Assistant OS, the technical obstacles encountered, and proposed future work to make container introspection a supported feature of IGLOO rehosting.

Background

Penguin is a firmware rehosting tool developed by IGLOO that takes a target-centric approach [1]. Specifically, Penguin generates a configuration file tailored to the target's boot process, expected devices, and workarounds [1]. The typical workflow to get a firmware running into Penguin is to use the fw2tar tool (also developed by IGLOO) to extract a .rootfs.tar.gz file from a disk image, which contains the entire Linux filesystem of that firmware [1]. Penguin then takes that file system and simulates it keeping in mind detected abnormalities.

Docker is an open-source application for deploying and running docker containers. Many modern applications today are written as an orchestration of Docker containers, and have sparked the need for firmwares such as BalenaOS and HomeAssistant to support Docker natively on embedded devices. Because these containers essentially act as isolated operating systems, having tools within Penguin to analyze them within a rehosted firmware would make dynamic analysis a lot easier and forgo the need to develop specialized tools.

Docker follows a client-server model. There is the dockerd daemon that implements the heavy logic and acts as a server, and the docker CLI tool that sends requests to the daemon via a REST API on a Unix socket [3]. On most modern Linux distributions, the dockerd daemon is installed as a systemd service, and often uses a feature called "socket activation" that starts the daemon only when it receives a request [3]. That means for a rehosted systemd system to support Docker, the rehost needs to successfully boot systemd, successfully start Docker services, and provide a working virtual filesystem to support the Unix socket.

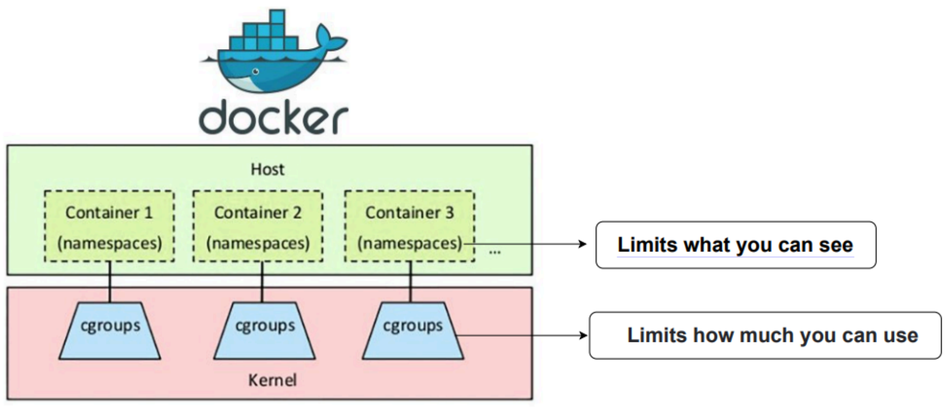


Fig 2. Docker cgroup from [4]

Docker introspection is the process of obtaining detailed information on a docker container. It can be done in one of two ways: the REST API dockerd provides as a web server, or by manually digging into the underlying implementation of dockerd. For this UROP, the latter approach was chosen to provide as much detail as possible. To that end, both of these kernel features that dockerd is comprised of were analyzed:

- cgroups are a Linux kernel feature that allows an application to be put into a “cgroup” and have its usage of CPU, memory, network bandwidth, and more limited and metered. Docker uses this to control how much system resource a container can use. [5]
- Namespaces are another Linux kernel feature that fool an application into believing that its PID, owner, accessible files, and more are different than what it actually is. Docker uses this to have the applications in its container believe they are in their own operating system. [5]
- As shown in Fig 2., each container in docker has its own namespace and cgroup. That means docker introspection is a per-container process. [5]

Introspection Plugin

I started by working through IGLOO rehosting modules hosted on pwn.college that covered the usage of fw2tar and Penguin, and common rehosting pitfalls. This provided a baseline understanding of interpreting boot logs, adjusting configs, Penguin plugins, and how to rehost any piece of firmware on the internet. After that, I looked through the penguin sources code to decide how docker introspection could be implemented.

Penguin supports Python plugins that run on the host and communicate with the rehosted system via internal APIs. Because of this, I decided to implement docker introspection as a plugin for several reasons:

- Communication with the rehosted system is already figured out by Penguin
- Plugins have access to the rehosted system’s console and has logging
- Plugins can easily be reused by future IGLOO users.

For a first proof of concept, I made a plugin skeleton, Introspection [8], and rigged one of its functions so that it could be manually called by sending a GET request to a background web server it started. This was done because at the moment, penguin doesn’t have a built-in method of manually invoking plugin functions for testing. I needed manual invocation of this function because it takes time for systemd to startup, so hooking onto any other event would cause this function to be called at the wrong time and possibly multiple times.

```python
class Instrospection(Plugin):
    def __init__(self):
        print("Introspection plugin initialized.")
        sys.stdout.flush()

        # 1) Start a thread on __init__ that runs a TCP listener
        self._server_thread = threading.Thread(
            target=self._start_tcp_listener,
            name="InstrospectionTCP",
            daemon=True,
        )
        self._server_thread.start()

    # 2) TCP listener on port 9999 speaking a minimal HTTP/1.1 protocol
    def _start_tcp_listener(self):
        host = "0.0.0.0"
        port = 9999

        srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        srv.bind((host, port))
        srv.listen(5)
        print(f"[Instrospection] TCP listener started on {host}:{port}",
flush=True)

        while True:
            conn, addr = srv.accept()
            # Handle each client in its own tiny handler thread
            threading.Thread(
                target=self._handle_client, args=(conn, addr), daemon=True
            ).start()

    def _handle_client(self, conn: socket.socket, addr):
        conn.settimeout(10)
        # Read until end of HTTP headers (very small/for demo)
        data = b""
        while True:
            chunk = conn.recv(4096)
            if not chunk:
                break
            data += chunk
            if b"\r\n\r\n" in data:
                break

        # 3) On any request, call dostuff()
        result = self.dostuff()
        if result is None:
            result = "OK"
        body_bytes = result.encode("utf-8", errors="replace")

        response = (
            b"HTTP/1.1 200 OK\r\n"
            b"Content-Type: text/plain; charset=utf-8\r\n"
            + f"Content-Length: {len(body_bytes)}\r\n".encode()
            + b"Connection: close\r\n"
            b"\r\n"
            + body_bytes
        )
        conn.sendall(response)

    # Removed unused args; now returns text so the HTTP handler can reply
with it
    def dostuff(self) -> str:
        lines = ["Reading from target system"]
        proc = subprocess.Popen(
            ["python3", "/igloo_static/guesthopper/guest_cmd.py", "cat",
"/etc/passwd"],
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True,
        )
        stdout, stderr = proc.communicate()

        if stdout:
            lines.append(stdout)
        if stderr:
            lines.append(stderr)

        out = "\n".join(lines)
        print(out, flush=True)  # still log to stdout as before
        return out
```

The manually called function `dostuff` prints out /etc/passwd as a proof of concept for docker introspection within the rehosted system through sending a shell command. My thought process was that if it could read /etc/passwd after the system has booted, it would have no problem reading any files needed for docker introspection. The shell command method was chosen from other methods built into penguin (portal file transfer, direct file-reading helpers, and kernel-level system calls) but due to this function being called asynchronously to the rest of penguin, these did not work as intended.

Basic Docker Introspection

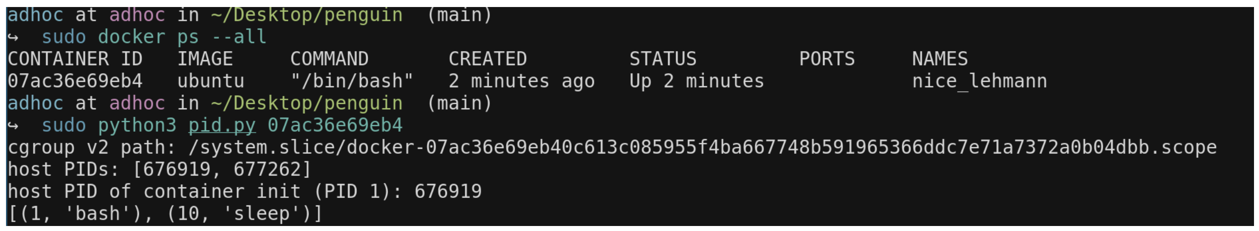
With this python plugin, I hypothetically only needed to figure out which files to read from the file system to do docker introspection. I did what my advisor recommended, which was to list all PIDs as seen within the docker container.

First, I looked through `/proc/<pid>/ns/pid`, where <pid> was the PID of the docker command that started the container. By reading this symlink and looking at `/proc`, I was able to get a list of all the processes within the container as a list of PIDs as seen within that container.
A list of PIDs isn't very helpful, so I got more information about what each PID was by looking at `/proc/<pid>/status`, where <pid> was the PID of each program after translating it back to the PID as seen on the host operating system. Two fields are what I looked for:

- `Name:` the process name
- `NSpid:` a list of PIDs showing how this process is numbered in each nested PID namespace. The last value in this line corresponds to the PID as seen inside the container.

To tie this back to a specific Docker container ID, I walk `/proc/*/cgroup` and search for cgroup v2 entries of the form `0::/…` that contain the container's ID in the path. This reveals the container's cgroup v2 path as mounted under `/sys/fs/cgroup`. Once I have that path, I recursively traverse the corresponding directory tree in `/sys/fs/cgroup` and read every `cgroup.procs` file. Each `cgroup.procs` file contains a list of host PIDs attached to that cgroup; aggregating these across the subtree yields the complete set of host PIDs associated with the container's cgroup hierarchy.

All in all, the python script I wrote (pid.py in [8]) returns the following output when done on a stock ubuntu image running the sleep command:



```
adhoc at adhoc in ~/Desktop/penguin  (main)
↪  sudo docker ps --all
CONTAINER ID   IMAGE     COMMAND       CREATED         STATUS         PORTS     NAMES
07ac36e69eb4   ubuntu    "/bin/bash"   2 minutes ago   Up 2 minutes             nice_lehmann
adhoc at adhoc in ~/Desktop/penguin  (main)
↪  sudo python3 pid.py 07ac36e69eb4
cgroup v2 path: /system.slice/docker-07ac36e69eb40c613c085955f4ba667748b591965366ddc7e71a7372a0b04dbb.scope
host PIDs: [676919, 677262]
host PID of container init (PID 1): 676919
[(1, 'bash'), (10, 'sleep')]
```

Now all that was needed to finalize this UROP was to test out the introspection with my plugin on a embedded system with docker already installed.

Case Study #1: BalenaOS

BalenaOS is a minimal Linux-based operating system designed specifically to run Docker containers on embedded devices [6]. It emphasizes reliability, multi-device support, and an update workflow tailored to IoT deployments. [6]

BalenaOS was appealing as an initial target to test the plugin on because:

- It is widely used in production IoT devices. [6]
- Docker is a first-class citizen in its design. [6]
- If rehosting succeeded, the system would likely start several containers out of the box.

I experimented with several BalenaOS images and eventually focused on the NVIDIA Jetson Xavier image, which booted the furthest in Penguin. However, introspection quickly revealed problems:

- Many docker state directories that exist on a typical Linux host, such as `/var/lib/docker`, were missing
- The docker client could not connect to the daemon
- Systemd itself appeared to be nonfunctional; core services were not starting correctly

From these and looking through the boot logs, I concluded that the Docker daemon never started in the rehosted BalenaOS instance. This is consistent with current limitations of Penguin in that systemd-heavy firmware requires additional modeling to boot correctly, and missing cgroup configuration and device models cause it to fail.

Because the containers never started, there was no Docker state to introspect. Despite that, this case study showed that penguin's lack of systemd support is gatekeeping containerized firmware.

Case Study #2: HomeAssistant

For a second target, I followed my advisor's original recommendation of attempting to rehost the Home Assitant Operating System. HAOS is another OS optimized for docker, specifically for hosting the Home Assistant Supervisor docker container [7]. This time the workflow was more complex:

1. Install HAOS in a virtual machine.
2. Ensure that docker works, then pull a simple Ubuntu container.
3. Snapshot the entire disk.
4. Run fw2tar to extract the filesystem.
5. Configure penguin to boot the extracted system

This time however, the system did not reach a point where Linux could god. Instead, Penguin failed early in the boot process due to missing critical system files no matter what modifications I made to the configuration file. After some investigation, the root cause was that HAOS was installed with multiple partitions, separating the boot, root, data, and other components into separate partitions.

fw2tar currently assumes that the entire Linux system resides in a single root filesystem, and does no dynamic mounting to assemble it into a single filesystem if it is split into multiple partitions. As a result, fw2tar only extracted the incomplete system partition and Penguin of course failed on boot.

I considered two potential workarounds for this:

1. Flatten HAOS into a single partition. This would require heavy modifications and would be hard to replicate for future contributors.
2. Build a custom firmware using Gentoo with a simple init system (non-systemd). This would be a fully controlled firmware that would be tailored for Penguin's and fw2tar's current limitations, and would allow Docker introspection to be test as a proof of concept easily.

Given time constraints and complexity of both options, especially the long Gentoo installation process, I was unable to complete either path during the rest of the UROP. However, this case study clearly highlighted the need for better multi-partition support in the rehosting pipeline.

Current and Future Challenges

There are currently two technical challenges that must be addressed before Docker introspection can be evaluated on real firmware at scale in penguin.

The first challenge is supporting systemd-based boots. Many containerized firmware images use systemd to manage both core services and the Docker daemon. Penguin has support for these systems, but only with expert knowledge and advanced configuration of the underlying firmware and penguin tooling. Because of this, it is hard to get dockerd to run in these environments to get any container running.

The second challenge is that both fw2tar and penguin are built on the assumption that the firmware is built on a single partition. Some pieces of firmware like the Raspberry Pi and HAOS have installations that are built on having multiple different partitions, limiting the use of penguin without expert knowledge.

Despite these two challenges, docker introspection can still be developed in penguin if a reproducible, stable single-partition firmware image can be acquired that uses the init system to run docker. A good choice for this would be to make a highly tailored Gentoo system with these qualities, given how reproducible and customizable the distribution is.

Once a running image is available, the plugin can be extended beyond the current proof‑of‑concept to:

- Mapping container IDs to:
    - image IDs,
    - command lines,
    - and associated cgroups/namespaces.
- Compute simple health metrics such as “number of containers in running state” or “containers that exited with non‑zero status.”
- Export results as structured JSON and integrate them into Penguin’s existing logging or visualization tools.
- Optionally, compare the observed container set against an expected configuration to flag missing or extra containers.

Longer‑term, it would be interesting to draw inspiration from Sysdig by correlating container state with system calls or network activity, but that likely requires additional instrumentation. (USENIX)

Conclusion

This UROP explored how to introspect Docker containers inside firmware rehosted with Penguin. I designed a Python plugin that can be triggered on demand to collect cgroup and Docker metadata from a guest and report it to the host. Experiments with BalenaOS and Home Assistant OS revealed that, in their current form, these images are challenging to rehost: BalenaOS suffers from systemd-related boot issues that prevent dockerd from starting, while Home Assistant OS uses a multi-partition layout that conflicts with Penguin's single-partition assumption.

Although the project did not reach fully automated container introspection on a production firmware target, it clarified the core obstacles and laid out a roadmap for future work: build a minimal Docker testbed, extend the plugin, and improve Penguin's handling of systemd and multi-partition images. Addressing these challenges will make it possible for IGLOO to analyze containerized firmware with the same fidelity it currently enjoys for simpler systems.

Sources

[1] bq. penguin [GitHub repository]. GitHub.
Available at: https://github.com/bq/penguin

[2] Linux Firmware Rehosting – Introduction [online module]. pwn.college.
Available at: https://pwn.college/rehosting~fbf8a0a3/module-intro/

[3] Docker, Inc. dockerd [CLI reference documentation]. Docker Docs.
Available at: https://docs.docker.com/reference/cli/dockerd/

[4] MrDevSecOps. "Docker Namespace Vs Cgroup: Namespace and Cgroup." Medium, 22 Oct 2021.
Available at: https://medium.com/@mrdevsecops/namespace-vs-cgroup-60c832c6b8c8

[5] Docker. "Cgroups, namespaces, and beyond: what are containers made from?" YouTube video.
Available at: https://www.youtube.com/watch?v=sK5i-N34im8

[6] Balena. Run Docker containers on embedded devices – balenaOS [product page].
Available at: https://www.balena.io/os

[7] Home Assistant. Home Assistant – Awaken your home [project homepage].
Available at: https://www.home-assistant.io/

[8] penguin branch [GitHub repository]. GitHub.
Available at: https://github.com/Desperationis/penguin