BGPEval: Automating Large-Scale Testbed Creation

Nils Rodday*†, Gabi Dreo Rodosek*

*Research Institute CODE, Universität der Bundeswehr München,
†University of Twente

Abstract—**BGP has been known to be vulnerable to hijacking and path manipulation attacks for many years. Several solutions have been proposed to secure either the origin, the path, or both. A known issue for new ideas is their evaluation. Simulation environments are easy to use but can only mimic real-world deployments to a certain extent. In this work, we propose BGPEval. A framework that is capable of creating large-scale testbeds based on the KVM hypervisor and Docker container technology that mimics the interconnection of ASes according to a provided AS graph. We use several layers of abstraction to spawn and inter-connect as many as 55,000 containers. Our work takes significant effort away from an evaluator, who can now focus on the implementation rather than the creation of testbeds for the evaluation.**

Index Terms—**Topology generator, BGP, RPKI, Security**

I. Introduction

The Border Gateway Protocol (BGP) has been known to be vulnerable to hijacking and path manipulation attacks for many years [1], [2]. Many security solutions have been proposed. Some focus on origin validation [3], others on path validation [4]–[9], and some very recently on path plausibility [10], [11]. Newly proposed algorithms are sometimes purely theoretical, sometimes evaluated in simulation testbeds [12], and in some instances implemented in Open Source routers [10]. In order to provide a framework to test innovative algorithms, the National Institute of Standards and Technology (NIST) developed the BGP-Secure Routing Extension (SRx) software suite, which allows a fairly easy integration of new security algorithms [13]. Once the algorithm is implemented, the framework allows for native deployment on the host or the instantiation of a containerized environment in which it is possible to connect multiple routers. However, the creation of testbeds is limited to a predefined configuration file, which has to be manually created and can only be executed on a single host, limiting the testbed's size. Our work extends the currently available NIST BGP-SRx software suite with the capability of spawning large-scale testbeds in a swarm on multiple hosts in a fully automated manner mimicking a provided input graph. The input graph used for our tests resembles the inter-domain routing infrastructure. We create a directed input graph from publicly available BGP collector data [14], [15] and enrich the graph with the CAIDA Autonomous System (AS) relationship dataset (as-rel2) [16].

Generalization. Our primary goal in proposing this framework is to ease the pain of creating testbeds to evaluate BGP security solutions. However, since the framework relies on a container-based solution, further application areas can be found within the networking domain and beyond where our topology generator can be helpful, *e.g.,* evaluation of intrusion detection systems or firewalls. As long as the evaluator is able to provide a container that encapsulates the software that needs evaluation or testing, our framework could be adopted to serve different purposes. Organizations are constantly looking for staging environments to test new additions to their network before the new software is rolled out. Existing tools can create a graph that represents the networking status of an enterprise network [17], [18]. These tools listen to existing network traffic from multiple vantage points and can visualize the network topology. Based on such a graph, our BGPEval framework can easily create a testbed that mimics the network topology of the real network using the same production software within the deployed containers. It allows for integrating additional components safely before rolling them out in a production environment.

Contributions. In this work, we propose a framework called BGPEval for automated testbed creation. In detail, we make the following contributions:

1) We propose an engine that accepts a directed graph as input and generates router configuration files to resemble the topology in a complex testbed with software routers.
2) We create an input graph from publicly available BGP collectors and enrich that data with the CAIDA AS relationship dataset to obtain a directed graph representing the inter-domain routing infrastructure.
3) We fully automate the creation of large-scale testbeds over multiple layers of abstraction, *e.g.,* hardware servers, hypervisors, and containerized solutions. We show the possibility to scale the testbed with our current infrastructure to up to 55,000 containers.
4) We validate our proposal by deploying two independent BGP routing daemons: Quagga and GoBGP. Both daemons are capable of connecting to the NIST BGP-SRx software suite to run BGP security algorithms.
5) We publicly release all source code and documentation of our framework[1].

The remainder of this paper is structured as follows: Section II provides background information on topology generation and simulation frameworks. Section III presents our

[1]https://github.com/nrodday/CNSM-23

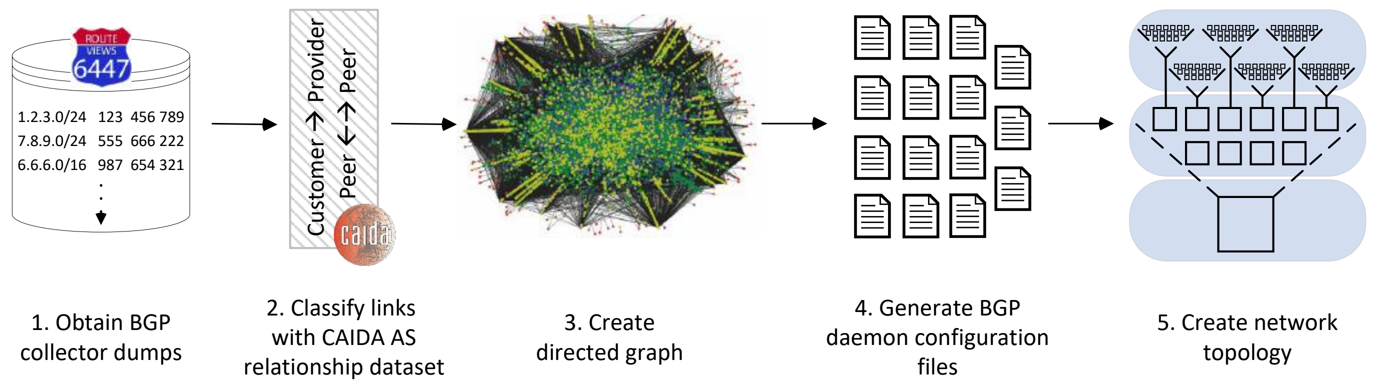
ROUTE VIEWS 6447
1.2.3.0/24 123 456 789
7.8.9.0/24 555 666 222
6.6.6.0/16 987 654 321
Customer → Provider
Peer ←→ Peer
caida
1. Obtain BGP collector dumps
2. Classify links with CAIDA AS relationship dataset
3. Create directed graph
4. Generate BGP daemon configuration files
5. Create network topology

Figure 1. BGPEval methodology. We obtain BGP collector data and enrich the data with CAIDA AS relationship information to obtain a directed graph. Afterwards, we generate BGP router configuration files and create the testbed via multiple layers of abstraction.

methodology and shows how BGPEval works, while Section IV introduces the NIST BGP-SRx software suite. We create a proof-of-concept in Section V and show how the topology generator can be helpful in practice. Limitations are discussed in Section VI while Section VII summarizes our findings.

II. Background

Methods to perform evaluations of algorithms in network environments have been discussed for many years. The most prominent simulation frameworks in use today are NS-3 [19], Omnett++ [20], and GNS3 [21]. On a theoretical level, the execution of an algorithm within a simulation environment to validate its correct working might seem adequate. On a more practical level, running code in production will always cause a multitude of errors or interconnectivity issues that could not be tested within the simulation environment. Also, source code typically needs to be ported into the simulation environment. The programming language might not match the one the simulation environment requires, and reimplementation is necessary. MiniNet [22] is a platform that enables network algorithm evaluations primarily for SDN-based technologies. Automated network configuration has been proposed for MiniNet [23] and GNS3 [24]. Our BGPEval framework differs from simulation or emulation environments in that it allows for the execution of the same binaries that will be executed later on within the production environment. Moreover, BGPEvals' focus lies on the automated creation of the topology of the testbed itself, something that still has to be done manually in all previously mentioned frameworks.

III. Methodology

To create our large-scale testbeds, we propose the following strategy. Figure 1 illustrates the workflow.

Input graph. First, our topology generator expects a directed graph as an input parameter. This graph is read as a serialized Python object and instantiated in memory. We use the Python library NetworkX [25] to create the graph. Our generator is designed to resemble the input graph as provided. Therefore, an evaluator is able to create with very few lines of code complex topologies that our framework accepts as input.

Since we want to instantiate the inter-domain routing infrastructure on an AS level, we use public BGP collector data from RIPE RIS [15] and Routeviews [14]. We obtain a collector dump for June 1, 2023, from 00:00:00 until 23:59:59. The resulting file contains 678 GB of data. We create a graph from the relationships between ASes inferred from the AS_PATH attribute of each announcement contained within. Afterwards, we apply the CAIDA AS relationship dataset (as-rel2) [16] to label edges between nodes. Edges that could not be labeled and isolated nodes are removed from the graph. The final directed graph consists of 74,110 nodes. Classifying them into tiers, according to [26], we obtain 105 tier one, 11,237 tier two, and 62,768 tier three nodes.

It is essential to highlight that we create a graph that is abstracted from the inter-domain routing infrastructure, but any other graph as input would be accepted and the topology built accordingly.

Configuration file generation. Second, BGPEval creates the configuration files for either Quagga or GoBGP daemon instances. Since their syntax differs, the respective output can be chosen with a switch. Our framework reads the input graph and translates the contained relationship information (customer-to-provider and peer-to-peer relationships) to BGP daemon readable format. Therefore, the amount of created configuration files matches the amount of ASes contained within the graph. Each file contains the instructions for the BGP daemon to connect with its peers.

In addition to the required information about peers, we implement the Gao-Rexford model [27] to imitate the proper propagation of BGP updates. We discuss the limitations of our approach in Section VI.

Testbed creation. Third, we instantiate the testbed. This process spans multiple layers of abstraction. Depending on the size of the input graph, more or less resources are required to support the testbed. Our setup works with a single or multiple hardware nodes. This might be particularly helpful in scenarios where many nodes are available, but each only provides a few

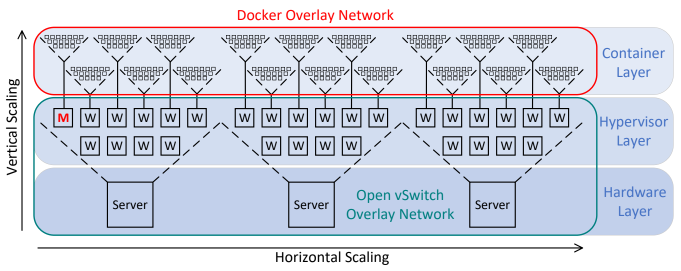

Figure 2. BGPEval architecture. We observe three layers. Servers at the hardware layer, VMs at the hypervisor layer, and containers at the container layer. The hypervisor layer has a single manager instance and many worker instances. Servers and VMs communicate via an Open vSwitch Layer 2 overlay network, while containers are interconnected via a Docker overlay network. Each container requires a static IP address to generate router configuration files.

resources. We had to create a testbed with multiple layers, as each Docker container instantiates a network interface on the host. The Operating System (OS) running on the server and within a VM becomes unresponsive if more than ∼800 of such containers are spawned. Therefore, we encapsulated 450 Docker containers into a VM. Each server is capable of hosting multiple VMs.

We show our architecture in Figure 2. At the bottom layer, we use either a single or multiple servers for horizontal scaling. These servers are interconnected via a data link layer overlay network. To keep it simple, we only require an SSH connection from one server to the other. Each server creates an Open vSwitch bridge and a tunnel interface for each adjacent server. The traffic of the tunnel interface for the adjacent server is routed via the SSH connection. We enable the Spanning Tree Protocol (STP) to avoid forwarding loops if more than three servers are interconnected. With such a simple setup, all servers are capable of exchanging traffic.

The middle layer consists of VMs hosted by a KVM hypervisor on each server. Each VM connects to the Open vSwitch bridge with its dedicated interface. Therefore, each VM can access the Internet via its regular interface and access the data link layer overlay network via the bridged interface. As a result, a VM hosted on one server can send traffic to any other VM hosted on any other server participating in our setup. Each interface connecting to the bridge needs an IP address assigned to exchange traffic. We implement this by hosting a DHCP server on a manager VM on the first server (marked in red in Figure 2). All other VMs are workers that send out DHCP requests. Depending on the resources available on each server, we spawn and interconnect more or less VMs with each other.

The top layer contains Docker containers. This final layer contains the actual BGP experiment testbed. We use Docker Swarm [28] to implement our testbed. The manager VM hosts the swarm manager. All other worker VMs cloned on each server contain a start-up script that joins them into the swarm. When all VMs are spawned, we can observe the same amount of VMs as Docker worker nodes within the Docker manager node. It is, therefore, fairly simple to prune and respawn our entire testbed by simply destroying the VMs and reinitiating the VM cloning process. The Docker image is deployed in a registry service. This service is available to all worker nodes. Containers using that image can be instantiated on any worker node. Each container is deployed as a Docker service. The configuration files for each AS are deployed as Docker configs. We use the swarm manager to distribute the Docker configs to all nodes that spawn a container requiring a specific configuration file. In order to interconnect the containers, we use another overlay network. We spawn each container with a dedicated network interface connected to the overlay network in addition to the default interface. However, Docker cannot assign static IP addresses to containers deployed within a swarm. This is a crucial requirement as we need to interconnect BGP routers within the containers and, therefore, need to know the IP address of the BGP peer already at the creation of the configuration files. To work around this issue, we automated the following procedure: First, we use SSH to log into each Docker node and disconnect the overlay interface from the overlay network. Second, we assign the predefined IP address for this container, and third, we restart the interface. The Docker manager is unaware of the change as it happens on the OS level within the container and not via the management interface. This becomes only a problem if a container crashes and Docker automatically respawns the container with its old address. However, when Docker containers crash, the computational limit has been reached in any case, and we do not exceed such limits to provide a stable testbed.

Within each container, we wait for two minutes until the IP address change for the overlay interface took place and start the NIST BGP-SRx server. Afterwards, we start the BGP daemon, which connects to the local BGP-SRx server instance and all BGP peers.

Finally, a fully functional BGP testbed is running that interconnects according to the provided input graph. The evaluator can now manually log into certain ASes to alter announcements (or automate such use cases) and observe the propagation of the announcements. Moreover, BGP security algorithms implemented in the BGP daemon and the BGP-SRx server can be thoroughly tested.

IV. NIST BGP-SRx Software Suite

Throughout our work, we heavily use the NIST BGP-Secure Routing Extension software suite [13]. It ships with multiple components: BGP-SRx server, Quagga routing daemon, Resource Public Key Infrastructure (RPKI) test harness, and BGPSecIO generator.

BGP-SRx server. The functionality of security algorithms is not implemented on routing daemons but instead encapsulated in the BGP-SRx server. This change in philosophy contrasts the Request for Comments (RFCs) that detail the execution of *e.g.,* BGPSec operations on the router itself [4]. On the one hand, it provides more flexibility as complexity is outsourced to the BGP-SRx server, and processing on the router is not delayed; on the other hand, such change requires the

BGP-SRx server as an additional component for processing and increases management complexity. The current BGP-SRx server implements RPKI (RFC 6811), Border Gateway Protocol Security (BGPsec) (RFC 8205), and Autonomous System Provider Authorization (ASPA) (Draft Version 1).

Quagga. The routing daemon has been extended to implement an SRx-Proxy API, which forwards validation requests of received BGP updates to the BGP-SRx server. The current Quagga software is capable of requesting validation for RPKI, BGPsec, and ASPA.

RPKI test harness. It functions as a Relying Party (RP) software instance and allows the evaluator to create RPKI or ASPA objects via a simple Command Line Interface (CLI). It does not implement the RPKI delegation, signing, and publication procedures but directly exports the Validated ROA Payload (VRP) to the BGP-SRx server, therefore reducing complexity.

BGPSecIO generator. It is a handy tool built on the exaBGP routing daemon, a Python-based routing implementation. The generator allows to script a sequence of announcements that the BGPSecIO generator sends towards a specific BGP peer. It is, therefore, possible to craft arbitrary announcements, including hijacks, manipulated path segments, route leaks, etc.

V. Proof of Concept

In order to test the correct workings of our BGPEval framework, we create two container images that are each deployed in a testbed, see Figure 3. The first image contains Quagga as a routing daemon; the second image features goBGP. We use the inter-domain infrastructure graph as an input to create the underlying topology as described in Section III. Each container image also runs a BGP-SRx server instance to which the routing daemon is locally connected. We use Zebra [29] to install routes the routing daemon receives into the Route Information Base (RIB). Once a route has been installed, we can send data plane packets to verify whether an address range is actually reachable. We limit the number of BGP-SRx instances connecting to a single RPKI test harness instance to 10,000. Hence, we need multiple RPKI test harness instances to support a large testbed. These are deployed as additional containers and distributed throughout the worker nodes to avoid overwhelming a single node with too many connection requests.

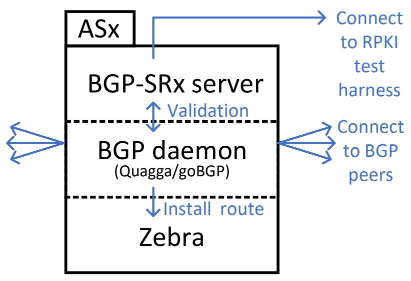

Figure 3. Container components.

Our physical infrastructure spans three servers:

1) 56 Cores @2.1 Ghz, 768 GB RAM
2) 128 Cores @2.45 Ghz, 2,048 GB RAM
3) 128 Cores @2.45 Ghz, 1,024 GB RAM

Our results show that we could create a testbed as large as 55,000 ASes for the Quagga image and 45,000 ASes for the goBGP image. Since we spawn containers linearly throughout all VMs to avoid overwhelming a single host, creating one testbed requires roughly two hours.

VI. Limitations

Scaling limit. Unfortunately, we could not scale the testbed to up to 74,110 containers as we experienced heavily increased latencies shortly after reaching the mentioned testbed sizes that led to connection renegotiations and, therefore, a collapse of the testbed. We suspect the management engine of Docker is saturated at some point that comes with the immense amount of network connections that need to be tracked and supported throughout the build of the topology. We have already increased several OS and application layer limits [30] to support larger networks. Moreover, we replaced the default networks with much larger /8 networks to support that many containers. However, we suspect that forwarding network packets from one container to the other becomes unsustainable at some point, considering that every packet has to cross multiple layers of abstraction. Hence, we observe huge latencies (>1sec) during scaling operations when going above the determined limits. CPU utilization and RAM usage do not seem to be a problem, as plenty of computational power and memory are unused when the problem occurs.

Nonetheless, scaling up to 55,000 containers is a significant advancement, as previous research and industry contributions were limited to $\sim$ 10,000 containers within a single testbed.

Gao-Rexford. In Section 1, we implemented the Gao-Rexford model [27] for proper routing within the testbed. A plain Gao-Rexford implementation is much simpler than private router configurations based on confidential agreements, *e.g.,* provider A is preferred for prefix B since the profit would be higher. However, such simplification is indispensable as private router configurations are impossible to obtain for each AS. By implementing the Gao-Rexford model, we abstract fine-grained policies and assume proper routing behavior for all participants, which is not always true. As a result, if *e.g.,* a route leak scenario needs to be tested within the testbed, the configuration of the leaking AS would require alteration.

AS-level abstraction. We consider an entire AS with many eBGP speakers a single entity with a common policy. In reality, that is very rarely the case, but it is a necessary abstraction to make testbed creation feasible. More detail for intra-AS topology would significantly increase the complexity of the testbed and, therefore, require an even higher number of supported containers. Moreover, intra-AS topology is hidden from the outside world and considered a business secret by the provider; hence, it is hard to infer. Yet, the limitation is essential to consider when trying to deploy scenarios with partial adoption of security algorithms within a particular AS, *e.g.,* RPKI filtering would be enabled on one eBGP speaker, but not on another, see [31].

RPKI test harness. We are required to create multiple instances to distribute the load to avoid overloading a single RPKI test harness instance. This is a drawback when different deployment scenarios are tested since the status of RPKI, ASPA, and AS-Cones objects needs to be precisely the same for all RPKI test harness instances. Otherwise, different ASes would obtain other objects, and a consistent evaluation would not be possible.

VII. Conclusion

In this work, we proposed the BGPEval framework. It is based on the NIST BGP-SRx software suite that allows for easy integration of experimental routing security algorithms and deploys large-scale testbeds based on a directed input graph. We generate the directed input graph from public BGP collectors and label the data with the CAIDA AS relationship dataset. Our framework generates router configuration files that represent the interconnection according to the provided graph and ensures a stable testbed creation over multiple layers of abstraction. We can scale the testbed to a size of 55,000 containers. The framework reduces the effort of an evaluator to create a test infrastructure significantly and allows for the quick generation of multiple different scenarios.

Acknowledgment

We thank the anonymous reviewers and Julius Hermelink for their constructive feedback. Moreover, we thank Oliver Borchert and the NIST Internet Technologies Research Group [32] members for their time for discussion.

References

[1] S. Cho, R. Fontugne, K. Cho, A. Dainotti, and P. Gill, "BGP hijacking classification," in *2019 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, 2019, pp. 25–32.

[2] K. Butler, T. R. Farley, P. McDaniel, and J. Rexford, "A Survey of BGP Security Issues and Solutions," *Proceedings of the IEEE*, vol. 98, no. 1, pp. 100–122, 2009.

[3] M. Lepinski and S. Kent, "An Infrastructure to Support Secure Internet Routing," RFC 6480, Feb. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6480.txt

[4] M. Lepinski and K. Sriram, "BGPsec Protocol Specification," RFC 8205, Sep. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8205.txt

[5] R. White, "Architecture and deployment considerations for secure origin bgp (sobgp)," IETF Secretariat, Internet-Draft, June 2006. [Online]. Available: https://www.ietf.org/archive/id/draft-white-sobgp-architecture-02.txt

[6] S. Kent, C. Lynn, and K. Seo, "Secure Border Gateway Protocol (S-BGP)," *IEEE Journal on Selected areas in Communications*, vol. 18, no. 4, pp. 582–592, 2000.

[7] T. Wan, E. Kranakis, and P. C. van Oorschot, "Pretty Secure BGP, psBGP." in *Network and Distributed System Security Symposium (NDSS)*. Citeseer, 2005.

[8] P. v. Oorschot, T. Wan, and E. Kranakis, "On Interdomain Routing Security and Pretty Secure BGP (psBGP)," *ACM Transactions on Information and System Security (TISSEC)*, vol. 10, no. 3, pp. 11–es, 2007.

[9] J. Karlin, S. Forrest, and J. Rexford, "Pretty Good BGP: Improving BGP by Cautiously Adopting Routes," in *Proceedings of the 2006 IEEE International Conference on Network Protocols*. IEEE, 2006, pp. 290–299.

[10] A. Azimov, E. Bogomazov, R. Bush, K. Patel, J. Snijders, and K. Sriram, "BGP AS_PATH Verification Based on Autonomous System Provider Authorization (ASPA) Objects," Internet Engineering Task Force, Internet-Draft draft-ietf-sidrops-aspa-verification-16, Aug. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-sidrops-aspa-verification/16/

[11] J. Snijders, stucchi lists@glevia.com, and M. Aelmans, "RPKI Autonomous Systems Cones: A Profile To Define Sets of Autonomous Systems Numbers To Facilitate BGP Filtering," Internet Engineering Task Force, Internet-Draft draft-ietf-grow-rpki-as-cones-02, Apr. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-grow-rpki-as-cones-02

[12] A. Cohen, Y. Gilad, A. Herzberg, and M. Schapira, "Jumpstarting BGP Security with Path-End Validation," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 342–355.

[13] O. Borchert, K. Lee, K. Sriram, D. Montgomery, P. Gleichmann, and M. Adalier, "BGP Secure Routing Extension (BGP-SRx): Reference Implementation and Test Tools for Emerging BGP Security Standards," National Institute of Standards and Technology, Tech. Rep., 2021.

[14] RouteViews Project, "University of Oregon RouteViews Project," http://www.routeviews.org, 2013, [Online; accessed 16-Juli-2020].

[15] RIPE NCC, "RIPE Routing Information Service (RIS)," https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris, 2020, [Online; accessed 16-October-2020].

[16] CAIDA, "The CAIDA AS Relationships Dataset, 20221001," 2022. [Online]. Available: https://www.caida.org/catalog/datasets/as-relationships/

[17] "Netdisco." [Online]. Available: http://netdisco.org/

[18] "Nessus." [Online]. Available: http://www.nessus.org/

[19] G. F. Riley and T. R. Henderson, "The ns-3 Network Simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[20] A. Varga, "OMNeT++," in *Modeling and tools for network simulation*. Springer, 2010, pp. 35–59.

[21] J. C. Neumann, *The Book of GNS3: Build Virtual Network Labs using Cisco, Juniper, and more*. No Starch Press, 2015.

[22] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.

[23] N. Rodday, R. van Baaren, L. Hendriks, R. van Rijswijk-Deij, A. Pras, and G. Dreo, "Evaluating rpki rov identification methodologies in automatically generated mininet topologies," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 530–531.

[24] R. Emiliano and M. Antunes, "Automatic Network Configuration in Virtualized Environment using GNS3," in *2015 10th International Conference on Computer Science & Education (ICCSE)*. IEEE, 2015, pp. 25–30.

[25] A. Hagberg, P. Swart, and D. S Chult, "Exploring Network Structure, Dynamics, and Function using NetworkX," 1 2008. [Online]. Available: https://www.osti.gov/biblio/960616

[26] N. Rodday, L. Kaltenbach, I. Cunha, R. Bush, E. Katz-Bassett, G. D. Rodosek, T. C. Schmidt, and M. Wählisch, "On the Deployment of Default Routes in Inter-domain Routing," in *Proceedings of the ACM SIGCOMM 2021 Workshop on Technologies, Applications, and Uses of a Responsible Internet*, 2021, pp. 14–20.

[27] L. Gao and J. Rexford, "Stable Internet Routing Without Global Coordination," in *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2000, pp. 307–317.

[28] F. Soppelsa and C. Kaewkasi, *Native Docker Clustering with Swarm*. Packt Publishing Ltd, 2016.

[29] K. Ishiguro, "Gnu Zebra," 2002. [Online]. Available: https://www.gnu.org/software/zebra/

[30] T. Petric, "Running 1,000 Containers in Docker Swarm," 2017. [Online]. Available: https://www.cloudbees.com/blog/running-1000-containers-in-docker-swarm

[31] N. Rodday, Í. Cunha, R. Bush, E. Katz-Bassett, G. D. Rodosek, T. C. Schmidt, and M. Wählisch, "Revisiting RPKI Route Origin Validation on the Data Plane," in *Proc. of Network Traffic Measurement and Analysis Conference (TMA), IFIP*, 2021.

[32] National Institute of Standards and Technology (NIST), "Internet Technologies Research Group," 2023. [Online]. Available: https://www.nist.gov/ctl/wireless-networks-division/internet-technologies-research-group