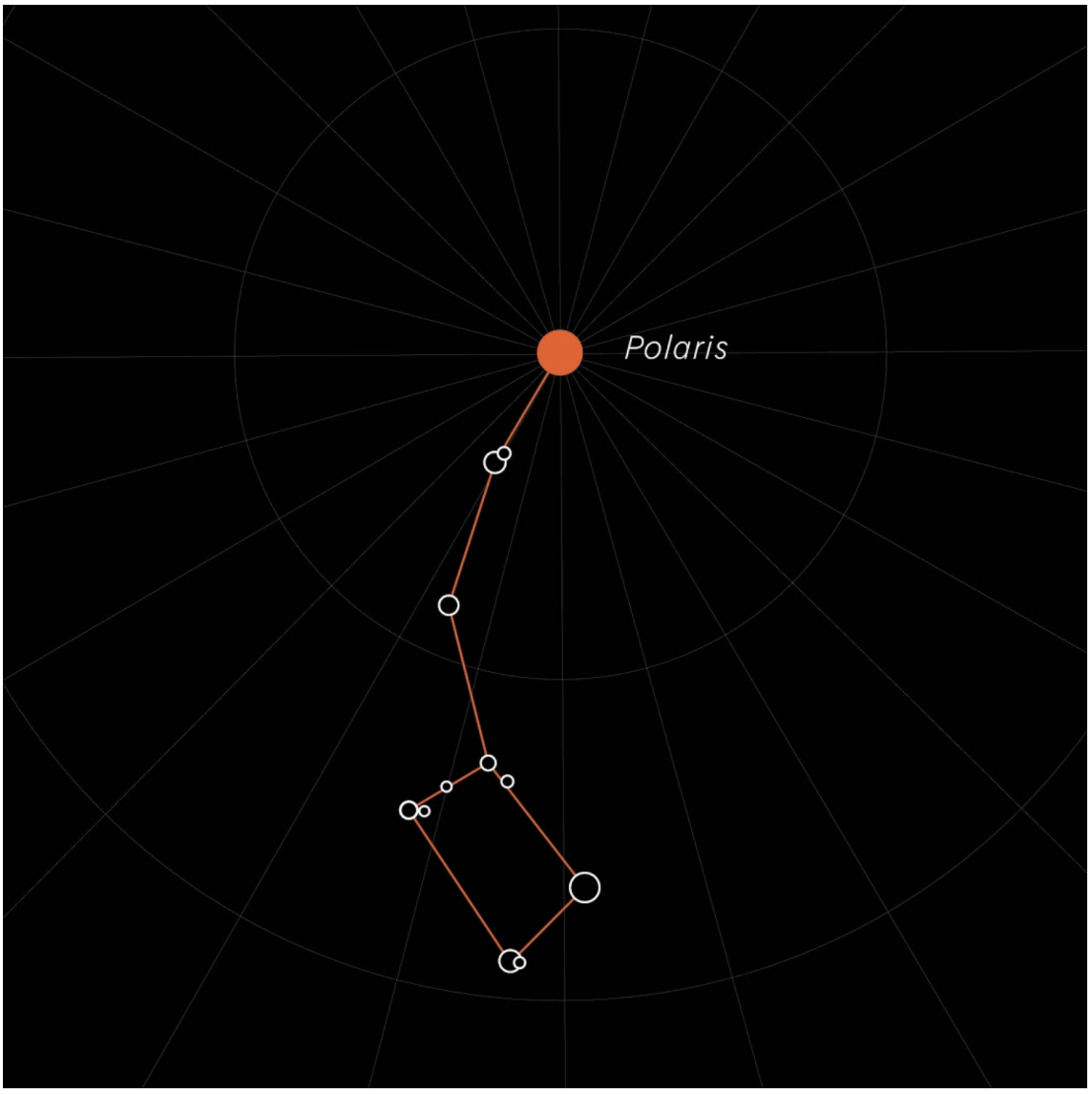
Northstar

Embedded Container Runtime



Authors
oliver.mueller@esrlabs.com
florian.bezold@esrlabs.com
peter.lawthers@esrlabs.com
felix.obenhuber@esrlabs.com

Version: 0.5.0

Table of Contents

List of Figures

Abstract

The trend in the automotive industry is to build more and more powerful onboard embedded systems that control parts of the vehicle. With increasingly powerful hardware more software features need to be tightly integrated. So a growing number of applications now need to share the same limited hardware resources. And often some of those applications are developed by third parties and thus absolutely need to be contained.

This implies a number of new challenges for building systems:

- **Robustness**
  guard against malfunctioning applications
- **Security**
  potentially malicious application must not be allowed to do harm
- **Provisioning and Update**
  secure way to install and update applications
- **Fast Startup**
  booting up the system needs to be fast

In other IT areas, those challenges are not new at all. What makes them unique for embedded vehicle systems is the absolute need for robustness and resource efficiency in restricted environments, as well as the need to start-up in a minimum amount of time.

As a solution to those challenges, we created a purely embedded version of a container runtime system that we call Northstar.

Northstar is designed as an edge platform that meets the special requirements of these systems. At its core, it makes extensive use of sandboxing to isolate applications from the rest of the system while at the same time orchestrating efficient startup and secure update scenarios. Such sandboxed applications run inside Northstar-containers and only rely on system services provided by the Northstar-platform. We select and use similar sandboxing techniques as are found in Docker and other containerization approaches to reach maximum isolation. To build the most efficient and robust solution, Northstar is completely developed in Rust, a language designed to afford the performance of C++ while at the same

time guaranteeing memory safety. The project is developed as an open source project under the Apache 2 license and can be found at github (https://github.com/esrlabs/northstar).

System design and complexity

Before we start it's important to set define some terms used in this document:

- **Northstar image**
  This is a deployable unit that can contain data and applications. Such an image can be executed in a sandboxed environment by the North runtime. Note that this is not an OCI compliant image! The format of Northstar images is much lighter weight.

- **Northstar Runtime**
  Controls the startup, shutdown, and update of the individual Northstar containers. It provides a sandbox environment.

- **Northstar container**
  A running instance of a Northstar image managed by the North runtime is called a container. Running on top of a Linux system, each container is executed as a supervised process.

- **Application**
  A sandboxed application deployed in a Northstar image, it is executed within the context of a container.

One of the primary principles of Northstar is simplicity. Northstar is explicitly designed for embedded environments, with design decisions always taking into account the limited resources of the platform. Each application that the North runtime is running is packaged in an individual, read-only image. That means it is self-contained and applications can always be updated individually.

While Northstar is polyglot and can support applications written in any language (e.g.. Java or C#), the cost of the additional needed virtual machines (JVM, CLR, etc.) can be substantial regarding CPU usage, RAM usage and startup time. Therefore the preferred approach for Northstar applications uses statically linked

binaries. That could be a pure native version built using Rust, C/C++, etc. or something like Go and Java (via GraalVM native images).

Northstar does not use an OCI-based container format or runtime such as Docker for containerization, which was designed for backend application deployment and contains complexity to support many features irrelevant for embedded devices. For maximum efficiency and security, the Northstar runtime and image format supports just the features required for this kind of applications.

The Northstar runtime monitors the state of all running applications. In the event of an application failure, Northstar reacts with a configured action for the specific container (report, stop, restart, quarantine, etc.).

Northstar Architecture

The Northstar system consists of several components. The base of the system is the lightweight Northstar daemon, a system service started and monitored by the init system of the OS, taking care of starting and monitoring containers. Furthermore, it implements strategies for error reporting and handling. The daemon starts, stops and supervises all running containers.

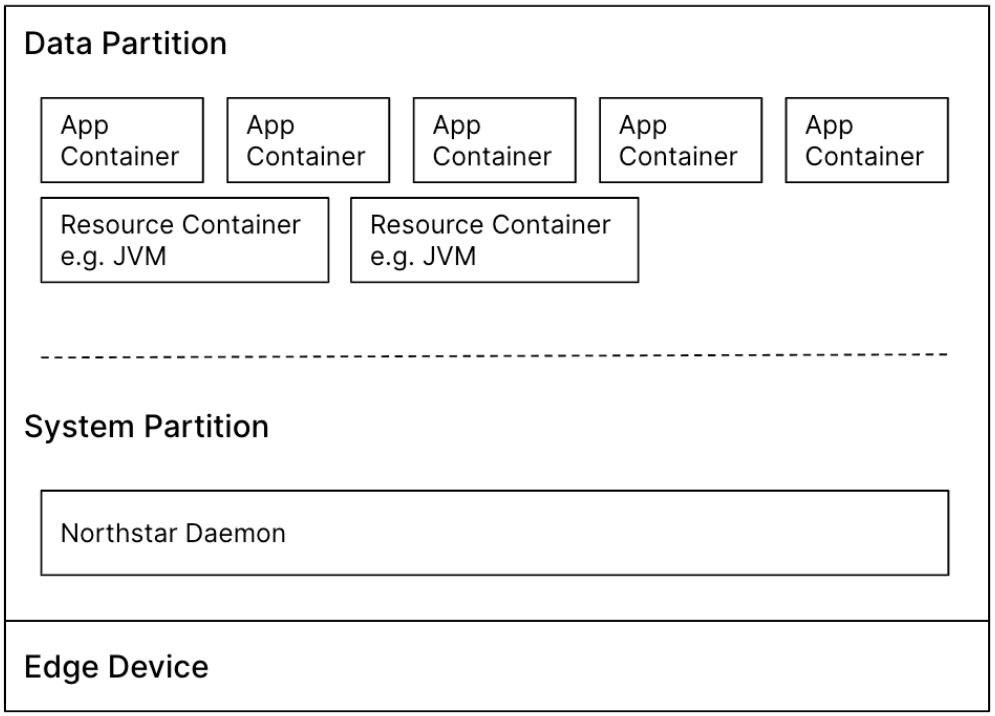


Figure 1 - Northstar Architecture

There are two different kinds of containers. Most basic are the resource containers. They can include libraries and resources needed by multiple application containers. This could for example be a C# runtime. Resource containers are optional and only should be used if there are resources that need to be shared among applications in order to reduce their memory footprint. This can be compared to docker overlays.

The primary container kind are **application containers** that can be securely downloaded and installed into a running system. Application containers only depend on resource containers, they should not have dependencies on each other. It's possible to enable application containers to communicate with each other if a suitable communication mechanism is provided by the operator. The number of containers that can be run by Northstar is only limited by system resources.

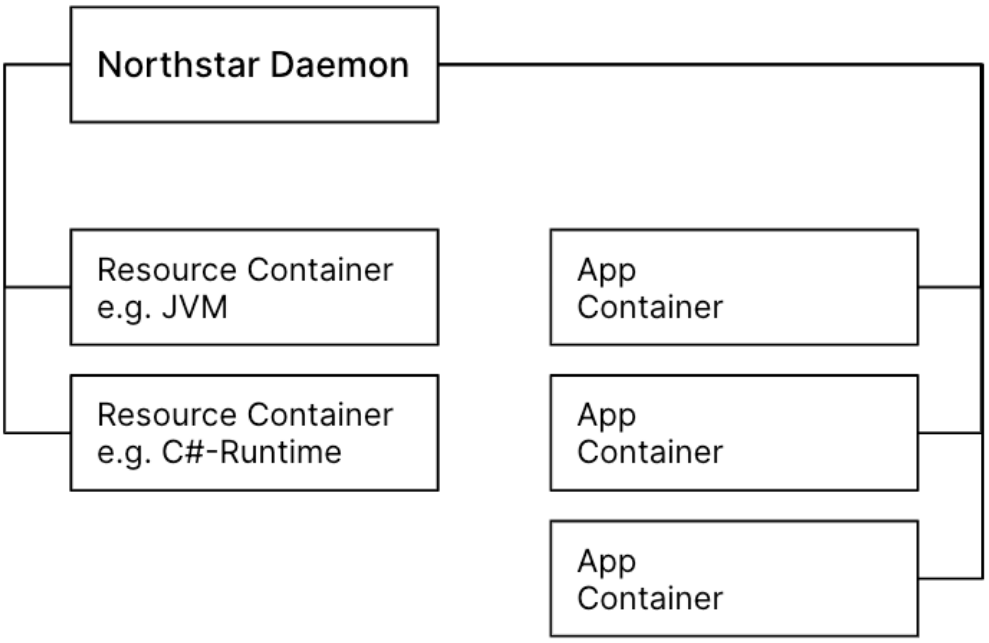


Figure 2 - Container Types

Both resource and application containers use the Northstar “NPK” image format, which is adapted from the Android **APEX** container format (https://source.android.com/devices/tech/ota/apex) as recently introduced by Google.

It contains a read-only filesystem image, a manifest describing the container and its runtime environment as well as a cryptographic signature of these two parts. A *NPK* container image is a single self-contained file, simplifying handling inside build systems, backend repositories and on the embedded device itself.

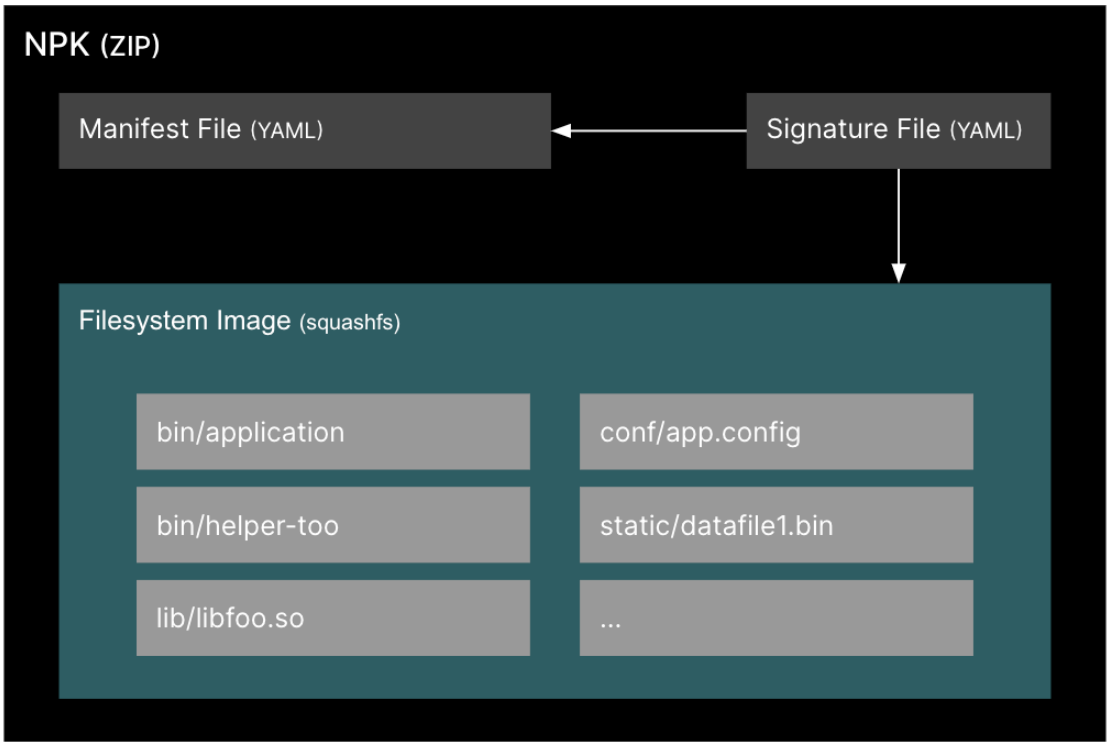


Figure 3 - Northstar NPK package

For runtime data, each application is provided with a writable directory inside of its namespace by the runtime, while the actual application image itself remains immutable. This data directory is kept when the container is updated, and deleted by the runtime if a container is uninstalled.

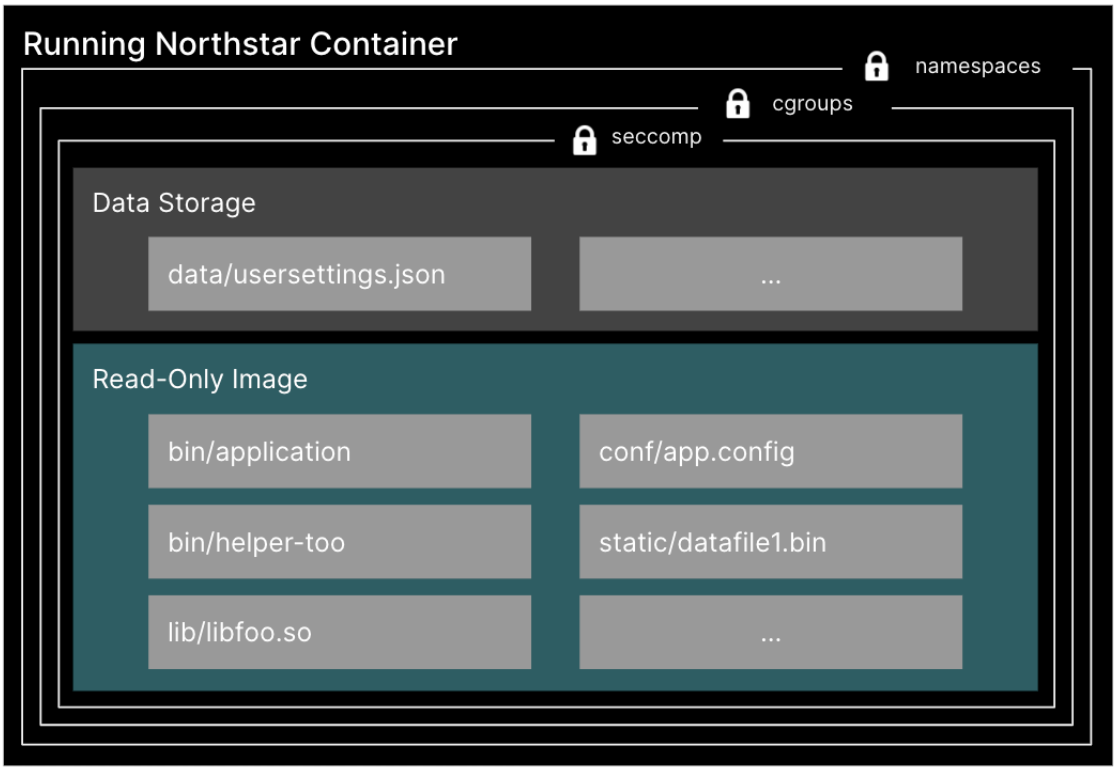


Figure 4 - Running Northstar Container

Security

Northstar provides a secure environment where unauthorized applications are prevented from executing. The integrity of applications is crucial. All updates that are downloaded are therefore verified and authenticated. Before any container is launched, the signature is again verified to ensure only valid applications are executed (see verified boot chapter).

In Northstar, there is no system management daemon running with root permissions. The runtime is a non-root service that monitors the state of all containers, optionally starting or terminating as needed, using just the minimum necessary set of Linux capabilities to set up the container environment.

Sandboxing

Sandboxing is at the heart of Northstar. Each Northstar container runs in a well-defined and isolated sandbox, restricting the damage that a compromised or malfunctioning application could inflict on the rest of the system.

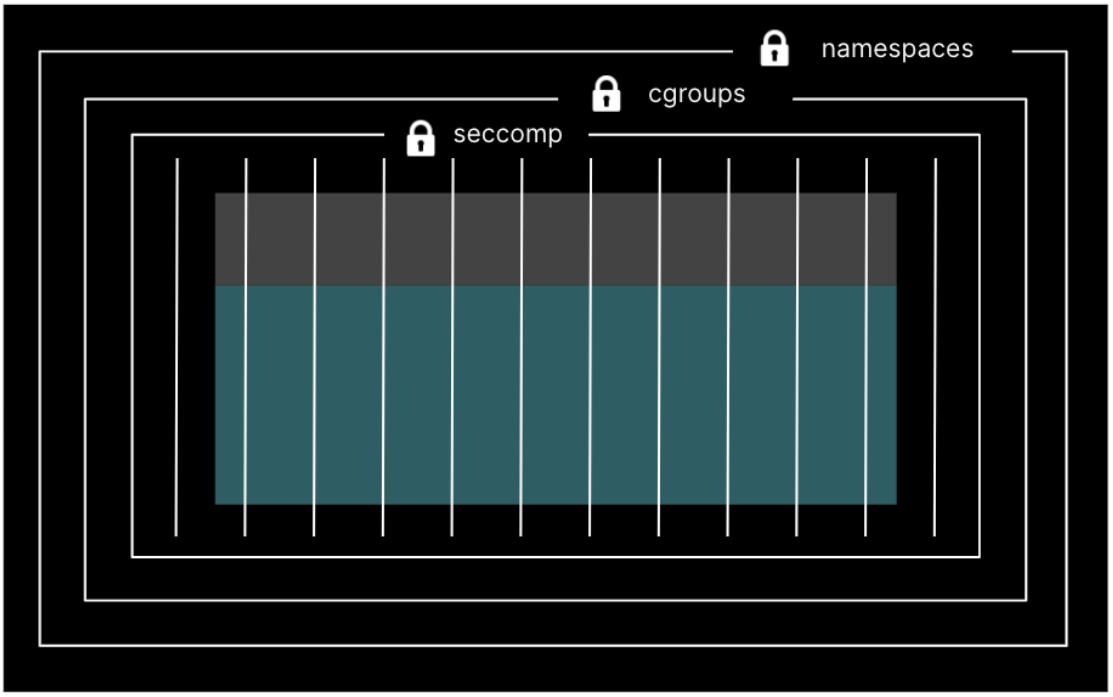


Figure 5 - Sandboxed Container

Each application has different sandboxing requirements. All parameters for sandboxing are specified in the manifest of each container. The Northstar runtime configures the sandbox according to the manifest specification. See Appendix I: Northstar Manifest Example.

There are numerous sandboxing features. To illustrate the concept we make a few simple examples:

- Each container gets a memory budget. Exceeding the budget indicates a memory leak and will result in a kill and restart of the container.
- By using network namespaces, a compromised application is not able to disrupt network communications in any other namespace (including the default or root namespace).

- Sandboxes for each Northstar container do not execute with root permissions; they are mapped to a non-root uid/gid.

The container isolation that Docker provides is taken as the base requirement for Northstar sandboxing. Applications within Northstar containers are restricted via the Linux `seccomp` mechanism to a limited set of system calls, which inhibits what a potentially compromised application can execute, either using a generic default profile that disables system management syscalls, or a customized profile derived from the specific application. Applications are further prevented from gaining additional permissions during execution.

In Linux, a sandbox is a combination of several kernel isolation and security features. For this, Northstar leverages the *minijail* technology developed by Google for Android and Chromium. An overview of the features and functionality of *minijail* can be found in Appendix A: Minijail.

Verified Container Boot

In order to prevent an attack by tampering with an already downloaded and installed application, each container image is re-verified at startup to make sure it contains exactly the data which was signed by the build system. This means all files within the container - binaries as well as data - need to be checked, even after the container was started.

DM-Verity

The verified boot is based on the device-mapper-verity (*dm-verity*) Linux kernel feature which was introduced to Android and Chrome OS to prevent attacks on those systems. To use this, a tree of cryptographic hashes is appended to the NPK's filesystem image at build time. The root hash of this tree itself is verified by the container signature. Since *dm-verity* checks the hashes of filesystem blocks only when they are read from flash, the container setup can be done in fast, constant time, and containers that do not access all their data at startup can start faster. If a hash is incorrect, the I/O operation as seen by the application fails. Additionally, the runtime checks for a hash failure and restarts the container in this situation.

Benchmarks

Benchmark Environment

For all the following benchmarks, we are relying on a dm-verity-backed squashfs filesystem. All benchmarks were conducted on an automotive 64-bit ARM dual-core system (Renesas R-Car M3).

Startup benchmark

A simple test container that runs a rust executable can be set up in well under 100 ms. The startup time can be split into the actual container startup that prepares the sandboxed environment plus the startup of the application running inside the container.

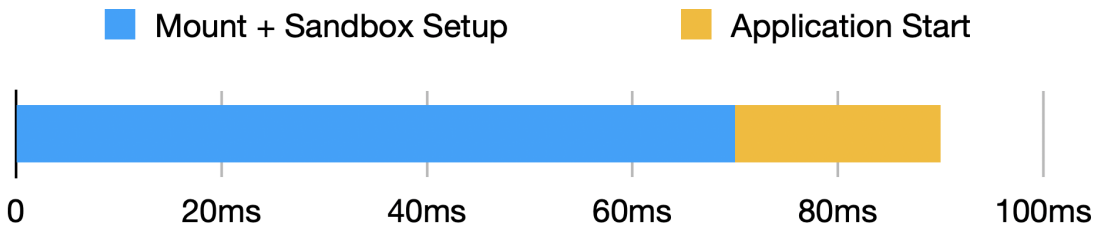


Figure 6 - Startup Benchmark

The container startup can be further analyzed, below we dig deeper into the blue section. In this section 2 tasks are completed:

1. Mounting + Signature Checking

To ensure the integrity of the container, signature checking needs to be done with every start of a container. We make use of dm-verity which even guarantees that no files were changed even after we started up.

The *mount*-part includes the loopback device setup, the verity setup, and the actual mounting. For the loopback setup section there already exists an optimization in the latest kernel version that will reduce this time even further.

2. Sandbox Setup

In order to run a container, the secure sandbox environment also needs to be set up completely before a container can be started. This means mainly the configuration of minijail. With this, one container is completely up and started after ~90 ms.

In the graph below you can see what happens in the ~70ms blue mount/setup-part.

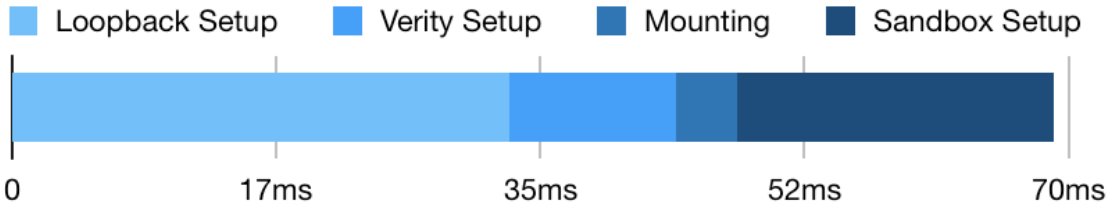


Figure 7 - Sandbox Setup

The major share of the *Sandbox Setup*-part consists of the setup of the remount of `/proc` used in the *chroot*-environment.

18-container MQTT-Benchmark

In an example scalability experiment with 18 containers, we measured the startup of all containers to the point where every container was completely started within its individual sandbox. The content of each container was a tiny rust application with an MQTT client.

To demonstrate full container functionality including a functioning network, we define a container to be started when the application within got connected to an MQTT broker. The complete startup (until every container executable was running) took ~4.0 seconds. The startup times for each individual container were affected by the increased system load.

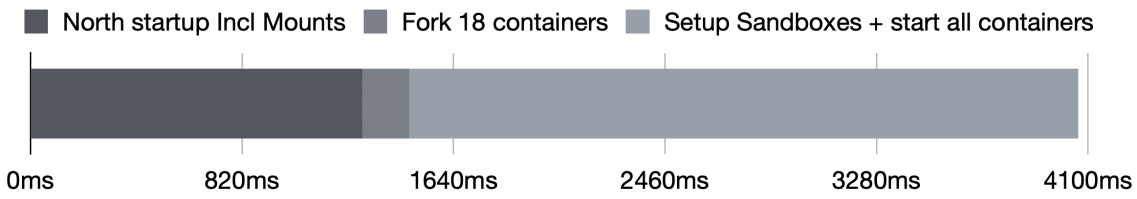


Figure 8 - MQTT Benchmark

Northstar vs Docker/OCI solutions

Compared to *Docker* or similar OCI-compliant container solutions, the Northstar design was intentionally focused on simplicity, to ensure reliability and maximize performance and security. This resulted in several notable differences:

Northstar uses a simpler execution and container lifecycle model: all containers are spawned and controlled by the runtime, which is itself expected to be started with system boot and running permanently. Also, while containers support standardized logging and communication through system services, there is no support to dynamically attach or detach from standard input/output of the processes. In contrast, *Docker* supports management daemon restarts while containers keep running as well as interactive I/O scenarios through a more complex shim process hierarchy, with an associated complexity and performance cost. The fact that Northstar containers do not outlive the termination of the runtime was additional motivation to write the implementation using the Rust language, which eliminates many classes of robustness and security errors while still producing performant native binaries.

OCI-compliant images are based on multiple filesystem layers which are managed and downloaded individually and potentially shared between many images. This reduces download traffic and storage space if many images are built upon the same, large base image, like a fully featured Linux distribution or large VM runtime. Since the target scenario for Northstar is running more lightweight containers in an embedded environment, the NPK image format is a simple self-contained file, which does not support sharing parts between images. This significantly simplifies image handling for both the runtime and the backend repositories and makes it possible to introduce powerful security mechanisms like *dm-verity* with little effort.

At runtime, OCI images are writable by default, creating an additional per-instance layer dynamically, whose lifecycle must be managed by the runtime as well. By design, Northstar images are always read-only and store their data in a specific part of the filesystem namespace instead, like a *Docker* volume. This ensures unintended write operations are prevented and identified.

Update and Remote Monitoring

The container-based architecture used in Northstar makes it possible and practical to update individual containers in a secure way. Out of the box, there is support for installing or updating components in a running system. It is easily possible to include a custom update check and download functionality that fits best for a customer use case. Since container images are built, distributed, and executed as a single file, the handling and management of images is simpler compared to the *Docker* world, where an image is a combination of several layers which are individually distributed and stored in a cache.

Monitoring the runtime

The Northstar runtime imposes crucial security restrictions on the managed containers. All containers are constantly monitored and runtime statistics can be tracked. If any container tries to misbehave or crashes, this is important information that needs to be recorded. The north runtime knows a lot about the state of the containers it is executing. Some of this knowledge can be very relevant to the operation and should be communicated to another component or even a backend.

To make this possible, northstar generates notification events for all interesting system events (e.g. an out of memory situation of a container, a container crash, ...) and propagates those events to a monitoring application that in turn can relate this information to a backend service. This allows for example to monitor application behavior in a whole fleet of vehicles.

Roadmap

While the possibility to use remote monitoring features is already baked into the Northstar runtime, there are a lot of possible use cases for using collected data, especially for security and stability reasons. This is an area we are currently evaluating.

To further improve security, micro-VMs such as Firecracker are promising, since they allow to further reduce the attack surface between container application and host system by relying on virtualization: each container is then running its own, minimal Linux kernel which communicates with the host kernel through a small set of virtual hardware interfaces provided through KVM. Since the isolation mechanisms (such as ***minijail***, *namespaces*, and *seccomp*) in Northstar are decoupled from process launching and management, Northstar can easily adapt to new technologies as they become stable. As NPK images contain a proper filesystem image already, this could be directly provided to the VM as a virtual disk image as well.

Summary

With the implementation of Northstar, we have created a secure runtime environment that meets the stringent demands of building complex embedded automotive systems. Limited computing resources are used economically and startup times are kept low while still meeting the critical security obligations. For the implementation, we made use of state-of-the-art isolation technologies that are also used heavily in Android and supported by all modern Linux kernel versions. That makes Northstar easily portable across a wide variety of embedded systems.

Appendix

Container integrity Checking

To ensure the integrity of the container, we compared two approaches:

- Completing a full signature check upfront
- and a *dm-verity* backed filesystem.

For this test, no additional sandboxing was applied. For completeness, we also used 2 different filesystems in the dm-verity setup.

With the *dm-verity* approach, there is no initial signature check needed since all signatures are verified on the fly. With the first approach, the startup overhead of the upfront checks is relatively small with 150 ms compared to roughly 110 ms for the dm-verity setup.

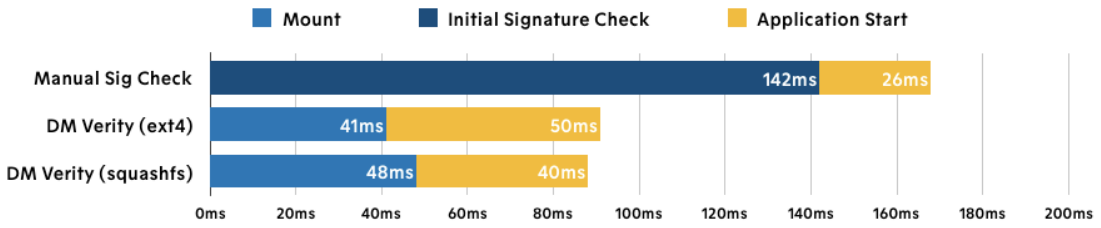


Figure 9 - Full Signature Check

The difference gets more pronounced when starting a complex container that e.g. contains a JVM. Here the image contains a lot more files that need to be verified at startup for the manual signature check.

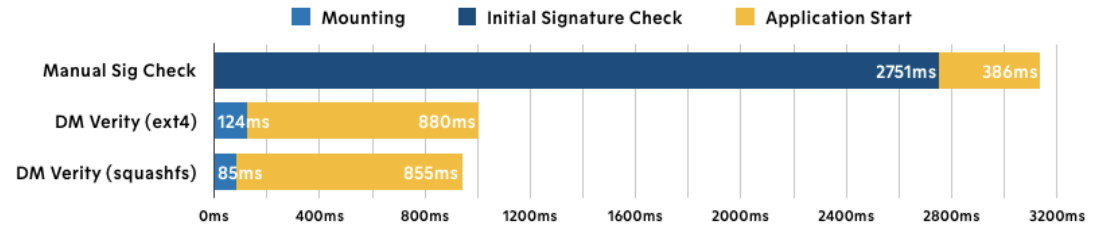


Figure 10 - DM-verity Backed Filesystem

Since there is no upfront cost for signature checking in the DM-verity case, the start of a small container with a simple application is similarly fast as the start of a more complex container that might be running a JVM.

Minijail

Minijail (https://android.googlesource.com/platform/external/minijail/) is used extensively on Chromium and Android to securely sandbox processes. Minijail provides very fine-grained control over the execution environment of a process, using features that are native to Android. Minijail ensures that a malicious process can not gain control over the system or interact in an unintended manner with other processes.

Minijail provides effective sandboxing of processes through Linux namespaces, `uid/gid` remapping, chrooting, capability(7) restrictions, and `seccomp`. In particular, minijail can mimic the process sandboxing of *Docker* by the use of:

- Network namespaces
- Pid namespaces
- UTS namespaces
- Mount namespaces
- IPC namespaces

Namespaces provide Linux kernel controlled visibility and accessibility of resources like processes, filesystems, interfaces, and devices. *Network namespaces* provide a robust way of isolation, as firewalling and routing from and to the namespace is configured outside of the namespace (and therefore outside of the jail), so an attacker can not see or modify these from inside the jail.

As with *Docker*, user namespaces are not typically used. User namespaces provide the ability to map a user ID in a namespace with a user ID outside of the namespace. Recent *Docker* versions provide this with the `userns-remap`.

Minijail provides extensive `seccomp` filtering hooks to restrict what system calls a process is allowed to perform. This provides very fine-grained granularity on a per-process basis of what explicitly a process is allowed to do.

With Northstar, and its explicit control over each and every application process, a system call trace can be generated that lists all the system calls used by the process, effectively locking it down. In comparison, the *Docker* seccomp facility is applied to all processes running inside a container; it is not intended to restrict system calls on a per-process basis.

In addition, the specific Linux **`capabilities`**(7) of a process can be specified, further restricting the actions and operations that the process can perform.

Firecracker

Firecracker (https://firecracker-microvm.github.io) is a virtual machine monitor that uses the Linux Kernel-based Virtual Machine (KVM) to create and manage lightweight virtual machines, called microVMs. These *microVMs* provide enhanced security and workload isolation over traditional VMs, while enabling the speed and resource efficiency of containers.

In addition to using user-space options such as namespaces and shared layers, Firecracker uses KVM to encapsulate every process or container within a virtual machine barrier. The microVMs run in an isolated, jailed environment, completely isolating each microVM from each other.

Northstar Manifest Example

Northstar manifests are defined in the YAML format. YAML files are human and machine-readable and can be versioned in version control systems because of their text form (unlike binary formats).

```yaml
name: hello
version: 0.0.2
init: /binary
args: [one, two]
env:
  HELLO: north
cgroups:
  mem:
    limit: 10000000
    action: RESPAWN(3)
  cpu:
    shares: 100
```

```
seccomp:
    fork: 1
    waitpid: 1
```

Table 1 - Northstar Manifest Example

Tooling

Creating images and managing containers is an important part of operating a Northstar system and is very important for a good development experience. ***sextant*** is a tool that supports all the usecases that are needed to create and maintain a northstar environment.

```
sextant -h
sextant 0.1.0
Northstar CLI

USAGE:
    sextant <SUBCOMMAND>

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

SUBCOMMANDS:
    gen-key    Generate a key pair
    help       Prints this message or the help of the given subcommand(s)
    inspect    Print information about a Northstar container
    pack       Pack Northstar containers
    unpack     Unpack Northstar containers
```

Version History

Version	Date	Modifier	Comment
0.1.0	03/27/2020	Oliver Mueller	initial version
0.2.0	04/01/2020	various	basic runtime description
0.3.0	04/15/2020	Oliver Mueller	added benchmarks
0.4.0	06/23/2020	Oliver Mueller	updated benchmarks
0.5.0	10/29/2020	Oliver Mueller	add npk descriptions, monitoring and sextant information

Table 2 – Version History