TPy : A Lightweight Framework for Agile Distributed Network Experiments

Steinmetzer, Daniel; Stute, Milan; Hollick, Matthias

(2018)

TPy: A Lightweight Framework for Agile Distributed Network Experiments

Daniel Steinmetzer*
Secure Mobile Networking Lab
TU Darmstadt, Germany
dsteinmetzer@seemoo.de

Milan Stute*
Secure Mobile Networking Lab
TU Darmstadt, Germany
mstute@seemoo.de

Matthias Hollick
Secure Mobile Networking Lab
TU Darmstadt, Germany
mhollick@seemoo.de

ABSTRACT

Experimental validation of novel network solutions, protocols, and applications gains increasing importance. The complexity of today's network systems makes evaluations in physical testbeds mandatory to capture real-world effects. However, this causes methodological and technical issues and challenges researchers in handling their *agile* testbed deployments. In contrast to Internet-scale testbeds, most *agile* experiments require specific topologies, specialized hardware, or a custom environment. They typically run only a few times and demand live user interaction. Existing management systems for Internet-scale testbeds do not accommodate these needs due to their complexity and maintenance overhead. In this paper, we present TPy, a lightweight and flexible framework to conduct distributed network experiments. TPy is written in Python and extendable via modules. To demonstrate its versatility and ease-of-use, we use TPy to perform experiments in the domains of millimeter-wave and secure multi-hop communications. We share TPy as open source software to support the community of experimental evaluation.

ACM Reference Format:
Daniel Steinmetzer, Milan Stute, and Matthias Hollick. 2018. TPy: A Lightweight Framework for Agile Distributed Network Experiments. In *12th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization (WiNTECH '18), November 2, 2018, New Delhi, India.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3267204.3267214

*Both authors contributed equally to the paper.

WiNTECH '18, November 2, 2018, New Delhi, India

1 INTRODUCTION

As the complexity of novel network solutions, protocols, and applications tends to increase, we need confidence that their designs work as intended. Analytical or simulation-based validation alone cannot provide this confidence as assumptions may be incorrect or real-world effects are not taken into account. Several conferences have started promoting experimentally validated research in recent years: ACM MobiCom invites *verification papers* that should "verify and/or characterize recent breakthrough results in mobile computing using rigorous experimental methodologies" since 2017 [2]. In the same year, ACM WiSec introduced the *reproducibility label* to "enable dissemination of research results, code and experimental set-ups, and to enable the research community to build on prior experimental results" [1]. While we welcome this shift, it forces researchers to follow a much more systematic approach towards managing their experiments.

In fact, a plethora of static Internet-scale testbeds that include powerful experiment automation and management software exists. For example, GENI provides a platform for networking and distributed systems research. It consists of many federated testbeds [8] that allow conducting large-scale network experiments. Some of these testbeds include wireless nodes, such as Emulab [20] and ORBIT [15]. There are also some dedicated wireless testbeds [4, 7, 11] consisting of wireless sensor nodes. Due to their scale, these testbeds are designed to support a large number of users that share the same resources. Therefore, parts of the software implements testbed virtualization [7, 10, 16] which requires dedicated machines that run the management software. Besides, using the testbeds requires the users to go through excessive pages of documentation or get familiar with a custom configuration language [16] or communication API [12].

For many of our past works, we did not require such a complex setup. For example, virtualization and user registration are unnecessary for small-scale specialized testbeds that are only used by a single group or individual researchers. Due to lack of a lightweight and flexible alternative that could be fitted to our varying needs, we typically resorted to writing "disposable" shell scripts to configure the testbed, automate experiment execution, and collect results. Apparently, this

approach resulted in redundant work since the same basic steps had to be repeated for every new testbed setup. And since every configuration was unique, code reuse was minimal. While we do not have empiric evidence, we believe that other researchers have made similar experiences. To ease this burden, we present TPy (Testbed Python), a lightweight framework for *agile*[1] distributed network experiments which facilitates experiment-driven research. With TPy, we reduce the effort that is required to set up one-time testbeds and conduct experiments with them. In addition, we believe that TPy ensures *reproducibility* of distributed network experiments. Researchers might be more willing to share the code of their experimental setups if the code is concise and is, therefore, more likely to be reused by others. To this aim, TPy implements the following design goals:

- **Flexibility of deployment**. TPy is not tied to specific hardware or software platforms. As the only requirements, testbed devices need to (1) run Python and (2) be accessible in the network.
- **Independence of infrastructure.** TPy does not require any additional hardware to operate. It only requires the testbed devices and a machine that acts as a controller.
- **Ease of use**. TPy experiments scripts are written in pure Python, while the accompanying configuration files use the simple INI format. In conjunction with open source scientific software such as *Pandas* [14] and *Matplotlib* [9], the entire "setup–run–collect–evaluate–publish" cycle can be implemented in Python. Also, TPy's quick start guide fits on a single page.
- **Extensibility**. We ship TPy with a number of basic modules, but everything (software tool, hardware chip, ...) that can talk to Python can become a module and can be integrated into the experiment.
- **Interactivity**. As a result of the above points, we can reuse the code that runs the experiments and generates publication-ready plots to build a demonstrator.

The rest of this paper is structured as follows: We present our TPy framework in Section 2 and provide a user guide in Section 3. Next, we showcase TPy by conducting network experiments in two distinct domains of millimeter-wave (mmwave) and secure multi-hop communication in Sections 4 and 5, respectively. We provide a discussion in Section 6 and conclude our work in Section 7.

2 TPY FRAMEWORK

The TPy framework allows controlling a large number of devices from a single point of control with low organizational

[1]We consider experiments that use specific topologies and hardware, require a user-controlled environment, are run only a few times, or allow for live user interaction as *agile*.

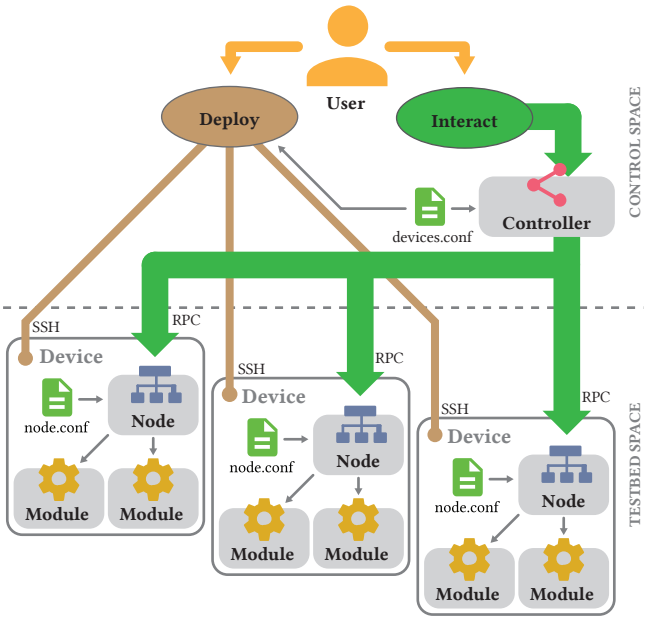


Figure 1: TPy Architecture

overhead. In this section, we describe the architecture of this framework. In particular, we present our system model, our design decisions, as well as the components and connections in the architecture. We further describe the deployment process and the interaction with testbed devices.

2.1 System Model

Our system model consists of multiple distributed testbed devices and a single control instance to run measurements from control space. We support heterogeneous devices with different capabilities and features. For example, a device can be a powerful server, an embedded device, or any other network device that matches our requirements. Our requirements are low, we only assume devices to run Python and be reachable from the network. To work with the testbed, we distribute the TPy framework and proper configuration files to the devices. Features that are required in the experiments and control the devices are encapsulated in modules at the devices and remotely accessible. Devices can be equipped with different configurations and modules to realize various functionalities in the testbed. TPy runs a node service on each device to expose modules as specified in the configuration. Users can connect to these nodes and access modules remotely. To handle multiple devices, TPy provides a controller. The controller interacts with all devices from the central control space. It is instantiated with a testbed-specific device configuration that specifies the available nodes and how they can be reached in the network. For example, it defines each testbed device by hostname or IP address. The controller consolidates access

to remote features allowing to perform various distributed network experiments. Figure 1 summarizes the system model comprising the controller, nodes, and modules.

2.2 Design Decisions

To address the heterogeneity of network devices and to be independent of specific hardware and software, we implement TPy in Python. Python is supported by many operating systems and hardware platforms and easily allows to wrap system calls. Due to its flexibility and the large set of available libraries, Python is the ideal candidate to handle agile experiments. To keep our system simple and easy to deploy, we directly connect to all distributed devices without using a dedicated control server. We assume IP connectivity to the devices and establish connections via simple TCP/IP sockets to be independent of specific network topologies. As these connections are easy to route, our testbed can spread over multiple subnetworks and pass firewall with proper configuration. Shell access to remote devices is optional and only needed for automated deployment. With Python's interactive mode, we enable dynamic handling of testbed experiments with "real-time" interaction which is particularly valuable for live demonstrations. Moreover, we encapsulate the testbed topology and configuration to separate it from the experiment description.

2.3 Components

The architecture of TPy consists of multiple components that are distributed among the testbed network. TPy distinguishes between (1) *modules* that expose specific features available on the devices, (2) *nodes* that represent the devices in the testbed, and (3) a *controller* that manages the testbed, connects to the nodes, and interfaces the modules. In the following, we describe these three components in detail and explain how to configure them in Section 3.

Modules. Modules encapsulate the features that are available on the distributed testbed devices. All modules are plain Python objects and derived from an abstract `TPyModule` class. They can have a state, store internal properties, and can be initialized with parameters. Modules can implement arbitrary Python code, include other libraries, and invoke systems calls. Possibilities are not restricted by TPy. For example, we provide an interface module that allows configuring network interfaces. It allows setting the IP and MAC address, and read properties such as the number of received packets and error rates. This is achieved by integrating additional Python libraries (e. g. PyRIC [21]) or invoking respective system commands (e. g. `ip`, `ifconfig`, ...). Modules can extend each other. For example, our wireless interface module inherits the interface module and adds specific features that are not available on wired interfaces such as reading the received signal strength and SSID. Moreover, we provide modules that wrap common system commands such as `ping` and `iperf`. Modules are instantiated and executed locally on the devices and, thus, independent from the actual testbed topology. With our modular concept, TPy easily allows to integrate new features: custom modules can be simply added to the framework.

Nodes. Nodes represent devices in the testbed and handle multiple modules that are exposed to the network. A node is a service that is instantiated from a `TPyNode` class. It binds to a specific TCP/IP port and exposes the available modules via remote procedure calls (RPCs). To this end, the node implements the Pyro library [5] to allow invoking methods from remote machines. The node itself provides only limited features and acts as a gateway to access the modules. Available modules are configured in a *node configuration* file that specifies their properties. The node creates the module instances given the respective configuration. TPy supports to instantiate modules multiple times with different names and properties. All modules in the network are identified by unique resource identifiers (URIs) that consist of the host address and port as well as the module name. Other devices in the network obtain access to the module by querying this URI. The node itself exposes a list of available modules.

Controller. The controller orchestrates the testbed and coordinates the network experiments. It establishes the connection to all nodes and provides direct access to the modules. The controller is instantiated by the `TPyControl` class. Available nodes are listed in a *device configuration* that specifies the host address and port at which the node service can be reached as well as other optional parameters which might be used in experiments such as the geographical location of the device. Default parameters can be provided globally to be valid for all nodes in the testbed. The controller connects to the nodes and queries all available modules. For each node, the controller creates a local stub that binds and forwards all method calls to the remote location. Therewith, all remote modules are accessible as Python objects in the control space. As a result, users directly interface remote modules and experience no differences by working remotely. TPy abstracts the distribution of devices and consolidates all features locally.

2.4 Connections

The controller connects to all devices in the testbed that are specified in the device configuration. While this is typically done with RPCs, we also consider optional secure shell (SSH) connections that simplify the deployment and configuration of devices. In the following, we describe both connections, RPCs and SSH.

Remote Procedure Calls. RPCs allow to call remote methods or procedures as they were local ones. For each remote

method, the caller creates a local stub. This stub binds to the remote method. It forwards all arguments to invoke the method at the remote location and transfers the result back again. In particular, the stub implements the same list of arguments, serializes them and sends them over the network. At the remote location, the callee deserializes the arguments and invokes the requested method with the provided arguments. The return value is serialized again and transmitted back to the caller. Fortunately, this is already implemented in the Pyro library [5]. As all arguments and results need to traverse the network, we keep our exposed methods as simple as possible and omit complex data structures. For example, we internally work with private objects to store the state in our modules, but only expose methods that interface this via primitive data types. Doing so, we keep the network and serialization overhead low.

Secure Shell. SSH connections to the distributed network devices are optional and only required for deployment and configuration. Obtaining shell access to remote nodes is highly beneficial as it allows to check and reconfigure devices while running experiments easily. In TPy, we utilize SSH connections to deploy the node service and modules to the devices. We trigger the node service to start, stop, or restart whenever necessary. Moreover, SSH connections easily allow changing the node configuration. Nevertheless, we keep SSH as an optional feature that can be substituted by manual access or other deployment strategies. RPCs do not use SSH by default since SSH is an optional requirement for TPy. However, if required, RPCs can be transparently tunneled using SSH port forwarding [22], thus, providing secure RPCs.

2.5 Deployment

To deploy nodes, modules, and configurations to distributed testbed nodes, TPy supports an auto deployment via SSH. TPy creates a Python distribution package from the source code of the node service and modules. It extracts the host information from the device configuration and pushes the archive through an SSH connection. At the remote system, it triggers the installation of the archive along with required dependencies and starts the node service with the provided configuration. Depending on the hardware, the installation of a Python package might take some time. To minimize deployment time, we parallelize the deployment process. Hence, deploying few devices takes nearly as long as deploying hundreds of devices. A manual deployment mode is appropriate if no direct SSH connection is available or deployment is handled by another mechanism. In this case, the user must ensure that the node service is running on the remote devices and exposes the required modules. The TPy controller provides a command to check the deployment and list all accessible nodes and modules.

2.6 Interaction

To interact with the testbed, the user instantiates the controller with a specific devices configuration. The controller then queries the nodes and exposes the features in the modules to be directly accessible. TPy allows to dynamically interact with network devices by using an interactive Python shell. Large measurement data can be collected with simple scripts. One of the great advantages of TPy is the abstraction of physical location. Nodes and modules are accessible as simple Python objects that hide the underlying connections. This makes distributed network experimentation very convenient and straightforward as features can be accessed with few lines of code. In the following section, we provide a user guide to explain the basic operation of network experiments with our TPy framework.

3 USER GUIDE

This section explains how to use TPy in the form of a step-by-step walkthrough in which we will configure and start a minimal working example. We follow the same steps as described in Section 2. Consequently, we need to configure (1) the controller as well as (2) the nodes, (3) deploy and start the node instances, (4) connect to the nodes via the controller, and finally (5) start interacting with them.

3.1 Configure the Controller

First, we describe our testbed via a device configuration file, i. e., `devices.conf`. The most basic device configuration file simply consists of a node `name` (section title) and a `host:port` pair under which the node is reachable. `host` can either be an IP address or domain name. The special `DEFAULT` section will apply to all other sections but can be overwritten in the node sections. Figure 2a shows an example `devices.conf` file that configures two nodes.

3.2 Configure the Nodes

Each node instance will start with its own configuration, e. g., `node.conf`. The `TPyNode` section includes information relevant to the node instance. The `port` setting must be the same as the `port` setting in the respective node section in `devices.conf`. After the `TPyNode` section follows sections for all modules that the node should expose. Each module section has the `module` setting which gives the module type. For convenience, `module` can be omitted if it is the same as the section title (see `Ping` example above). Giving a module a different name allows for instantiating multiple modules of the same type. Each module must be explicitly listed in the

```
[DEFAULT]
port = 42337
[NODE1]
host = 10.10.10.1
[NODE2]
host = node2.example.org
```

(a) Sample `devices.conf`

```
[TPyNode]
host = 0.0.0.0
port = 42337
[Ping]
[AdHoc]
module = AdHocInterface
interface = wlan0
ipaddress = 10.0.0.1
channel = 1
bssid = c0:ff:ee:c0:ff:ee
ssid = tpy-test
```

(b) Sample `node.conf`

Figure 2: TPy configuration files. The simple INI file format consists of sections with a *title* (in brackets) and optional *properties* (in `key=value` format).

configuration file even if the section body is empty. We depict an example `node.conf` file for `NODE1` in Fig. 2b.

3.3 Deploy and Start the Nodes

We provide a command line tool `tpy` which allows for deploying and (re-)starting the node instance.

```
tpy deploy -d devices.conf
tpy restart -d devices.conf
```

If you are developing new modules, simply prepend a `make` command to the above two commands to build and push the new version to the devices.

3.4 Connect to the Nodes

When all nodes are running, we can establish a connection to them from Python via the controller. In a Python shell, run

```
import tpycontrol as tpy
devices = tpy.Devices('devices.conf')
ctrl = tpy.TPyControl(devices)
ctrl.showinfo()
```

When initializing, `TPyControl` reads the `devices.conf` and attempts to connect to them. After this, `showinfo` prints all connected node instances as well as their exported modules.

3.5 Interact with the Nodes

After initializing the controller, we can start interacting with the remote nodes. Remote node objects are accessible via the controller's `nodes` dictionary attribute. Similarly, modules are accessible as an attribute of their respective remote node's object. In the following example, we start the nodes' Wi-Fi interfaces in IBSS mode and send an ICMP echo request from node 1 to node 2:

```
for name, node in ctrl.nodes.items():
    print('%s: set ad hoc up', name)
    node.AdHoc.up()
src = ctrl.nodes['NODE1']
dst = ctrl.nodes['NODE2']
src.Ping.ping(dst.AdHoc.ipaddress)
```

TPy enables the user to write experiments in an extremely *concise* way. Instead of cluttering the experiment scripts with (unimportant) details such as setting the Wi-Fi channel, SSID, BSSID, and IP address via the `iw` and `ip` command line tools, TPy scripts contain high-level experiment descriptions. In this case, running `ping` via an Wi-Fi interface in ad hoc mode. The details are encapsulated in their respective modules and their configuration file entries.

4 USE CASE A: MM-WAVE EXPERIMENTS

The first use case of TPy is to measure the throughput of directional IEEE 802.11ad connections with different modulation and coding schemes (MCS). Specifically, we consider a testbed with four TP-Link Talon AD7200 tri-band routers that run a customized OpenWrt system [19] and are distributed throughout an office room with few meter distance. In particular, we configure one device to act as an access point (AP) and the other three to act as stations. Two stations are located in line-of-sight with the AP at distances of about 2 m and 4 m, respectively. The third device is placed behind an obstacle at 4 m distance. We individually connect the stations to the AP and measure the achievable uplink throughput while sequentially increasing the enabled MCS from index 0 to 12.

4.1 Configuration

This experiment requires additional modules to control the Talon devices, namely, (1) a `Hostapd` module to turn one device into access point mode (2) a `WPASupplicant` module to connect another device to the AP, (3) a `WiGigInterface` module to configure the enabled MCS in the IEEE 802.11ad chip, and (4) an `IPerf` module to measure the actual throughput. All these modules act as a wrapper for system commands that

```
[DEFAULT]
port = 42337
[T01]
host = 192.168.1.1
[T02]
host = 192.168.1.2
; similar for T03 and T04
```

(a) `devices.conf`

```
[WiGigInterface]
interface = wlan2
[Hostapd]
interface = wlan2
[WPASupplicant]
interface = wlan2
[IPerf]
```

(b) `node.conf`

Figure 3: TPy configuration files for use case A.

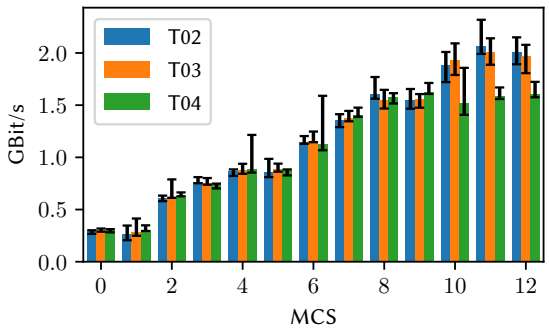


Figure 4: Achievable TCP uplink throughput at the distributed stations showing the average and the 95 % confidence intervals.

are available on the OpenWrt system. The `WiGigInterface` module sends direct commands to the firmware running on the WiFi chip via the `wil6210` kernel driver. Doing so, we can reconfigure the internal rate search algorithm and limit the enabled MCS. Our node configuration is shown in Fig. 3b. We configure all Talon devices with a routable IP address and sequentially name them `T01`, `T02`, `T03`, and `T04`. Our `devices.conf` is shown in Fig. 3a.

4.2 Interaction

In our sample experiment script, we set the first device, `T01`, to AP mode by launching a `hostapd` daemon. We iterate over all other nodes and set one at a time in station mode to connect to the AP by using the `WPASupplicant` module. On all other devices, we disable the IEEE 802.11ad network interface. As soon as the devices are associated, we iterate over all available MCS from 0 to 12 and successively enable them. In the first round we only enable MCS 0, in the second MCS 0 and 1, and in the last round all 13 MCS. In each of these steps, we start an `iperf` server on the AP node and measure the uplink TCP throughput from the station for 10 seconds. The results are parsed from the JSON output of the `iperf` command and converted to Python dictionaries. The results are directly transferred to and stored locally at the controller. After completing the measurements for all MCS, the device disconnects, and we continue the measurements with the next station. We repeat this process three times. Even though the measurements take some time to complete, intermediate results are directly accessible. After all the measurements are performed, we combine the results in a single dictionary structure and compute the average and the 95 % confidence intervals.

4.3 Results

We show the results of this use case in Fig. 4. The first two stations that are in line-of-sight to the AP, both, achieve a maximum throughput of about 2.0 GBit/s. The distance towards the AP only causes minor differences. The third station that is blocked by an obstacle requires environmental reflections to communicate with the AP. However, it appears that our office room contains objects that are good reflectors—the third station achieves a throughput of about 1.6 GBit/s. Moreover, we see that the highest throughput at this station is already achieved with MCS 8. The first two stations obtain additional gains with higher MCS such that the throughput saturates with MCS 11. In summary, these results demonstrate the spatial propagation effects of mm-wave communications: line-of-sight connections are required for high data rates but also strong reflections provide a decent link quality.

While this use case constitutes a rather simple scenario, it can be easily extended to large-scale networks with hundreds of distributed devices: simply adding more devices to the configuration file is sufficient. The script iterates over all available nodes and, thus, directly scales with the size of the testbed without any code changes.

5 USE CASE B: MULTI-HOP EXPERIMENTS

In our second use case, we use TPy to conduct wireless multi-hop experiments. In particular, we evaluate a secure communication protocol that combines routing and data forwarding [17] which so far has only been evaluated in a simulator. Since the protocol has been implemented in the Click modular router [13], we can deploy the protocol implementation on a Linux-based testbed. Our testbed consists of ten APU nodes [6] which are distributed in an office building.

5.1 Configuration

For this experiment, we use three additional modules: (1) The `Click` module maintains a router instance and can communicate with the router via a local TCP socket at runtime.

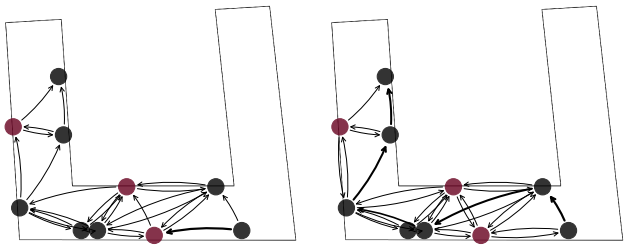
```
; only showing two nodes
[APU05]
host = mesh-apu05
coordinates = 159,18
[APU11]
host = mesh-apu11
coordinates = 105,15
attacker = True
```

(a) `devices.conf`

```
[AdHoc] ; as in Fig. 2b
[Click]
config = conf.click
socket_port = 7777
[NTP]
server = ntpd.local
[ITG]
```

(b) `node.conf`

Figure 5: TPy configuration files for use case B.



(a) $k \in [0, 15]$ **(b)** $k \in [16, 255]$

Figure 6: Path Selection. Flow from bottom right to top left. Attackers are red. Edge thickness idicates usage frequency.

(2) The `NTP` module allows us to query the node's time synchronization status towards a pre-configured NTP server. With this, we can bound the maximum error in message delay measurements which rely on log entries on the source and destination nodes. (3) The `ITG` module provides a wrapper to a traffic generator [3]. Figure 5b shows the corresponding node configuration. In the device configuration (Fig. 5a), we add additional per-node attributes: the geographical coordinates for plotting (Fig. 6) and a boolean flag to indicate which node should later act as an attacker.

5.2 Interaction

The protocol [17] selects next hops based on the neighbor's reliability. By using end-to-end acknowledgements, it learns from failures and, thus, converges towards an attacker-free path. We evaluate how the protocol [17] behaves under a blackhole attack. In particular, we are interested in the packet delivery rate (Fig. 7), the delivery delay (Fig. 8), and the detailed path selection (Fig. 6). We average the results over 100 flows with 256 packets each.

While we cannot include the full evaluation script in this paper, we briefly summarize the main high-level steps to describe the experiment in the *interaction* phase: (1) Prepare all nodes, i. e., set all interfaces to ad hoc mode and start the Click router. (2) Activate malicious nodes via the Click control socket. (3) Wait until all nodes have synchronized to the NTP server with an error below 0.1 ms giving us an upper bound of 0.2 ms for the error in the delay measurements. (4) Start the traffic generator endpoints on the source and destination nodes. (5) Wait for the traffic generator to stop. (6) Poll packet log entries from all nodes via the Click control socket. (7) Process log entries as *Pandas* [14] data frames and generate the plots using *Matplotlib* [9]. With TPy, we can not only automate experiment evaluation but use *the same code to build a demo* which collects and processes data from the nodes in "real time." In this instance, we periodically perform the last two steps immediately after step (4) to build a live map of the current path selection.

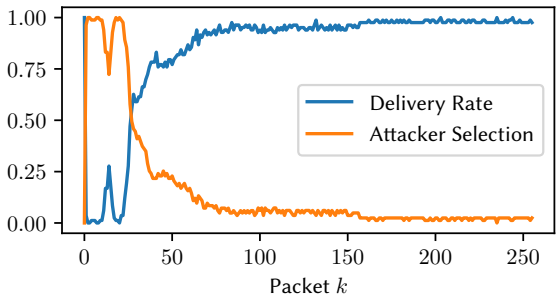


Figure 7: Packet Delivery Rate and Attacker Selection

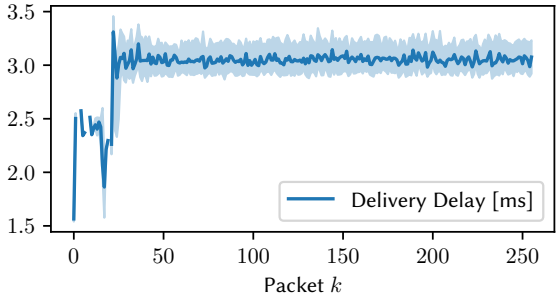


Figure 8: Delivery Delay. The solid line shows the mean, and the shaded areas highlight the 0.25 to 0.75 quantiles.

5.3 Results

We examine the performance under a blackhole attack where an attacker forwards all broadcast packets to comply with route exploration but drops unicast packets to disrupt route exploitation. We see how the protocol converges towards an attacker-free path with each packet that the source sends to the destination via a 5-hop route with three attackers in Fig. 6. Figure 7 shows the packet delivery rate as well as the attacker selection. The attacker selection metric is one if, for a given packet k, at least one attacker was selected during route exploitation by any non-attacking node in the network. Consequently, attacker selection is axis-symmetric to the packet delivery rate. We further show the delivery delay in Fig. 8. For the first few packets, the delay is about 2.5 ms which indicates that a route via an attacker could be faster. However, delay stabilizes around 3.0 ms when an attacker-free path is chosen primarily.

6 DISCUSSION

We discuss the feasibility of using TPy for interactive demonstrators, the overhead of deploying full-fledged management systems for testbeds with a short lifetime, and future work.

Interactive Experiments. In Section 5, we discuss the possibility to use TPy for research demos by the example of

a live network map. We note that performance of such a live map very much depends on capabilities on the nodes (CPU time required to generate the data), link between nodes and controller which aggregates the data (bandwidth), and the processing capabilities of the aggregator which needs to process the data and generate the plots within an update interval. In addition, since we need to query the nodes during the experiment, we might influence its outcome which should be fine for the purpose of a research demonstrator.

Management Overhead. Quantifying the benefits of different testbed automation and management frameworks is difficult. Existing systems serve different purposes, most of which require an extensive one-time setup phase that is amortized by simplified experiment management. Therefore, the overhead of deploying a management system highly depends on the expected lifetime of the testbed itself. This lifetime usually short for individual projects. A suitable management system should be chosen in respect of the available infrastructure and long-term perspective. TPy does not compete with existing management solutions but provides a lightweight alternative for scenarios in which they cause too much overhead.

Future Work. With TPy, we provide basic modules to address common tasks. In the future, we aim to extend the set of built-in modules to cover a wider range of tasks. However, the purpose of the provided modules is to give a baseline that others can reuse in their code and build on. To this end, we make the sources of TPy as well as a basic set of modules publicly available [18].

7 CONCLUSION

With the increased complexity of new solutions, protocols, and applications for networked systems, we need mechanisms to validate these proposals experimentally. Traditional testbed management and experimentation frameworks do not account for the requirements of *agile* deployments which often have a short lifetime and, thus, would not amortize the costs of setting up a full-fledged management solution. To provide flexibility of deployment and simplicity of use, we propose an experimentation framework with a minimalistic yet powerful design written in Python called TPy. Users can extend the framework with re-usable plug-and-play modules, enabling different flavors of experimental setups. As examples, we describe experiments in millimeter-wave and secure multihop communications that use TPy for the complete "setup–run–collect–evaluate–publish" cycle. Finally, we make TPy available as open source software [18].

ACKNOWLEDGMENTS

This work is funded by the LOEWE initiative (Hesse, Germany) within the NICER project and by the German Federal Ministry of Education and Research (BMBF) and the State of Hesse within CRISP-DA.

REFERENCES

[1] ACM Conference on Security and Privacy in Wireless and Mobile Networks. 2017. Reproducibility Label. Retrieved July 4, 2018 from http://wisec2017.ccs.neu.edu/reproducibility.html
[2] ACM MobiCom. 2017. Call for Papers. Retrieved July 4, 2018 from https://www.sigmobile.org/mobicom/2017/cfp.php
[3] Alessio Botta, Alberto Dainotti, and Antonio Pescapè. 2012. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks* 56, 15 (2012), 3531–3547.
[4] Ioannis Chatzigiannakis, Stefan Fischer, Christos Koninis, Georgios Mylonas, and Dennis Pfisterer. 2010. WISEBED: An Open Large-Scale Wireless Sensor Network Testbed. In *Sensor Applications, Experimentation, and Logistics*. Springer Berlin Heidelberg, 68–87.
[5] Irmen de Jong. 2018. Pyro 4.x - Python Remote Objects. https://github.com/irmen/Pyro4
[6] PC Engines. 2018. APU platform. Retrieved June 27, 2018 from http://www.pcengines.ch/apu.htm
[7] Future Internet Testing Facility (FIT). 2018. IoT-LAB: a very large scale open testbed. Retrieved June 27, 2018 from https://www.iot-lab.info
[8] GENI (Global Environment for Network Innovations). 2018. Federated Testbeds. Retrieved June 27, 2018 from http://www.geni.net/geni-partners/federated-testbeds/
[9] John D. Hunter. 2007. Matplotlib: A 2D Graphics Environment. *Computing in Science Engineering* 9, 3 (2007), 90–95.
[10] imec. 2018. jFed is a Java-based framework for testbed federation. Retrieved June 27, 2018 from https://jfed.ilabt.imec.be
[11] IMEC iLab.t. 2018. w-iLab.t. Retrieved June 27, 2018 from https://doc.ilabt.imec.be/ilabt-documentation/wilabfacility.html
[12] Institute of Telematics, University of Lübeck. 2018. Testbed Runtime. Retrieved June 27, 2018 from https://github.com/itm/testbed-runtime
[13] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click Modular Router. *ACM Transactions on Computer Systems* 18, 3 (Aug. 2000), 263–297.
[14] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Python in Science Conference*. 51 – 56.
[15] ORBIT Project. 2018. Open-Access Research Testbed for Next-Generation Wireless Networks (ORBIT). Retrieved June 27, 2018 from http://www.orbit-lab.org
[16] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. 2010. OMF: A Control and Management Framework for Networking Testbeds. *SIGOPS Oper. Syst. Rev.* 43, 4 (Jan. 2010), 54–59.
[17] Milan Schmittner, Arash Asadi, and Matthias Hollick. 2017. SEMUD: Secure Multi-hop Device-to-Device Communication for 5G Public Safety Networks. In *IFIP Networking Conference and Workshops*. IEEE.
[18] Daniel Steinmetzer and Milan Stute. 2018. TPy Implementation. https://seemoo.de/tpy
[19] Daniel Steinmetzer, Daniel Wegemer, and Matthias Hollick. 2017. Talon Tools: The Framework for Practical IEEE 802.11ad Research. https://seemoo.de/talon-tools
[20] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 255–270.
[21] Wraith Wireless. 2018. PyRIC: Python Radio Interface Controller. https://github.com/wraith-wireless/PyRIC
[22] Tatu Ylonen and Chris Lonvick. 2006. The Secure Shell (SSH) Connection Protocol. *RFC* 4254 (Jan. 2006).