

NixNet: Reproducible Virtual Network Experiments

Benedikt Spies¹, Marcin Bosk¹, Paulo Mendes², Jörg Ott¹

¹TUM School of Computation, Information and Technology
Department of Computer Engineering
Chair of Connected Mobility

²Airbus Central Research & Technology
Munich, Germany

**Network experiments are difficult
to share and reproduce!**

Problem



- Underspecified dependencies
- Resources no longer available
 - package manager no longer has specific version



- Different build-time environment
 - different compiler or compile flags



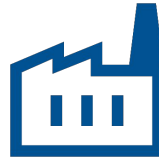
- Different run-time environment
 - different JIT or interpreter
 - different host machine OS or configuration
 - leftover state from previous experiments

**One should be able to build on
today's experiments,
tomorrow, as well as 10 years
from now!**

Why sharing and preservation is important



Academia



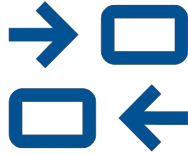
Industry



Teaching

- Building on prior work
- Interop
- Bug analysis
- Understanding & validate behavior
- Portable lab environments

Running real software in controlled environments



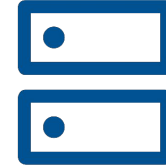
Simulated Testbeds

- deterministic
- scalable
- models behavior



Emulated Testbeds

- real software
- flexible



Hardware Testbeds

- real software
- real hardware
- expensive

State of the Art

Mininet

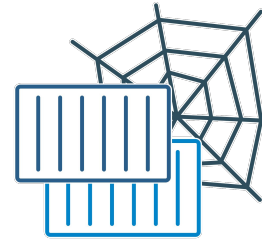
- side-effects
- no isolation
- no dependency management
- high abstraction
- error-prone



handwritten
Bash + tc-netem



CONTAINERlab



Containernet

- heavy-weight
- not reproducible from source
- slow startup



NixNet



Full dependency management



No manual setup



Hermetic build & execution



Run on single machine (Linux)



Single unprivileged command



No manual cleanup

How does it work?

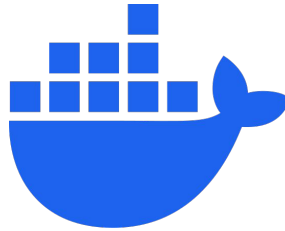
Dependency Management

Dependency management



Version Control
Git

- tracks source file history
- no build instructions
- no dependencies
- preserved long time (diffs)



OCI containers
Docker

- bundles all binary dependencies
- not reproducible from source
- no source tracking
- large image sizes
- frequently dropped from registries



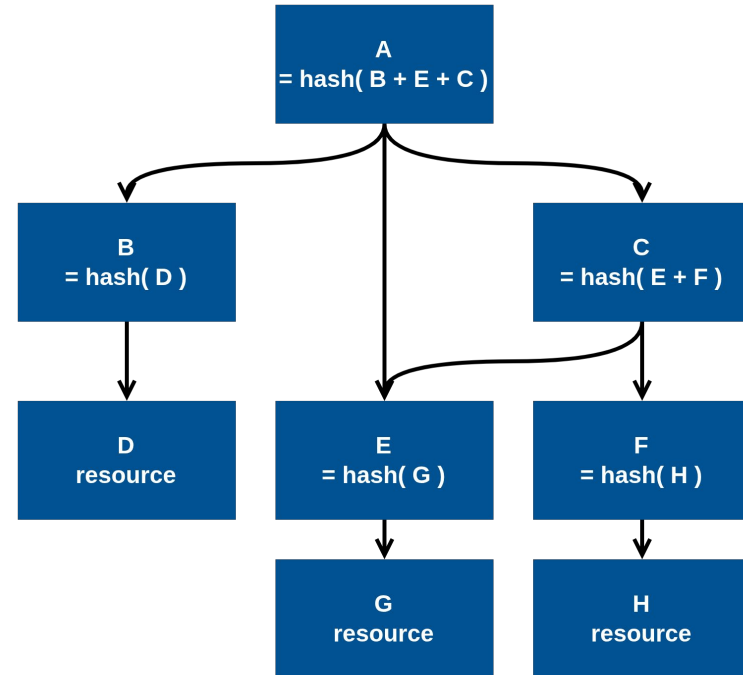
Binary package managers
APT

- manages binary dependencies
- no source dependencies
- outdated versions are dropped fast
- semantic versions

Merkle Graph

- graph of cryptographic hashes
- hash uniquely identifies node
- hash changes if any child changes

=> identify experiments by its cryptographic hash



Nix store

```

/nix/store
├── 7f3dbmm1dpbq20kxv4j85mvpkmkz0ns80-my-experiment
│   └── bin
│       └── my-experiment
├── 2hjsch59amjs3nbgh7ahcfzm2bfw18zi-bash-5.3p9
│   └── bin
│       └── bash
├── c9bn9jz0f89hhy5n1m7k4sv1b79i33r5-iproute2-6.18.0
│   └── bin
│       └── ip
├── ilk5qzvkdadj7lx58hfinfv17jmhr1q6-util-linux-2.41.3-bin
│   └── bin
│       └── unshare
├── gcqgrfnwq3pn0acbma9lwb1shl0l8a70-iperf-3.20
│   └── bin
│       └── iperf3
├── 56ncsa3ljzvc200n5a476d578y8d0pv-simple-starlink-ebpf
│   └── starlink.o
├── 1012111lmylczj1ihqn351af2kyp5x19-glibc-2.42-51
│   └── lib
│       └── libc.so.6

```

Dependencies

Realizations

Run-time closure
graph of run-time dependencies



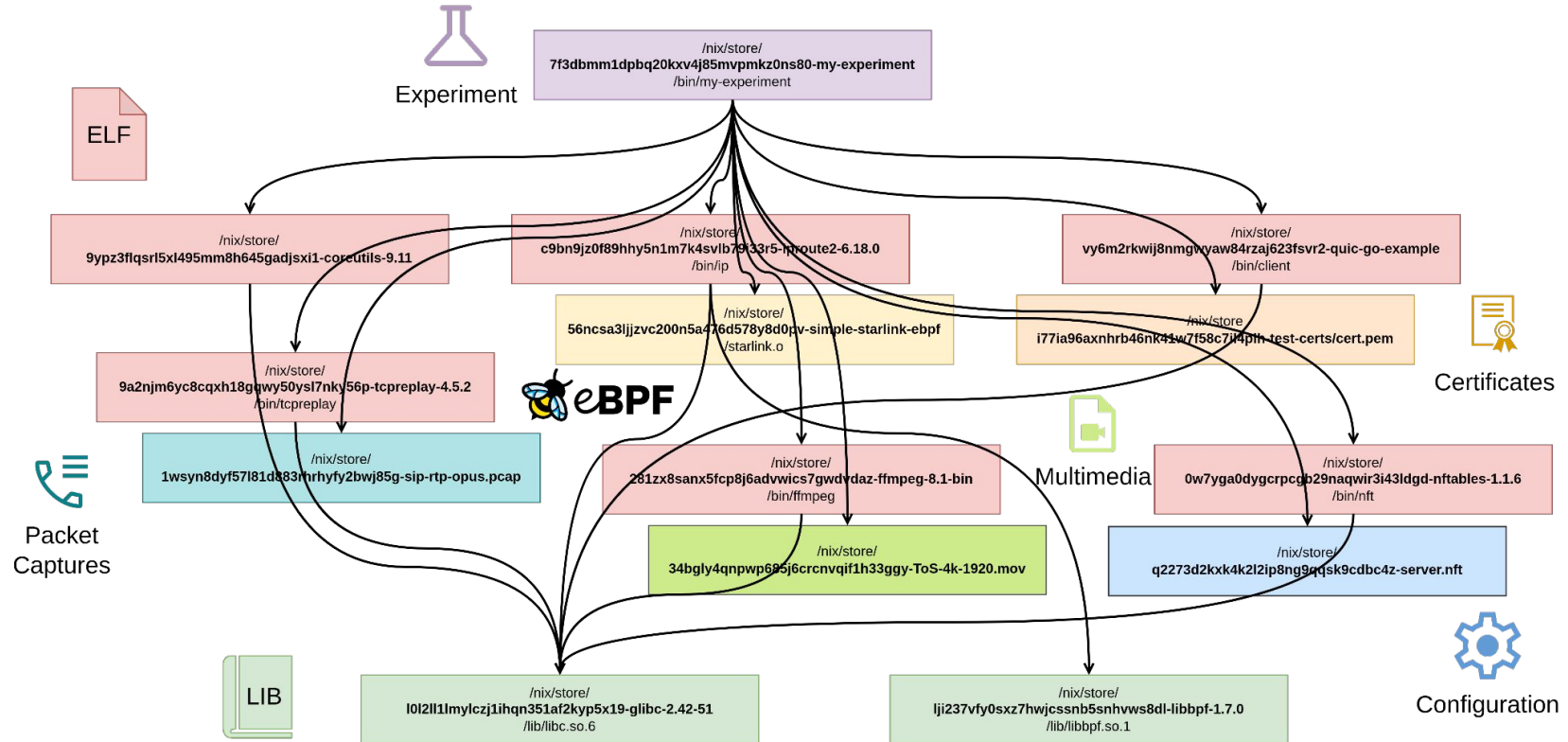
Build-time closure
graph of derivation (build recipes)

```

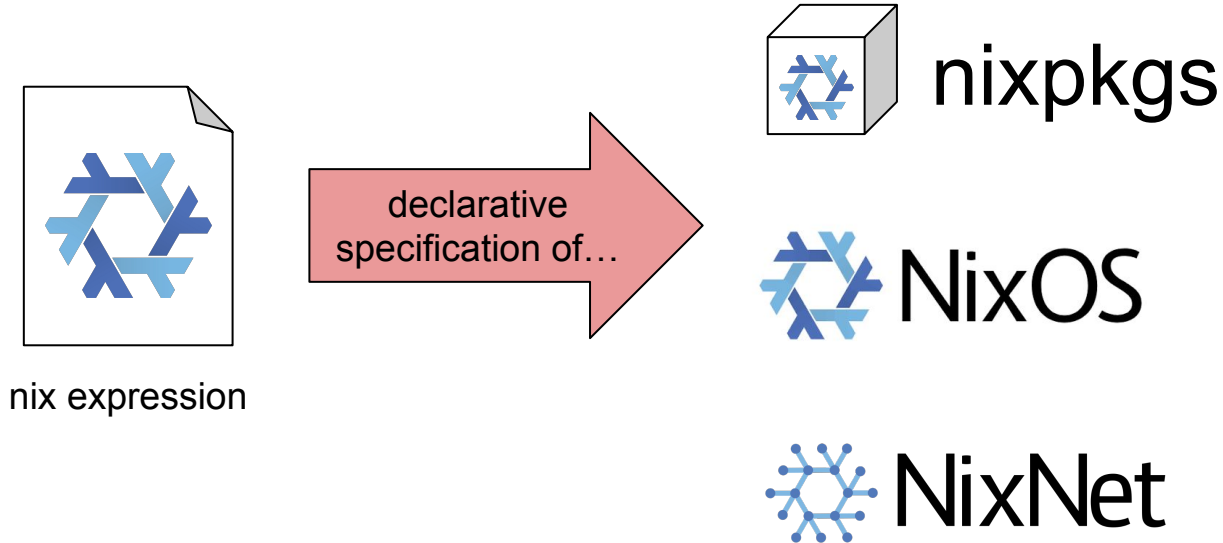
/nix/store
├── y0iwp7kds3jphy7vd3v619xvw8mv0chf-my-experiment.drv
├── kqv4h5cjq17nm4l64v9bbvpgz2sgax4-iperf-3.20.drv
├── s17kqafdpstrgbfygig72hgv04vr905yw-gcc-15.2.0.drv
├── b5v55bcrd56kq884s88k4c7yd0m98qxb-simple-starlink-ebpf.drv
├── dviq2hgv6jn6fr65shq1i3g5jamsxmzx-clang-wrapper-21.1.8.drv
├── sw87dnq97qby10dh22ij6zlnr16la7qg-libbbpf-1.6.3.drv
├── dvl9ms10kdc3wf0yr1y2c6k0b8dpbds-starlink.c
└── ...

```

NixNet run-time closure example



Leveraging the Nix build system



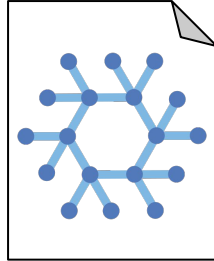
- nix builds are hermetic

Experiment declaration

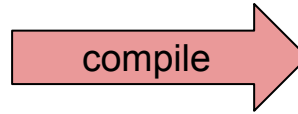
- Nix language
- json-like
- version controllable
- Schema borrowing options from NixOS
- nodes
 - scripts
- links
 - characteristics
 - tc-netem
- hooks

```
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-26.05";
    nixnet.url = "github:birneee/nixnet";
    starlink.url = "github:birneee/simple-starlink-ebpf";
  };
  outputs = inputs@{ nixpkgs, ... }:
    let
      system = "x86_64-linux";
      pkgs = nixpkgs.legacyPackages.${system};
      nixnet = inputs.nixnet.legacyPackages.${system};
      starlink = inputs.starlink.packages.${system}.default;
    in
      {
        packages.${system}.default = nixnet.mkExperiment {
          nodePackages = with pkgs; [ coreutils iperf3 ];
          nodes = {
            client = {
              networking.interfaces.eth0.ipv4.addresses =
                [{ address = "10.0.0.1"; prefixLength = 24; }];
              scripts.main = {
                exec = "sleep 0.1; iperf3 -c 10.0.0.2 -t 30";
                await = true;
              };
            };
            server = {
              networking.interfaces.eth0.ipv4.addresses =
                [{ address = "10.0.0.2"; prefixLength = 24; }];
              scripts.main.exec = "iperf3 -s";
            };
          };
          veths.eth0 = {
            netem.delayMs = 40;
            a.node = "client";
            b.node = "server";
          };
          postSetup = ''
            ip netns exec client ip link set dev \
              eth0 xdp obj ${starlink}/starlink.o sec xdp
            ip netns exec server ip link set dev \
              eth0 xdp obj ${starlink}/starlink.o sec xdp
          '';
        };
      };
}
```

Low abstraction



NixNet
flake.nix

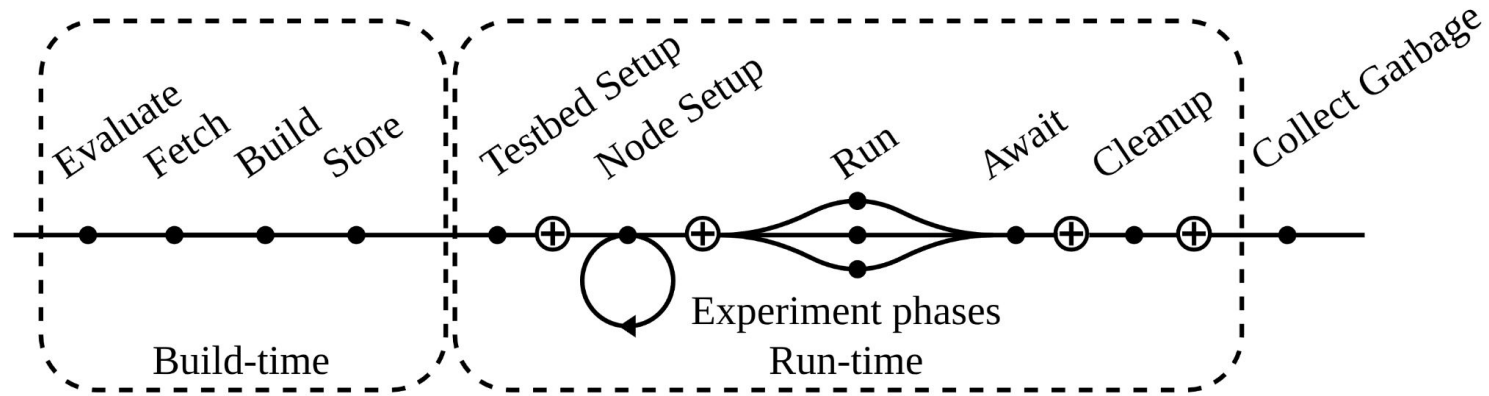


human-readable
Bash

- familiar terminal commands

Running experiments

NixNet timeline



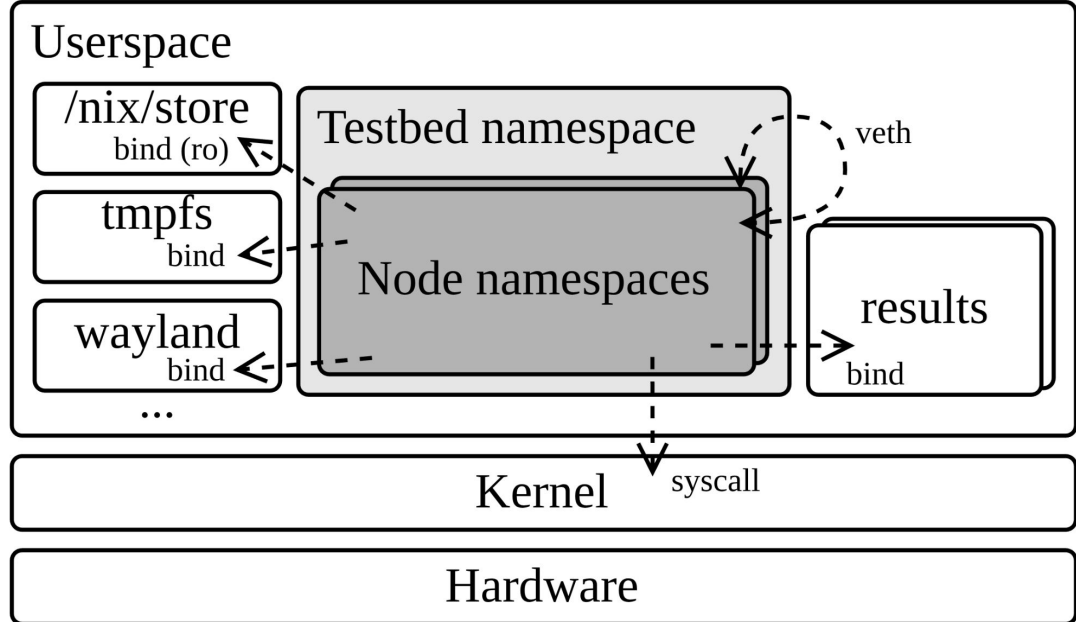
- isolated builds
- isolated runs

⊕ hook point

Run-time isolation

Multi-layer isolation

- strong isolation to prevent side-effects
- fully managed userspace
- Linux namespaces
 - user
 - net
 - mount
 - pid
 - ipc
 - uts



Example

```
$ nix run
testbed| workdir: out/42
testbed| network topology set up
client| main (PID 673) started
server| main (PID 674) started
testbed| PID 673 exited with 0
testbed| PID 674 killed
testbed| cleaning up...
```

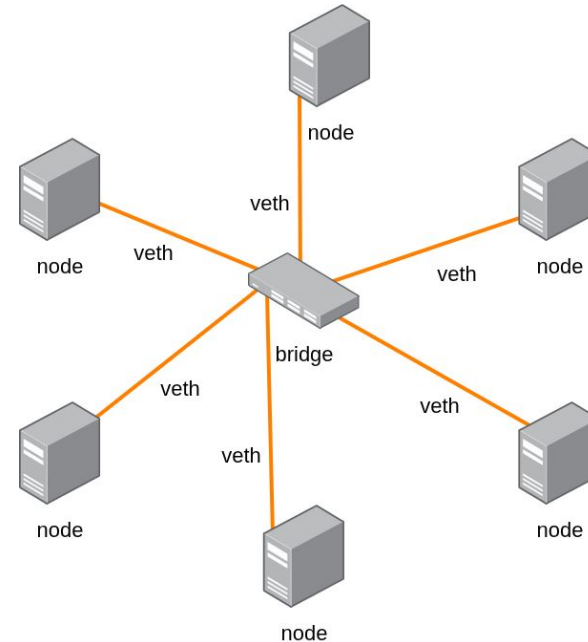
```
$ tree out/42
out/42
├── client
│   ├── flamegraph.svg
│   ├── perf.data
│   ├── stderr
│   └── stdout
├── run.json
└── server
    ├── flamegraph.svg
    ├── perf.data
    ├── stderr
    └── stdout
```

3 directories, 9 files

- nix also installs the pinned version of NixNet

Performance evaluation

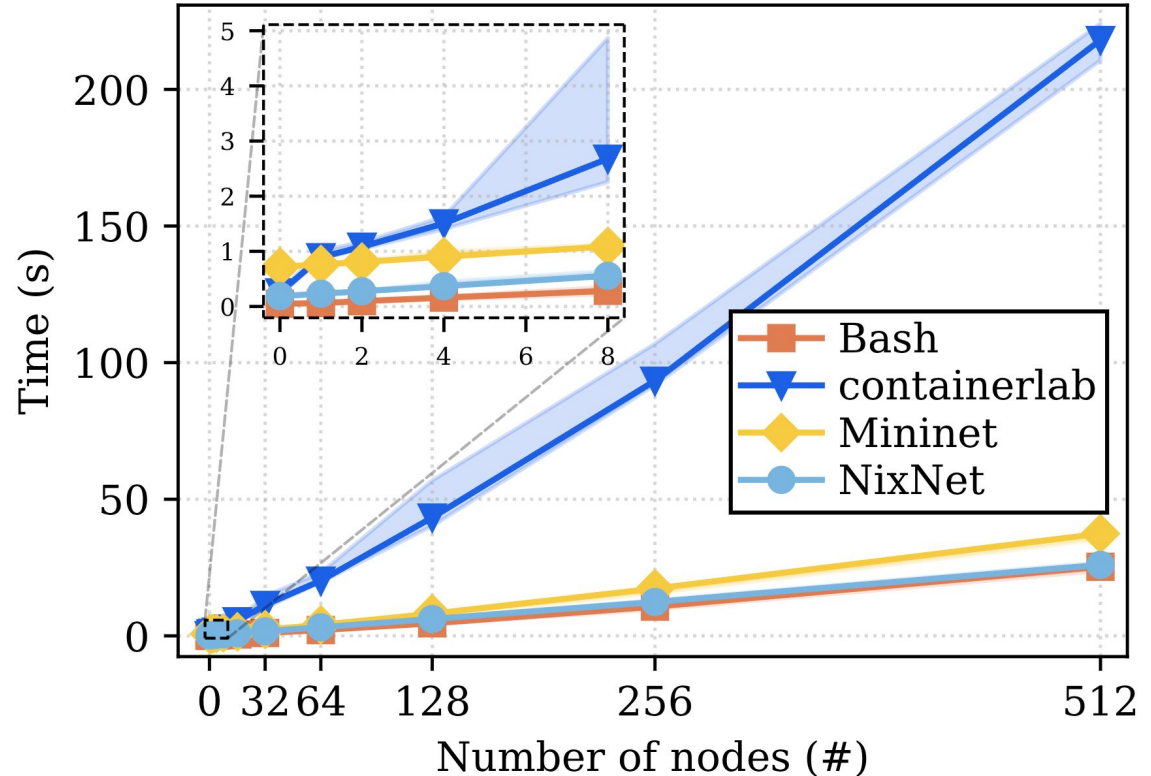
- repeat experiment runs
 - sweep parameter sets
 - re-setup testbed for every run
 - prevent inter-experiment leakage
-
- star-topology with n nodes
 - up to 512 nodes



Setup + cleanup time

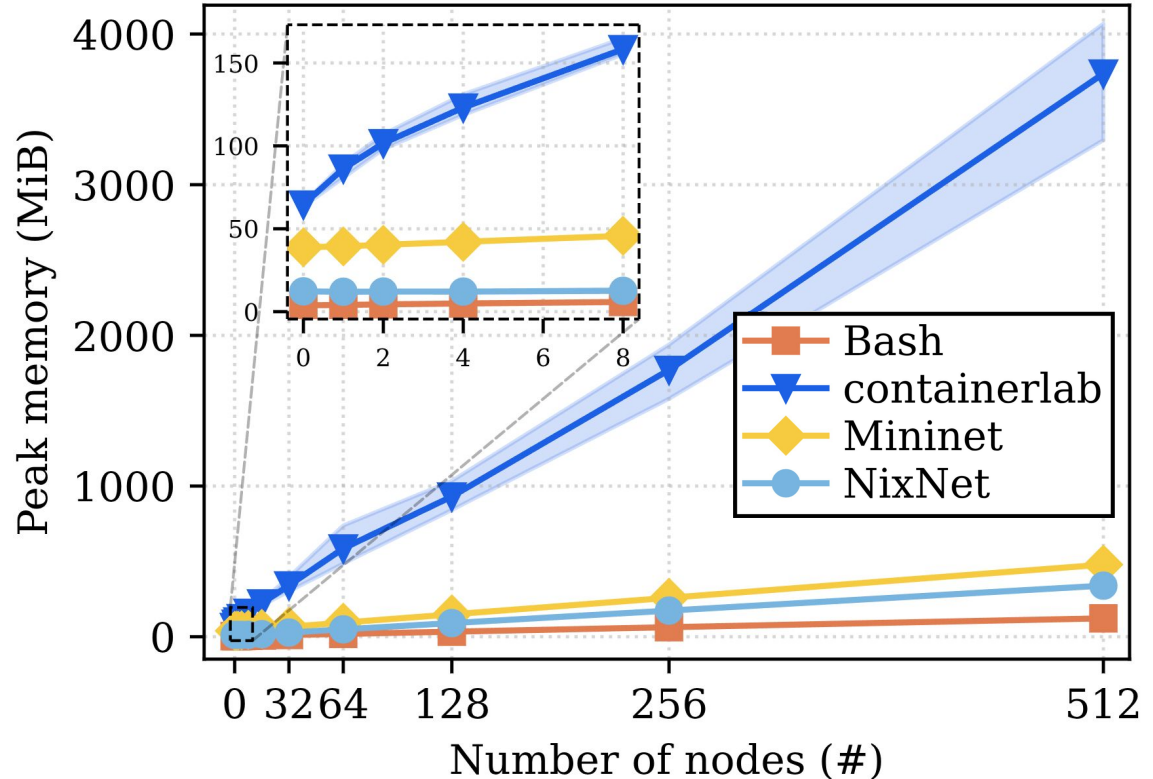
- starting fresh Docker containers takes time
- NixNet even outperforms Mininet (no isolation)
- e.g.:
 - 8 nodes
 - 100 runs

save 3.5 minutes



Peak memory

- Docker containers consume much more memory
- e.g.:
 - 128 nodes
 - clab ~1 GiB
 - NixNet < 100 MiB



Limitations & future work

- nodes share host CPU and memory resources
- pinned-kernel NixOS image
- Nix does not pin its own version
- Docker container integration
- Windows & macOS support
- compiled executable outperforms script with unshare/mount/nsenter processes

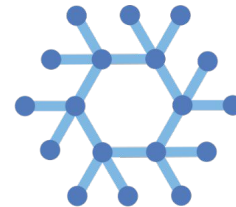
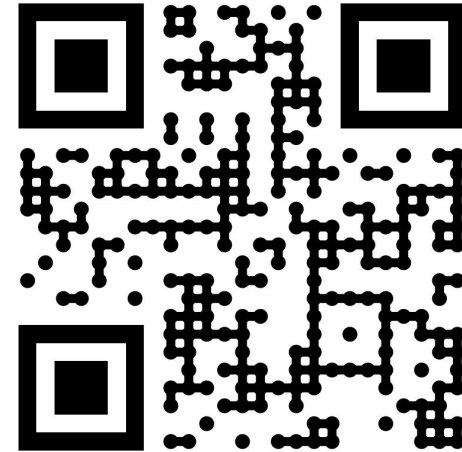
Conclusion

- Pins full source-based dependency closure
- Lightweight Linux namespace isolation
- Hermetic builds and declarative syntax provided by Nix
- Low-abstraction

=> Experiments defined with NixNet are easy to share, preserve, and to build upon

Sources

- <https://nixnet.org>
- <https://github.com/birneeee/nixnet>
- <https://github.com/birneeee/anrw26-nixnet-material>



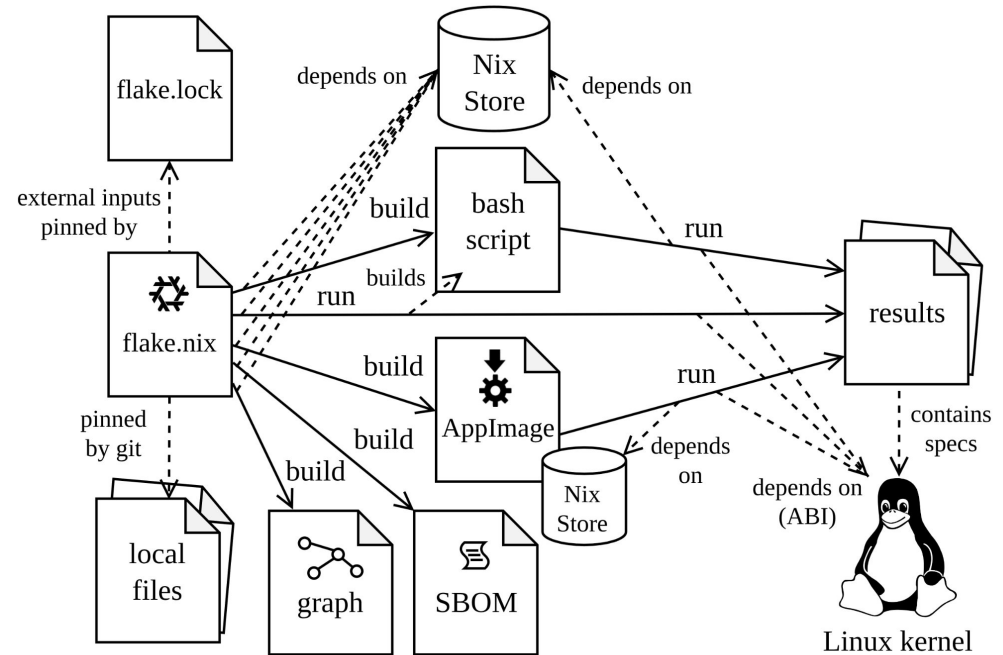
NixNet

Q&A

Backup Slides

NixNet artifact relations

- NixNet expression can derive multiple outputs
- runs on any machine with Nix installed
- or as bundled AppImage



Setup and cleanup time (s)

Nodes	Bash	clab	Mininet	NixNet
0	0.047	0.255	0.718	0.189
2	0.096	1.083	0.805	0.276
8	0.283	2.679	1.089	0.556
32	1.056	11.32	2.291	1.682
128	4.599	43.31	8.119	6.167
512	25.28	218.1	37.41	26.00

Peak memory usage (MB)

Nodes	Bash	clab	Mininet	NixNet
0	3.861	64.60	38.84	12.25
2	4.408	101.6	40.35	12.13
8	5.861	158.3	45.76	12.59
32	11.60	340.8	66.22	27.63
128	33.46	930.8	148.5	90.50
512	122.0	3737.3	478.7	339.1