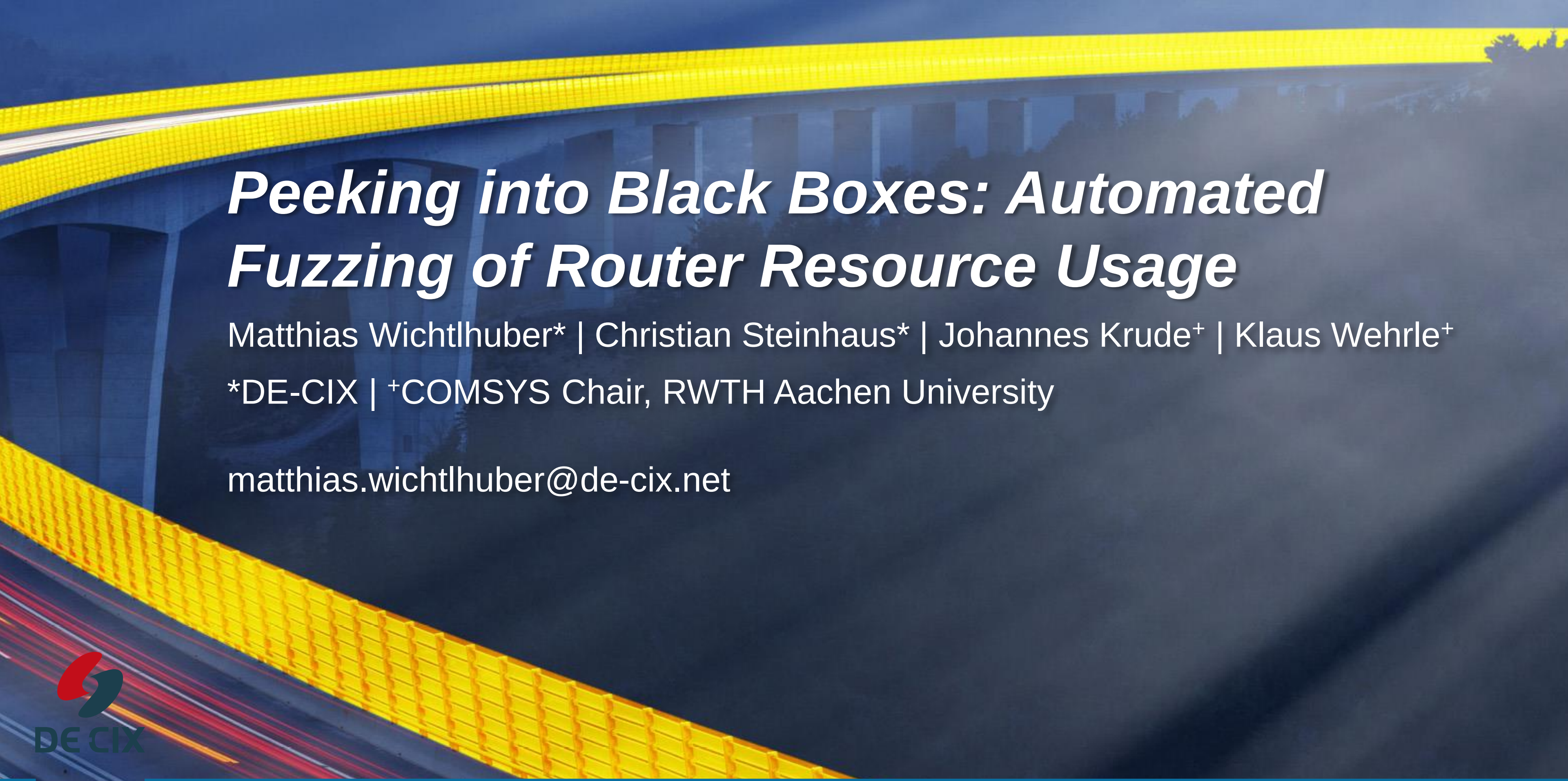


Peeking into Black Boxes: Automated Fuzzing of Router Resource Usage

Matthias Wichtlhuber* | Christian Steinhaus* | Johannes Krude[+] | Klaus Wehrle[+]

*DE-CIX | [+]COMSYS Chair, RWTH Aachen University

matthias.wichtlhuber@de-cix.net

Motivation

→Managing changes in critical networking infrastructures requires thorough testing

→Router resource testing often is a blind spot in test plans (black box)

- Where are the router‘s limits (e.g. TCAM space)?
- How much headroom is left for future innovation?

→Reasons:

- Vendors are tight-lipped on hardware resources
- Complex topic (even for vendors)

Our Approach

→Fuzzing/fuzzy testing in security research [1]

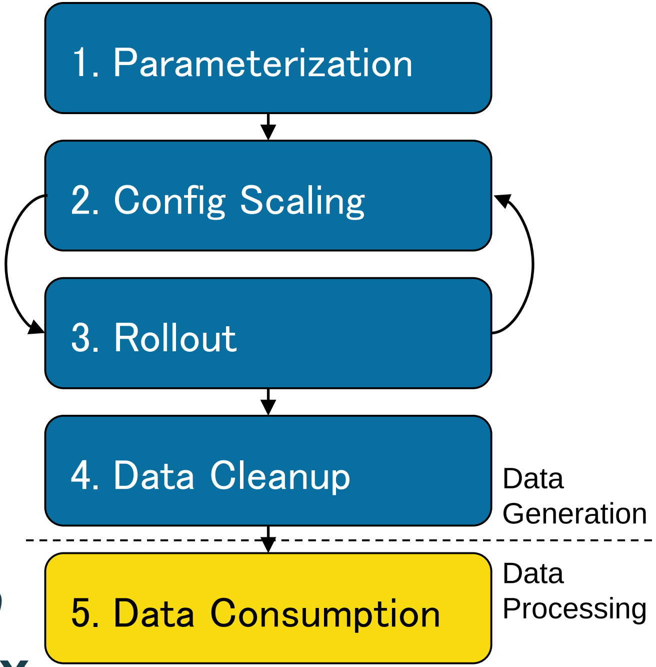
- Automated security testing
- Generate random inputs, monitor implementation behavior

→We are not looking for security issues but explore the HW limits of the router under test

1. Generate masses of guided, valid configuration changes
2. Measure router behavior: runtime errors and exposed hardware counters
3. Correlate configuration and measurements, identify scaling behavior and possible bottlenecks

[1] https://en.wikipedia.org/wiki/Fuzzing

Fuzzing Framework Design



→ Modular five stage framework

- Based on Python and Jinja2 for templating
- Flexible and adaptable (e.g. different vendors)

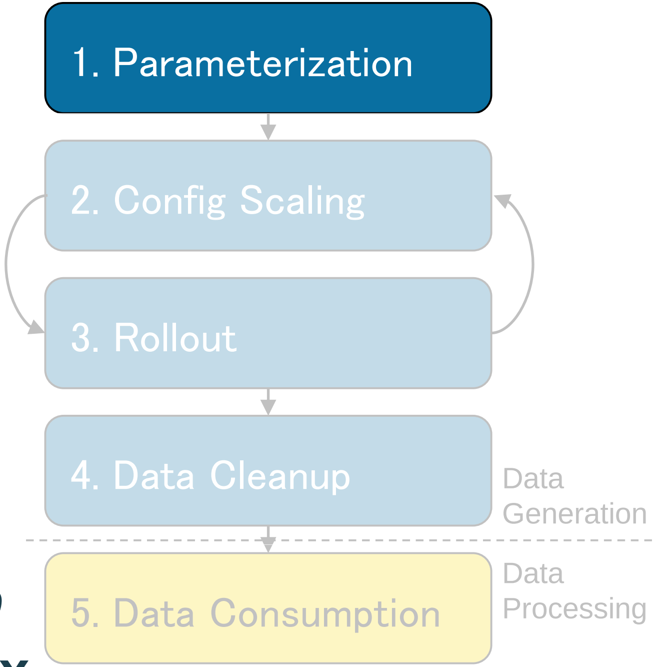
→Use existing production configuration

- Extend as required by test case
- Evaluate scalability of single routers (i.e. vertical)

→ Design goals

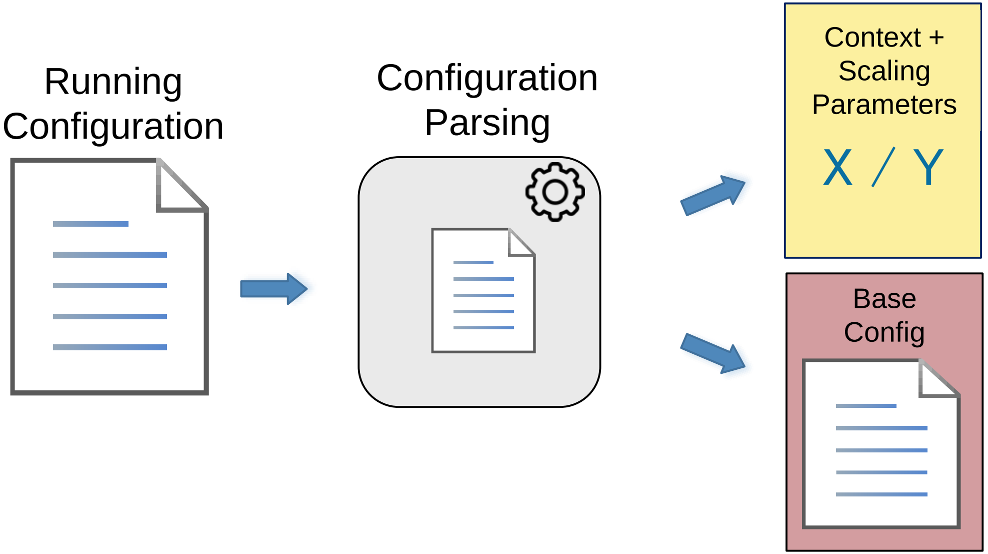
- Automation of repetitive steps (>1000 configurations)
- Visualization + Identification of bottlenecks
- User support in all stages

Framework Design: Parameterization

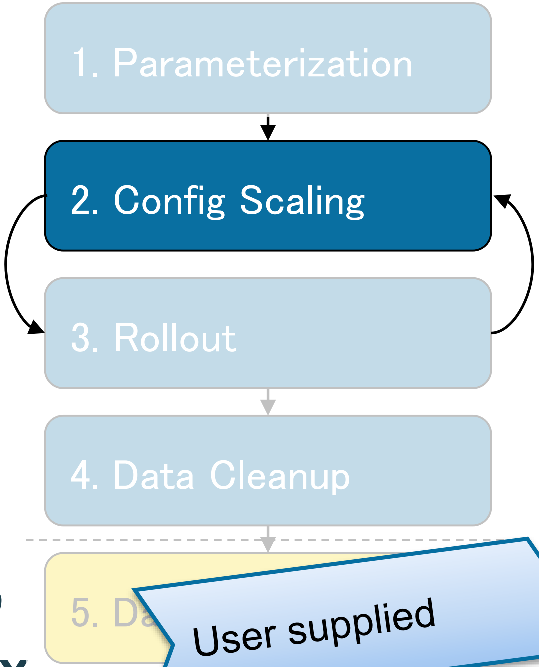


→ User provides production configuration

→ Extract scaling parameters



Framework Design: Config Scaling



```
Context
“e_pol”: {
  1: {
    “descr”: “This is policy 1”
  },
  . . .
  678: {
    “descr”: “Last policy”
  }
}
```

From parameterization

+

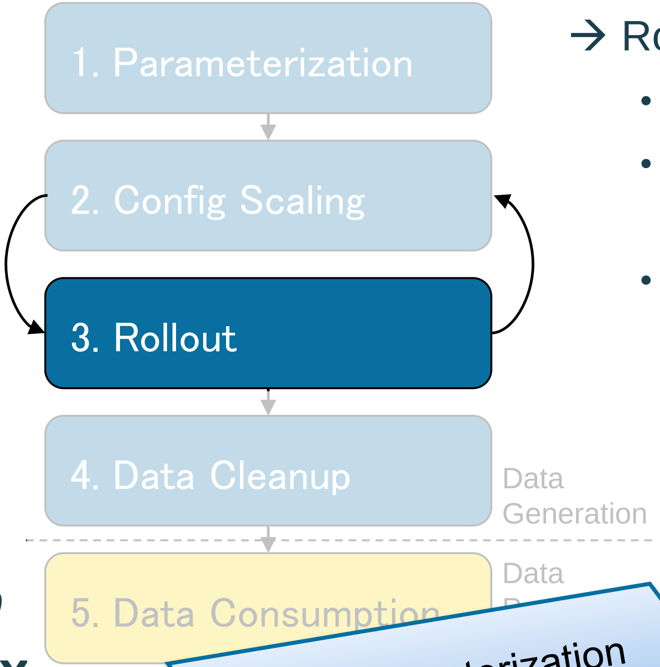
```
Jinja Template
forwarding
  {% for policyid, policy in e_pol.items() %}
  new rule {{ policyid }}
    description {{ policy.description }}
  done
done
```

User supplied

```
Rendered Extension Config
forwarding
  new rule 1
    description “This is policy 1”
  done
  . . .
  new rule 678
    description “Last policy”
  done
```

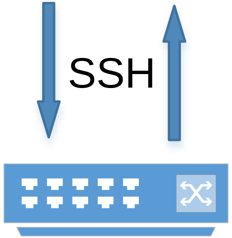
Inlined python

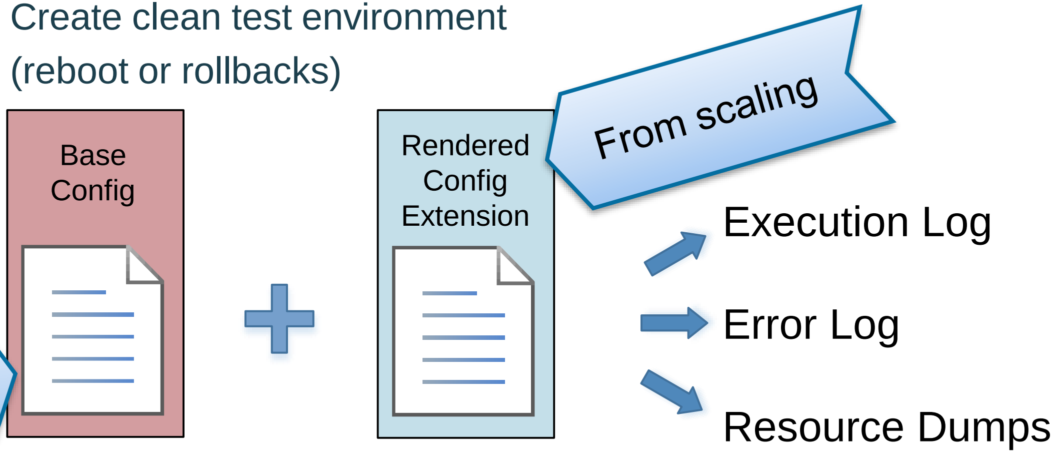
Framework Design: Configuration Rollout



→ Rollout

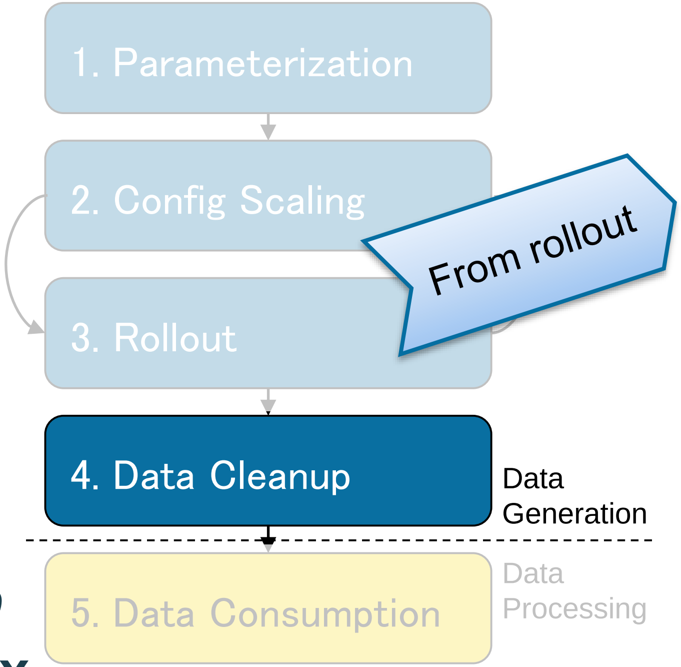
- Apply configuration on test router
- Gather logs (accepted/rejected, errors, hardware counter dumps, …)
- Create clean test environment (reboot or rollbacks)





From parameterization

Framework Design: Data Cleanup



→Extract data

- Execution log
- Error codes
- Resource usage

→Create data pool

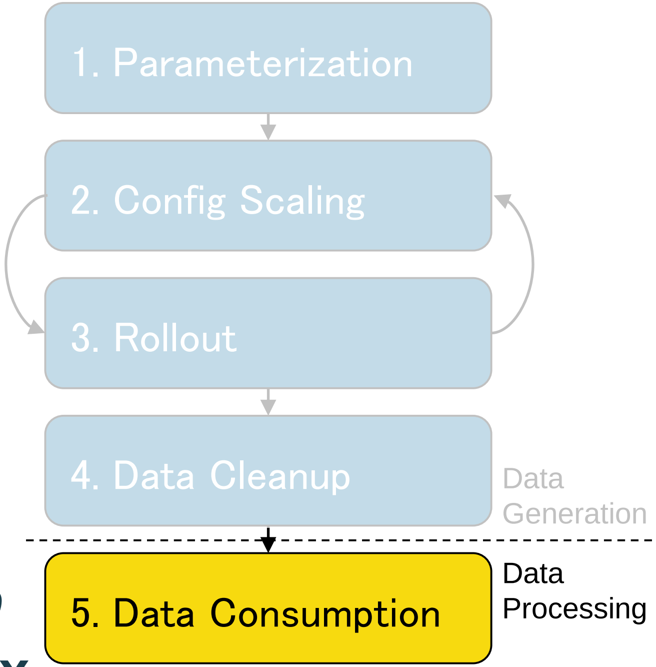
- Combine data from all runs

→Standardize output

- (see right)

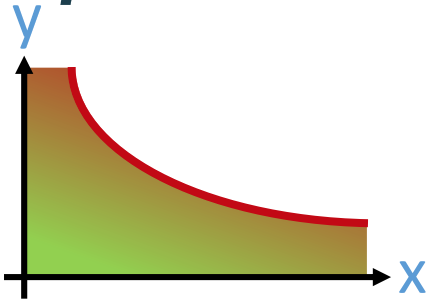
```
{
  "desc": "Resource X",
  "linecard": "20",
  "allocated": null,
  "scaling_x": 111,
  "scaling_y": 222,
  "error": "42",
  "errordesc": “Resource X exhausted“
}
```

Framework Design: Data Consumption



→Visualization

- Generate plots on measured errors
- Visualize resource usage



→Predictive modelling

- Decision tree model
- Provides flow-chart like predictions whether a scaled configuration will exceed available resources
- Useful for management and predictive maintenance

Case Study: QoS+ACLs for Traffic Filtering

→ Use Case

- Drop DDoS/unwanted traffic at IXP
- How many QoS policies+ACLs per port can we apply before running out of resources?
- How does resource usage scale?
- What are the bottleneck resources?

→ Test Setup

- Complex service router with multiple line cards
- Production configuration of a multiple Tbps/>100 ports router as a base configuration
- Generated extension configurations scale #QoS policies and #ACLs per policy

In the following, axes of plots are obfuscated due to NDAs.

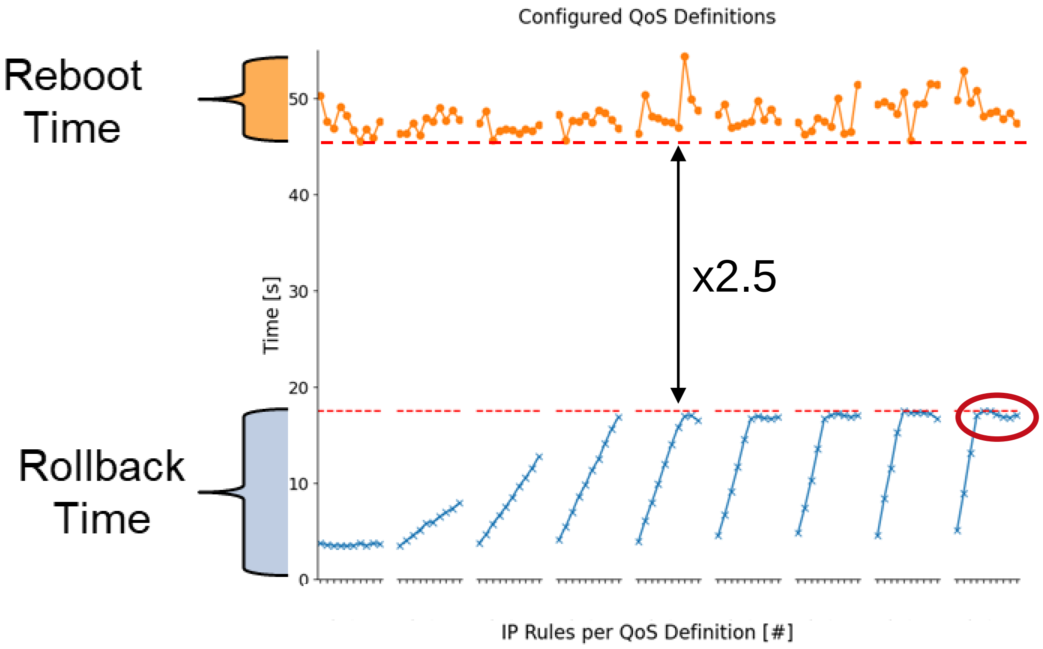
Timescale of Experiments

Step Size (#QoS and #ACLs/QoS)	**10**	**20**	**40**	**80**
Time	23h11m	5h44m	100m	23m
#Data Points	2295	598	168	48

→ Total runtime

- Configuration scaling, rollout, data collection, environment cleanup
- Environment cleanup is the bottleneck (see next slide)

Environment Cleanup: Rollback vs. Reboot



Reboot Time

x2.5

Rollback Time

- At least 2.5 times speedup with rollbacks compared to reboots per measurement

- Rollback time depends on size of extension configs
- Plateaus (○) indicate large failing configs

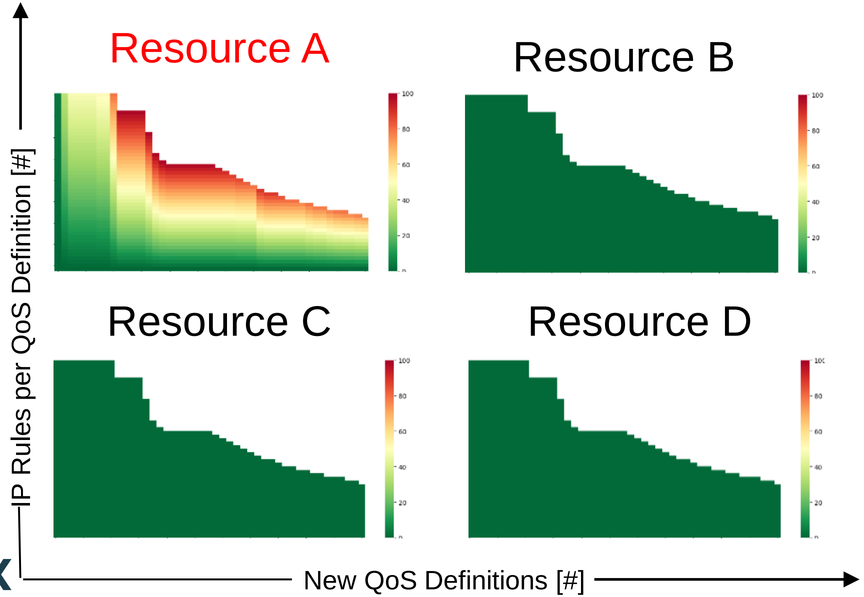
Resource Usage

→Aggregation across all line cards

- Identify bottleneck (max. HW counters)

→Drill down per line card for Resource A

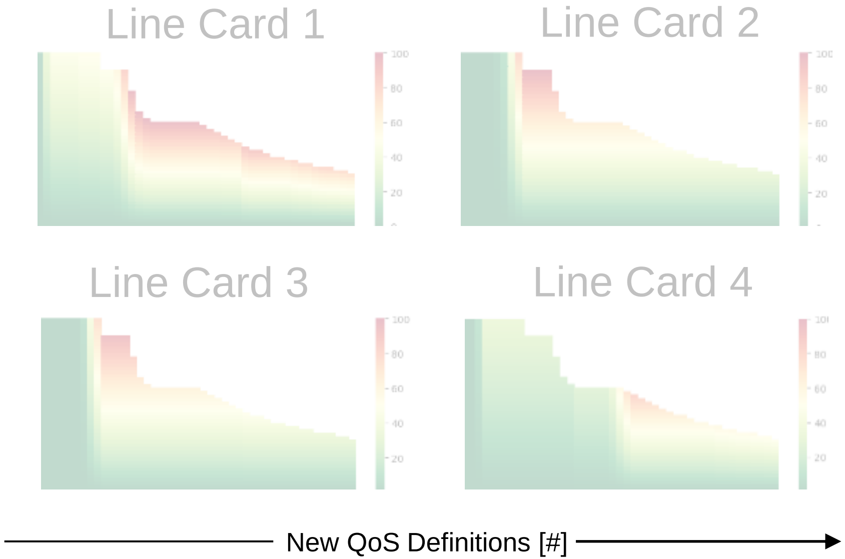
- Identify bottleneck hardware module


Resource A

Resource B

Resource C

Resource D


Line Card 1

Line Card 2

Line Card 3

Line Card 4

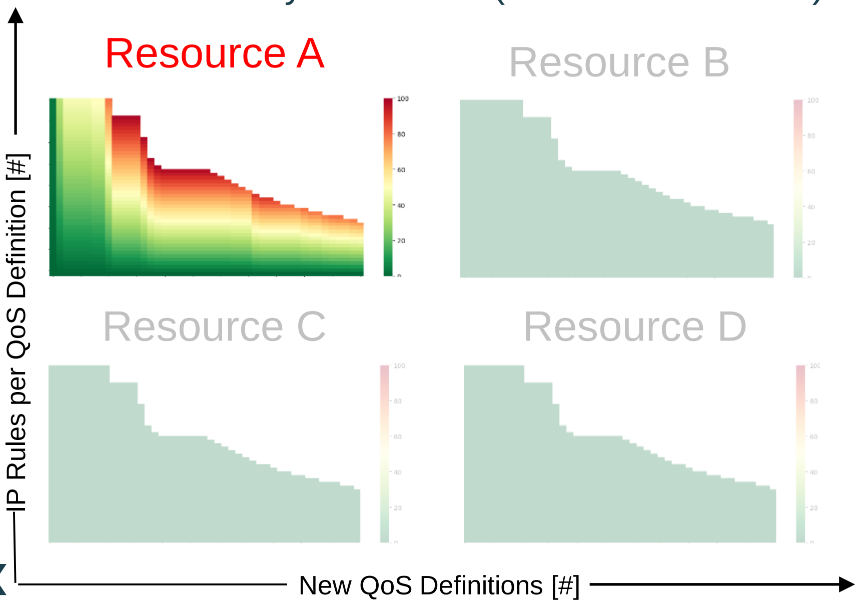
Resource Usage

→Aggregation across whole device

- Identify bottleneck (max. HW counters)

→Drill down per line card for Resource A

- Identify bottleneck hardware module

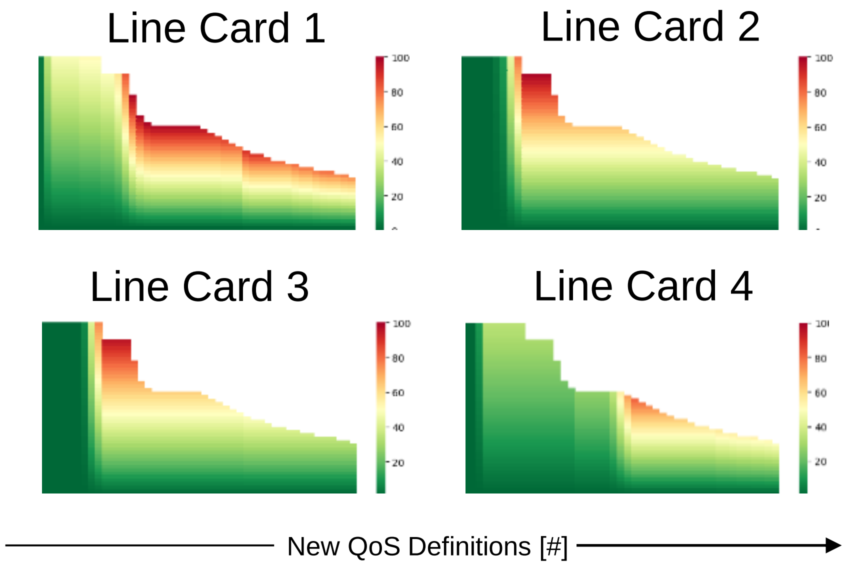


Resource A

Resource B

Resource C

Resource D



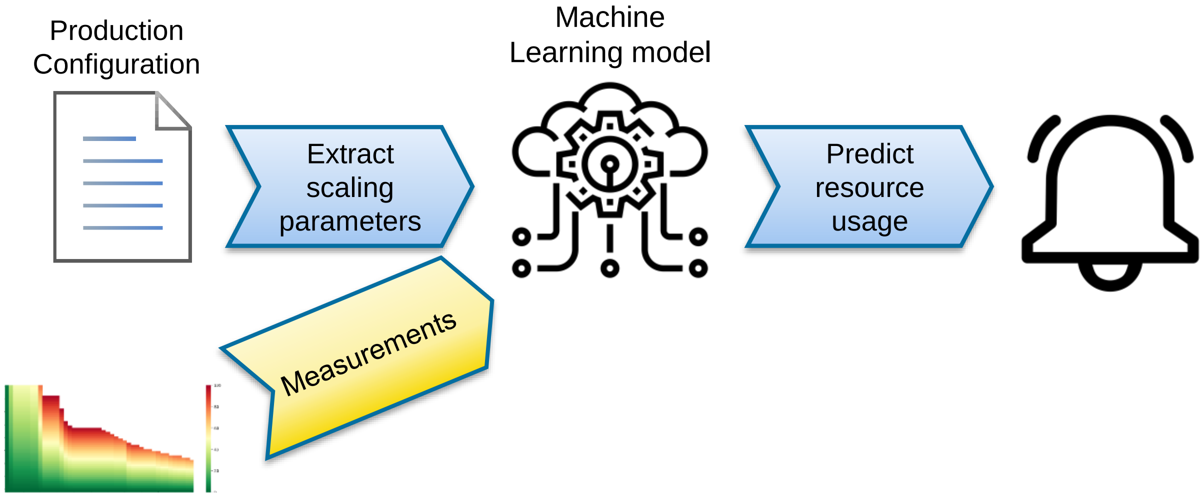
Line Card 1

Line Card 2

Line Card 3

Line Card 4

Monitoring without Measurements



Production Configuration

Extract scaling parameters

Machine Learning model

Predict resource usage

Measurements

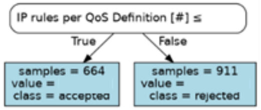
→ Can we use the datasets to approximate resource usage without measurement?

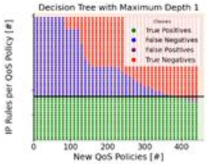
→ Predictive maintenance can help mitigating problems even before deployment

Decision Tree Resource Model

- →Decision Tree ML-model
- →Train/test (70/30) split of measured data
- →Prediction accuracy > 98%
- →Tree depth allows tuning understandability vs. accuracy trade-off

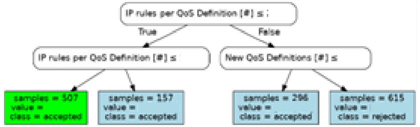
Maximum Depth 1

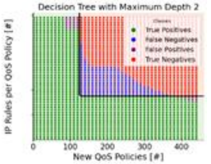




Accuracy 77.93%

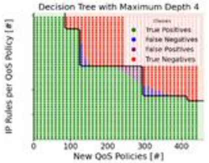
Maximum Depth 2





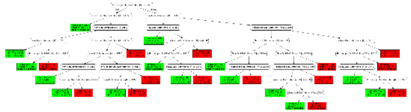
93.04%

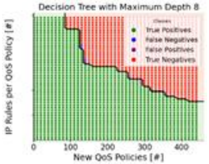
Maximum Depth 4



98.22%

Maximum Depth 8





98.67%

Summary and Conclusions

→Deploying new features to critical infrastructure often requires resource testing

- Vendors are tight-lipped on hardware resources
- Resource testing can become complex quickly

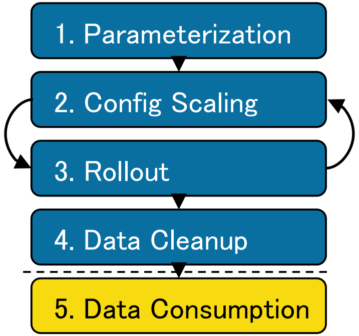
Summary and Conclusions

→ Deploying new features to critical infrastructure often requires resource testing

- Vendors are tight-lipped on hardware resources
- Resource testing can become complex quickly

→ Our method/framework supports resource testing automation

- Generates more than 2000 data points in < 24 hours
- Identifies bottleneck per router and per line card
- Creates accurate and human readable prediction models



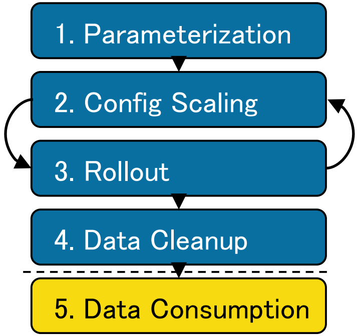
Summary and Conclusions

→ Deploying new features to critical infrastructure often requires resource testing

- Vendors are tight-lipped on hardware resources
- Resource testing can become complex quickly

→ Our method/framework supports resource testing automation

- Generates more than 2000 data points in < 24 hours
- Identifies bottleneck per router and per line card
- Creates accurate and human readable prediction models



→ The general method has been used in practice by DE-CIX

- For assessing configuration changes and for dimensioning products
- For assessing the accuracy of simulated router instances vs. real-world hardware
- For validating vendor claims on HW capabilities

Thank You for Your attention!

DE-CIX Management GmbH | Lindleystr. 12 | 60314 Frankfurt | Germany
Phone + 49 69 1730 902 0 | sales@de-cix.net | www.de-cix.net