Kölner Beiträge zur technischen Informatik
Cologne Contributions to Computer Engineering
Band 1/2022

Ergebnisse des Workshops vom 26.11.2021 des Forschungsschwerpunkts Vernetzte intelligente Infrastrukturen und mobile Systeme (VIMS)

Technology
Arts Sciences
TH Köln

Impressum:

Forschungsschwerpunkt Vernetzte intelligente Infrastrukturen und mobile Systeme
Technische Hochschule Köln
Fakultät für Informations-, Medien- und Elektrotechnik
Betzdorfer Str. 2
D-50679 Köln
tobias.krawutschke@th-koeln.de
Stand: März 2022
ISSN: 2193-570X

Inhaltsverzeichnis

Editorial **4**

Analyzing the latency of a LXI device using a Raspberry Pi (Theodor Fischer) **5**

Vulnerability Detection in Source Code Using Machine Learning and Graph Based Representations (Feras Zaher Alnaem et. al.) **12**

Design and Evaluation of a Quality Measurement Framework for News and Broadcasting Applications (Lukas Bluhmki) **20**

Editorial

Im Jahr 2021 ist der Forschungsschwerpunkt VIMS aus dem Vorgängerschwerpunkt VMA hervorgegangen. Mit diesem Band wird die Veröffentlichung in der Reihe *Kölner Beiträge zur technischen Informatik* fortgesetzt. Die Fakultät für Informations-, Medien- und Elektrotechnik am Institut für Nachrichtentechnik ermöglicht Master Studierenden nicht nur aus dem Bereich der technischen Informatik eine Möglichkeit Ihre Forschung zu veröffentlichen, die im Rahmen von Forschungs- und Entwicklungsprojekten an der TH Köln und/oder bei Projektpartnern entstand.

Ziel ist es, die Ergebnisse laufender Arbeiten aus den Forschungs- und Entwicklungsaktivitäten des Forschungsschwerpunkts nach außen zu kommunizieren und Informatiker und Informationstechniker außerhalb des Forschungsschwerpunkts z.B. aus dem Kölner Raum einzuladen, neue Ergebnisse aus Wissenschaft und technischer Anwendung im Rahmen von *Cologne Open Science* zu publizieren.

Der Herausgeberkreis freut sich, diesen neuen Band der Reihe der Fachöffentlichkeit zur kritischen Prüfung und zur möglichen Mitwirkung vorlegen zu können.

Rainer Bartz
Andreas Behrend
Andreas Grebe
Tobias Krawutschke
Hans W. Nissen
Beate Rhein
René Wörzberger
Chunrong Yuan

Analyzing the latency of a LXI device using a Raspberry Pi

Theodor Fischer
Technische Informatik (Master)
T.H. Köln
Cologne, Germany
theodor.fischer@smail.th-koeln.de

***Abstract*—The work in this paper focuses on determining the performance or rather the latency of a LXI capable power supply. Additionally, different VISA Libraries and the communication protocols of the LXI standard VXI-11, HiSLIP and raw TCP sockets are compared. A Raspberry Pi system combined with an additional microcontroller to provide ADC capability is used to conduct the measurements. The results of the tests show that the created system is capable of adequately analyzing the power supplies latency. Furthermore, the tests show that in this use case the choice of VISA Library or communication protocol results in no significant difference in the delay for a requested voltage.**

***Keywords*—LXI, performance analysis, latency, VISA, VXI-11, HiSLIP, Raspberry Pi**

I. Introduction

The LXI standard defines many different protocols, each with its supposed advantages and disadvantages [1]. To be able to use these protocols from an operating system a Virtual instrument software architecture (VISA) library is commonly needed. VISA is an I/O API widely used in the test and measurement (T&M) industry and is a standard implemented by many T&M companies like Rohde&Schwarz, Tektronix, National Instruments and Keysight Technologies [2]. Nearly all those T&M device manufacturers provide their own VISA library, which must all follow the official VISA specifications [3]. The work in this paper focuses on determining the performance differences of the different protocols and different VISA libraries by analyzing a common use case with a LXI complainant power supply.

II. Research Goals

The main goal of this project was to analyze the performance of the LXI power supply and foremost to determine the delay of requesting a new voltage level. To test this a series of experiments were devised. Before conducting the tests, a suitable test environment had to be created. To provide reliable experiment results every part of the created test environment should be carefully analyzed to detect potential points where delays could occur. A further goal in this project was to analyze if different VISA libraries or the available LXI communication protocols like VXI-11, HiSLIP or raw TCP sockets could make a difference in the delay tests.

III. Test Environment

A. Setup

The main device used in this project is a Raspberry Pi 4, which handles the Ethernet Communication with the Rohde&Schwarz NGE100B power supply, and which collects the experiment test results. Additionally, a NXP K64 microcontroller is used to read the (analogue) voltage on the channel output of the LXI power supply. The voltage level is read via an ADC (Analog-to-Digital converter) on the K64 microcontroller and then transmitted to the Raspberry PI using SPI (Serial Peripheral Interface). The power supply can deliver up to 32.2V, what would have been too high for the K64 microcontroller (3.3V) to handle. So, a voltage divider was made up and used to bring the voltage down to a maximum of 3.3V. An overview of the hardware setup is depicted in Fig. 1.

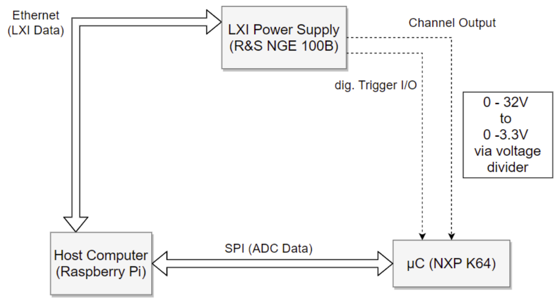


Figure 1. Depiction of the general hardware setup

Due to the inability for the Raspberry Pi to be used as a SPI Slave it was set as SPI Master and the NXP K64 was set as SPI Slave [4]. This resulted in the disadvantage that to receive new ADC data the SPI interface must be constantly polled by the Raspberry Pi. It also required extra effort to ensure that the NXP K64 was always ready to transmit, because new SPI transfer requests could arrive at any time. The SPI data structure chosen for this setup consists of 5 Byte per transmission and is depicted in Fig. 2. To ensure data integrity during transmission a 16-bit CRC error detecting code was added to the structure.

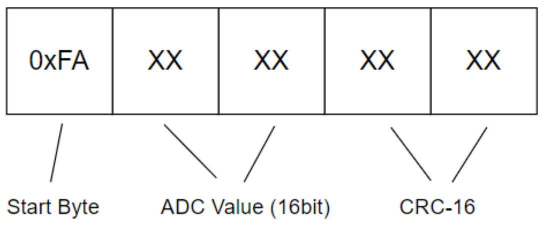


Figure 2. SPI data structure

All investigations were made with self-written Python programs. All diagrams and plots were created using the Python package matplotlib [5]. Boxplots are defined as [6]:

- Median: Q2 (50th percentile): middle value of dataset
- First Quartile: Q1 (25th percentile): median of the lower half of the dataset
- Third Quartile: Q3(75th percentile): median of the upper half of the dataset
- Interquartile Range: IQR: distance between upper and lower quartiles (Q3 – Q1)
- Whisker Maximum: Q3 + 1.5 * IQR: upper range of values without outliers
- Whisker Minimum: Q1 – 1.5 * IQR: lower range of value without outliers

Due to using a Raspberry Pi and Python as the programming language the choice of compatible VISA libraries was limited to only the official VISA Library for the power supply from R&S and a third-party library from the open-source python package PyVisa-Py.

B. *Preliminary Investigation*

To provide accurate and reliable data resulting from the main experiments some preliminary work had to be done to analyze the performance or rather the latency of the individual components in the test system.

1) Examining SPI & ADC Performance

The first step involved measuring the SPI, and therefore also the ADC, performance between the Raspberry Pi and the K64 microcontroller. A simple setup was created where a GPIO pin of the Raspberry Pi was configured as output and connected to the ADC input of the microcontroller. This setup is depicted in Fig. 3.

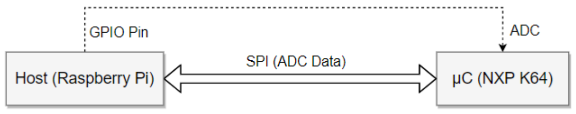


Figure 3. Depiction of simple ADC performance test setup

To minimize delays the microcontroller application was created with minimal overhead and complexity in mind. It continuously samples the ADC via an interrupt and stores the value in a global buffer. The main application loop retrieves this global value and waits indefinitely for the SPI master, the Raspberry Pi, to start the SPI transaction.

In its standard configuration the K64's ADC produced noisy and therefore unprecise data. To combat this a higher clock speed as well as a hardware averaging feature of the ADC was activated. This setting was set to use 32 ADC samples to generate the final averaged output sample. Additionally, the so-called "Long sample mode" was also activated and set to use 20 extra ADCK cycles. Setting this mode changes how long the ADC waits before sampling a signal. The drawback to activating these features is an increase in the conversion time and therefore delay of new ADC data. To reduce this delay the "High-speed mode" of the ADC was also activated. It configures the ADC to use an alternative conversion sequence and allows for lower conversion times.

The resulting ADC data and corresponding delay measurements before the ADC configuration changes can be seen in Figure 4. The results after the changes are shown in Fig. 5. The delay measurements were done in multiple runs and the resulting data is presented as combination of three subplots. Two plots show the voltage curve of a single run and the bottom one is zoomed in closer to the relevant signal part. The subplot in the top right shows the distribution of the delay values of all 100 runs. Changing the ADC settings resulted in a delay increase of about 200µs to 400µs depending on the request position (3.3V or 0V). At the same time, the accuracy of the sampled ADC data was greatly increased. This tradeoff was accepted, because while the delay is higher than before it is still mostly below 1ms, and as later tests will show, well below other delay factors in the system.

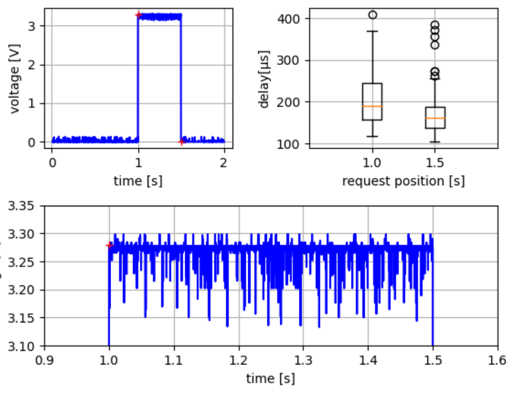


Figure 4. ADC performance & delay test (default settings)

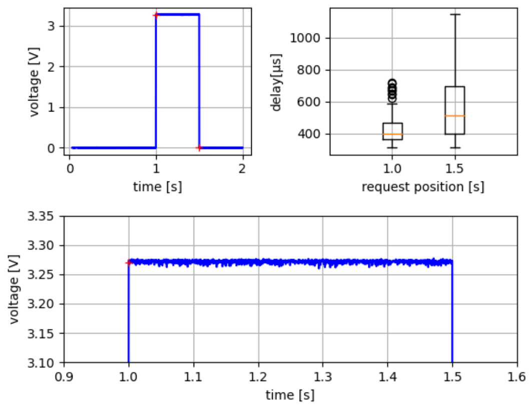


Figure 5. ADC performance & delay test (optimized settings)

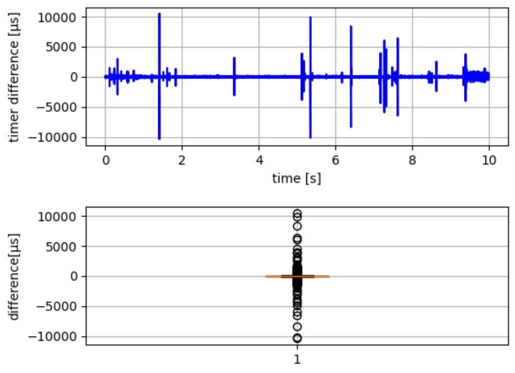


Figure 6. Time difference test (high background system load)

2) *Examining Timing Accuracy*

The standard Raspbian operating system with the standard kernel is not a real time system, which results in a certain unpredictability and variance in timing relevant measurements. And as all timing measurements for the experiments were to be done directly on the Raspberry Pi, this timing variance would influence the final results. The function “perf_counter” of the python “time” package was used for all python time measurements as it provides a high-resolution timer value [7]. A simple test setup was devised to measure the timing variance with the help of the NXP K64 microcontroller. The microcontroller was set up with a 100 µs interrupt-based timer. The python program would collect this timer via the SPI connection, as it is the SPI Master. This would be repeated in a loop while at the same time the value of the “perf_counter” function would be taken. It would then be calculated how much time passed for both timer values since the last loop. These resulting two values should then, if both timers ran at the same speed, be the same. If one timer has a higher difference it would mean that one timer is running slower than the other.

Different system loads were put on the Raspberry Pi while the test was running to check if timer differences could be provoked. The result for a high system load test is shown in Fig. 6. The results are each divided in to two subplots, where the upper one shows the timer differences across the whole test run. The lower subplot shows the distribution of the timer difference values. In the high system load test the background task consisted of watching a video in an internet browser in the background while the test was executing. The data shows high spikes throughout, but foremost at the beginning and near the end of, the test. The spikes reach up to 10ms of time difference between the python and microcontroller application. Figure 7 shows a test with no, or very little, background system load (Note the different Y Axis scaling). Apart from a single spike of about 1.5ms close to the beginning of the test there are no other significant points of interest. Comparing these two results shows clearly what an impact background processes and therefore high system load can have on time crucial measurements.

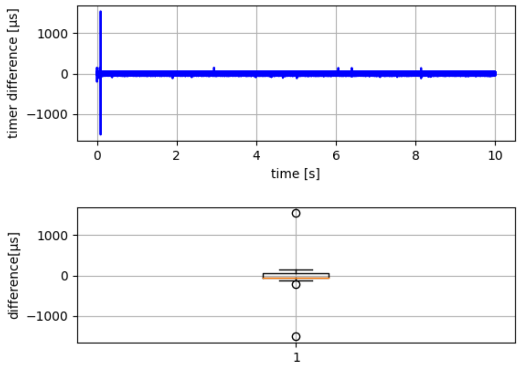


Figure 7. Time difference test (low background system load)

In a second test the absolute timing accuracy between the two timers were measured. The test was run for 20 seconds, and the results showed a deviation of 50ms between the two timers. With extrapolation this would result in a deviation of 1 second every 400 seconds.

Together these two tests showed that the operating system can have a great impact on time measurements done with the “perf_counter” function. But given a low system load with minimal background processes it can be said that the “perf_counter” function can be used to measure timing with sub-millisecond precision with a small chance of measurements glitches.

3) *Examining Ethernet Delay*

To complete the test environment analysis the simple ethernet connection between the Raspberry Pi and the LXI power supply was also tested. The two devices are directly connected with each other. A simple ping test was used, because only the minimum amount of delay was of interest in this connection. Bandwidth was not of interest in this project. The test was conducted multiple times to produce reliable data and the corresponding boxplot can be seen in Figure 8.

The median of the resulting data is at about 210µs and the data is very closely packed together with the max/min whisker distance only being about 25µs. There are only a few data points outside of this range, which, with a repetition count of 200, suggests a reliable ping delay.

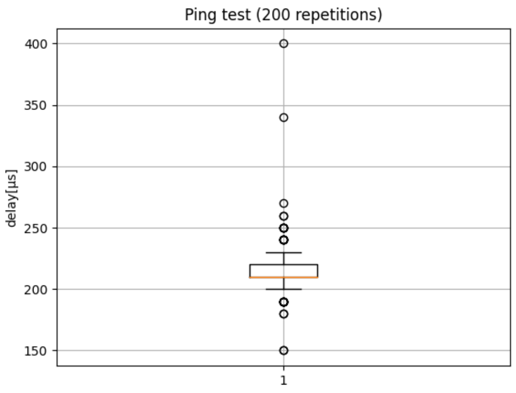


Figure 8. Results of the ping test

IV. LXI EXPERIMENTS

A. *Digital I/O Delay*

The R&S NGE100 series power supply provides four digital input and output trigger pins. When set up as output, a pin can be configured to enable the output when a certain event occurs, or a condition is fulfilled within the power supply. There are multiple events available, for example a fuse trip event or a current level event, which occurs when a preset current level is exceeded. When set up as an input, an outside device can trigger a pin and, for example, activate a channel. The configuration of these digital I/O pins can be done directly via the buttons on the device or also be set via LXI messages.

The goal of this experiment was to find and create a setup to test the minimal possible latency of the different communication protocols and VISA libraries. The I/O pins were deemed suitable for this task as they presumably provided faster switching properties in comparison to the voltage on the main channel outputs. A test setup was created where a I/O pin would be configured manually beforehand directly on the device. It would be configured as output and active low but with the channel state still deactivated. As a trigger condition no particular event was chosen. The output was connected to the ADC input of the K64 microcontroller. During the test a LXI message would be sent to the power supply with the instruction to activate the aforementioned channel. And due to the active low setting and also the trigger event not being active the output would then immediately turn to active. This would then be measured at the ADC input where it would be transmitted to the Raspberry Pi via SPI. The final delay would then be calculated and stored for later processing. This test was repeated 500 times for each configuration of protocol and VISA library. Figure 9 shows the results of all test configurations in the form of boxplots.

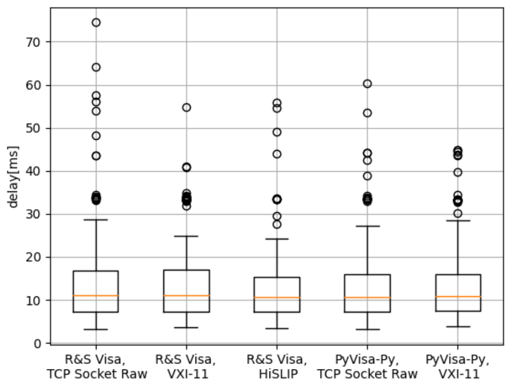


Figure 9. Results of the Digital I/O test (multiple configurations)

Overall, it can be said that the delay times between all configurations does not differ significantly and that the delay times are very low with a median of about 11ms.

To confirm the validity of these results an oscilloscope was used as a second measurement setup. For this the test was slightly modified. Before sending out the LXI message for the I/O pin channel activation a GPIO pin on the Raspberry Pi would be activated. Signal probes of the oscilloscope were hooked up to this GPIO pin and the digital output pin of the power supply. The test was then run and automatically repeated for 100 times. The configuration of R&S VISA library and VXI-11 connection protocol were used during this test. Using a function of the oscilloscope the average and standard deviation of the data was captured and calculated. The results are shown in Figure 10 and 11.

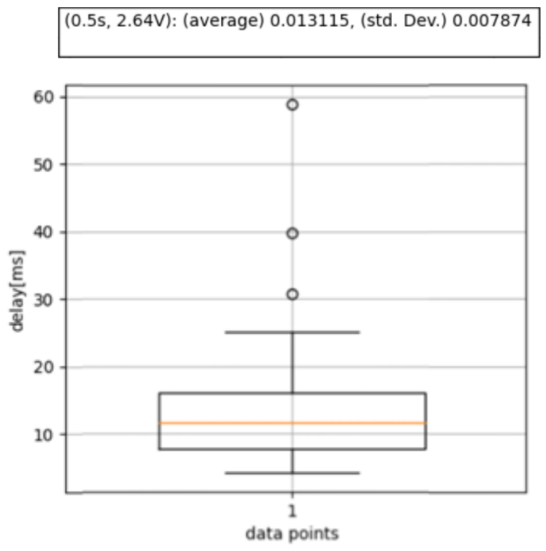
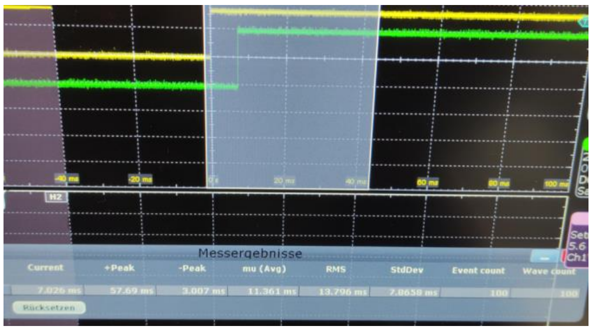


Figure 10. Digital I/O test validation, Raspberry Pi capture



Messergebnisse

Current	+Peak	-Peak	mu (Avg)	RMS	StdDev	Event count	Wave count
7.026 ms	57.69 ms	3.007 ms	11.361 ms	13.796 ms	7.8658 ms	100	100

Figure 11. Digital I/O test validation, oscilloscope capture

The results of the oscilloscope, in Figure 11, are shown in the table located near the bottom of the image. The relevant columns are “+Peak”, “-Peak”, “mu(Avg)” and “StdDev”. In comparison to the results of Fig. 10 the average is lower by 1,75ms but with practically the same standard deviations of 7,87ms. The highest peak of 57.69ms, seen by the oscilloscope, can also be seen in the Raspberry Pi measurement with a higher value of about ~58.5ms, which falls in line with the average offset between the two measurements. This shows that while the Raspberry Pi has higher overall measurement times it still provides reliable data. It also confirms that the peaks measured by the oscilloscope and the Raspberry Pi must originate in the power supply and not the testing equipment.

B. *Manual voltage request delay*

The main application of this power supply is to provide power via its three channel outputs. These channels can be controlled directly via the user interface on the device or by using LXI messages. In addition to the channel output state, the voltage level und maximum current level can be configured as well.

This experiment was supposed to be based on a real use case, where certain voltage levels would be requested for a certain time slot. It should be determined how fast the power supply can switch to a newly requested voltage and how the latency possibly differs across the whole voltage range. The experiment was set up as described in chapter III and depicted in Figure 1. The python test program was designed with the ability to take a variable test configuration as input. One entry in the test configuration consists of a voltage value Y and a time value X. This entry is used by the test program to determine at what time X, relative to the test starting time, the voltage Y should be requested from the power supply. The test configuration can contain any number of time-voltage entries. To increase the reliability of the data each test configuration was executed 100 times. Fig. 12 shows a code snipped of the part where the test configuration is defined.

```
loopCount = 100
voltMargin = 0.005 #0.5% margin
testSetupRunTime = 5.6 #seconds
testSetup = np.array([[1.0, 5.0],  # At 1 Second set to 5V
                      [1.8, 15.0], # 1.8 seconds, 15V
                      [2.4, 0.0],  # ...
                      [3.2, 25.0],
                      [4.0, 31.0],
                      [4.8, 0.0]])
```

Figure 12. Code snipped of the test configuration definition

The aforementioned test configuration was executed, and the results collected and processed. Fig. 13 shows the voltage curve with time slots and the boxplots of the delay for each combination of VISA Library and connection protocol. The results again show no clear difference between all the different configurations. It can, however, be seen that the voltage levels all produce different delays. So, that means when using the device, a consistent delay across different voltage levels cannot be assumed.

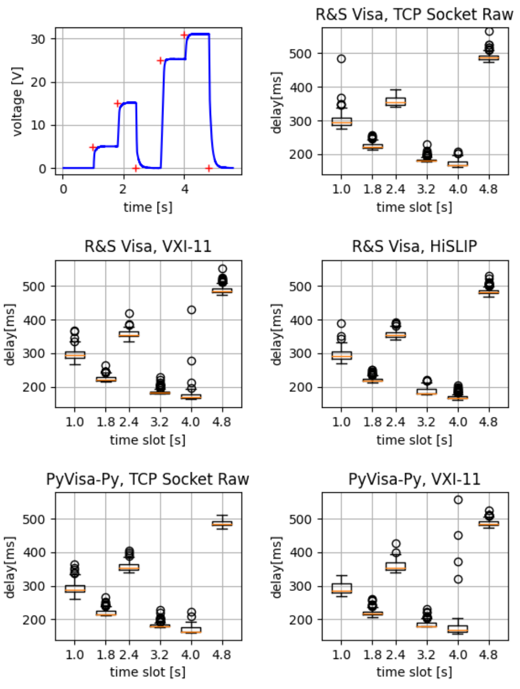


Figure 13. Manual voltage request test. All configurations

C. *Alternative voltage request: R&S EasyArb function*

The python test program provides the possibility of requesting a certain voltage at a certain time. However, if very fast voltage changes are requested in a short period of time the power supply begins to struggle. This can be seen in Figure 14 where a voltage change from 5V to 10V, and vice versa, was requested every 50ms. The requested voltages are not even reached fully in time before the next voltage is requested. The first assumption was that some sort of internal limit of the power supply was reached and that faster voltage transitions were not possible. The following test, however, disproved this assumption.

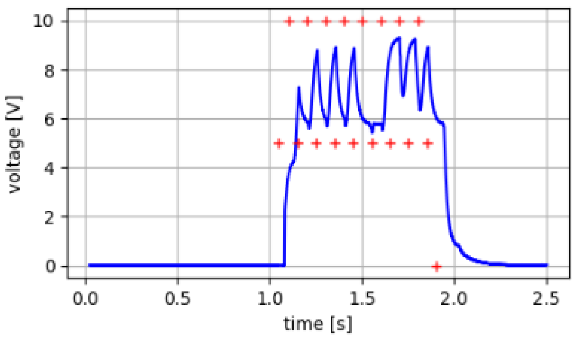


Figure 14. Voltage request with fast switching frequency. Limits of manual voltage request exposed.

In addition to the standard functions of a usual power supply the R&S NGE 100B it provides a few more advanced functions. One of the functions is called "EasyArb" and it provides the ability to define arbitrary waveforms. It takes as input a voltage level and the duration of how long this voltage level should be stayed at. Multiple datapoints can be defined for one waveform and the repetition count of the whole waveform can be configured as well. All these settings are fully controllable via LXI messages.

With a slight modification to the python test program from the previous experiment it was possible to use this feature and compare it to manual method. The same data points as previously mentioned were used for this test. Inspecting the voltage curve, depicted in Figure 15, one can see a much clearer curve compared to the one previously shown (Fig. 14). All requested values were clearly reached in time and with a small delay. This disproves the assumption that the device had reached some internal limit concerning the voltage curve generation. But it certainly highlights some kind of limitation when directly and quicky requesting voltage levels from the power supply.

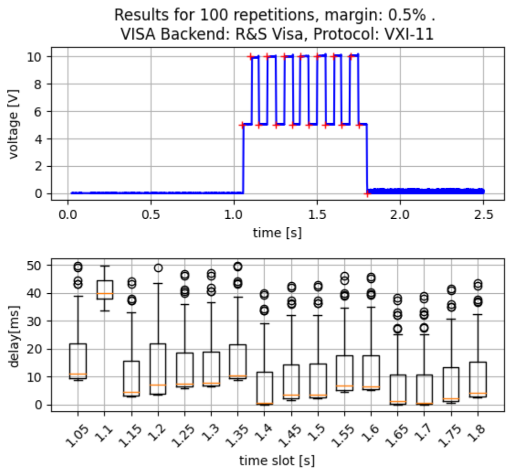


Figure 15. Voltage request with fast switching frequency using advanced "EasyArb" function.

Using the same values as in the first voltage request configuration in the second experiment (Fig. 12) it can be shown that even in a normal use case the overall delay is lower than when manually requesting a new voltage. This test was again carried out for all available combinations of VISA libraries and connection protocols. The results are shown in Figure 16. Comparing to the results from the previous experiment (Fig. 13) there is a clear reduction in delay across all the configurations visible. Also noteworthy is the sharper transitions in the voltage graph, especially from 2.4s (15V) to 3.2s (0V).

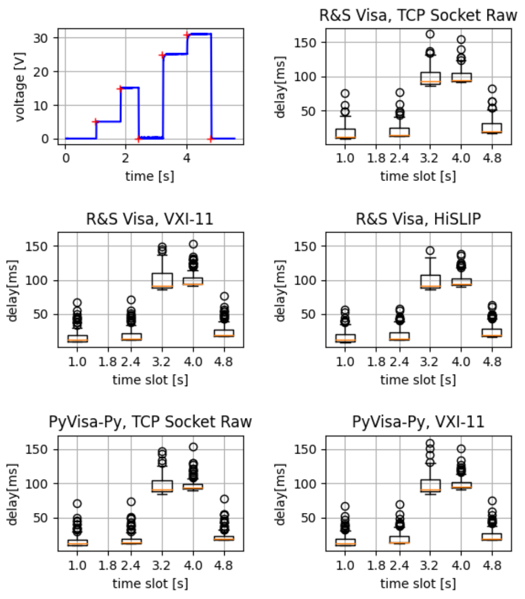


Figure 16. Voltage request using advanced "EasyArb" function.

V. Conclusion

The experiments in this project showed that for the use case of requesting voltage for a precise time slot the choice of VISA backend library or the underlying connection protocol is mostly irrelevant as most of the delay originates from the power supply itself. It could also be shown that, using some advanced functions of the LXI power supply, the delay of voltage requests can be significantly reduced. Furthermore, apart from the LXI testing, it was also shown that the Raspberry Pi, with an additional microcontroller for analog-to-digital signal conversion, can be set up to reliably measure time critical signals.

References

[1] LXI Standard, "LXI Protocols", Url: https://www.lxistandard.org/About/LXI-Protocols.aspx

[2] W. Cheng, F. Wang, H. Ma, „Remote Automatic Test System based on MATLAB using VISA over LAN", Shandong Academy of Sciences, 2015

[3] IVI Foundation, “The VISA Library”, Url: https://www.ivifoundation.org/downloads/Architecture%20Specifications/vpp43_2020-11-20.pdf

[4] Raspberry Pi Documentation, “SPI”, Url: https://www.raspberrypi.org/documentation/hardware/raspberrypi/spi/README.md

[5] Python Package, “matplotlib”, Url: https://matplotlib.org/

[6] J.W. Tukey, “Exploratory Data Analysis”, Princeton University and Bell Telephone Laboratories, 1977

[7] Python function, “time.perf_counter()”, Url: https://docs.python.org/3/library/time.html#time.perf_counter

Vulnerability Detection in Source Code Using Machine Learning and Graph Based Representations

1st Feras Zaher Alnaem
TH Köln - University of Applied Sciences
Cologne, Germany
feras.zaher_alnaem@smail.th-koeln.de

2nd Jonas Kaltenbach
inovex GmbH
München, Germany
jkaltenbach@inovex.de

3rd Marisa Mohr
inovex GmbH
Hamburg, Germany
mmohr@inovex.de

4th Chunrong Yuan
TH Köln - University of Applied Sciences
Cologne, Germany
chunrong.yuan@th-koeln.de

***Abstract*—Due to the increasing number of end-user devices and the complexity of modern software projects, organizations need to continuously expand and improve their cybersecurtiy measures. There is a growing interest in using algorithms as a software-based methodology for the development of trustworthy software systems. Particularly, Machine Learning (ML) algorithms have the potential for uncovering security defects, minimizing disruptions and reducing development costs, since they can be applied during the early lifecycle of software development, even before the code is executed. By integrating graph-based representations and data-driven ML models, we have developed in this work a new methodology for the detection of software vulnerabilities. We have proposed two approaches for the learning of graph embeddings from source code. In the first approach, each graph is represented as a fixed length feature vector. Using this feature vector as input, classical ML models such as Support Vector Machine (SVM) and Multi-Layer Perceptron (MLP) have been applied for the classification of the code as vulnerable and non-vulnerable. The second approach is based on the use of Graph Neural Networks, which provide an end-to-end capability for node-, edge- and graph-level embedding and predictions. It has been shown through comparative study that both approaches achieve state-of-the-art performance, with the first having a better generalization capability and the second having a better accuracy in vulnerability detection.**

***Index Terms*—machine learning, graph representation, graph neural network, source code analysis, vulnerability detection**

I. Introduction

It is no longer possible to imagine our daily lives without software systems, which are used in various areas such as healthcare, energy supply, transport and, in general, all devices connected to the internet. Developing software systems is costly as software engineers have to handle the complexity of developing a software and deliver highly functional software products on schedule while at the same time avoiding bugs and vulnerabilities. A software vulnerability is a fault or weakness in the design, implementation or operation of a software which can be exploited by a threat actor, such as a hacker, leading to unauthorized actions on a system [1]. Security vulnerabilities often arise during the coding stage of the software development life cycle [2]. Thus, it is important to identify vulnerable components in the early stages of software development, as the cost for finding and fixing errors increases dramatically as the development progresses. Achieving early identification of vulnerable components can thus help programmers build highly secure and robust software products from the ground up [3].

In order to prevent security breaches and detect vulnerabilities in source code during development, different analysis techniques can be applied. Both static and dynamic analyzers are able to uncover common vulnerabilities in software [2]. Static analyzers examine software by taking into consideration its form, structure, content, or documentation without the execution of programs. Dynamic analyzers repeatedly execute programs with many test inputs on real or virtual processors to identify weaknesses. However, both static and dynamic analyzers are rule-based tools and thus limited to their hand-engineered rules. They are not able to guarantee full test coverage of code bases [4]. Moreover, these tools can not handle incomplete or incorrect information very well – data that does not have an associated rule can be ignored [5].

Alternatively, models and algorithms from the field of Artificial Intelligence (AI), especially Machine Learning (ML), can be used for this purpose. ML techniques can be used both for the intelligent analysis of huge amounts of data and for the automation of different application processes [6]. A useful property of many data-driven AI systems lies in the automatic extraction of relevant features which are not obvious to humans. For example, deep learning techniques have been proved to be particularly useful for extracting relevant features together with the building of effective data-driven models. Currently, many cybersecurity companies try to build advanced ML solutions for early threat detection. For example, ML methods have been successful applied for the prevention of personal information leakage [7]. A recent survey on the ML based vulnerability analysis can be found in [8].

The problem to be solved in this work is the detection of vulnerabilities in source code using ML. In order to develop a data-driven model, we need a proper representation of the source code. In this work, we focus on the study of graph-based representations of source code and their ability in handling errors in the syntax, structure and semantics of computer programs. We have studied two different graph embedding approaches. The first approach makes use of graph2vec for creating graph embedding vectors. The generated graph embeddings are then classified by models such as SVM or MLP. The second approach is based on the use of Graph Neural Networks (GNNs) for both graph-level embedding and vulnerability classification.

The rest of the paper is organized as follows. In Section II, a review of some related noteworthy works is provided. Then, in Section III, the project pipeline is presented, with detail explanation and description of the proposed approaches. Section IV demonstrates the experiments carried out for the evaluation of different approaches. Finally, Section V summarizes the whole paper.

II. Related work

In the fields of vulnerability detection and program analysis, there exist various methods for the representation of source code. One possibility is based on the extraction of software metrics. Most commonly used metrics are size of the code, complexity, code churn, developer activity and number of line changes during code reviews, etc. For example, in [9]–[11], the authors use manually designed metrics obtained from source code and development history to predict whether a code base is vulnerable. Although these metrics can be used successfully to detect vulnerabilities, they do have some problems. For example, two pieces of source code may have the same metric, such as complexity, but have different behavior, resulting in misclassification. In addition, these metrics cannot adequately capture both the semantic and syntactic representation of the source code, which is an important prerequisite for building accurate predictive models [12].

Alternative approaches that are based on text mining can be found in [4], [12], [13], where text mining and deep learning approaches have been used for the automation of threat detection. In these studies, source code are treated as plain text sequences, and the potential use of natural language processing (NLP) techniques has been explored for the purpose of vulnerability detection. These approaches are based on the assertion that programming languages have predictable statistical properties that can be captured in statistical language models. However, they have limitations in the expression of logic and structures in source code.

In order to be able to capture semantic information in source code, graph-based representations have been investigated in [5], [14], [15]. Each source code fragment is transformed into a graph model, e.g., in the form of an abstract syntax tree (AST), control flow graph (CFG), data dependency graph (DDG), or code property graph (CPG). For instance, the authors in [5] extract and convert the AST of a given source code fragment into a binary AST and then into a numerical array. This approach preserves structural and semantic information contained in the source code. Based on this array representation and by using a Convolutional Neural Network (CNN), vulnerability prediction has been carried out.

III. approach

Since graph representations are useful for capturing comprehensive program semantics, we decide to use representations based on AST. Similarly to [5], we extract AST from each piece of program code. However, different to [5], which generates hand-crafted features by converting graphs into numerical arrays, we use embedding methods such as graph2vec and GNN that are capable of generating automatically learned features [16]

Fig. 1 visualizes the entire pipeline of our approach, including both the learning of graph embedding and vulnerability detection. First, source codes are taken as input. After a data processing step, ASTs are extracted for representing source codes as graphs. In order to generate graph embeddings from the extracted ASTs and classify them, two approaches are considered. The first approach is based on the use of a transductive method[1], namely graph2vec, which takes a graph as input and transforms it into a lower dimension embedding vector. The vector is then classified by ML models such as SVM and MLP. Both act as binary classify and output a prediction label, with 1 for vulnerable and 0 for non-vulnerable code. The second approach is based on the use of an inductive method [2], namely GNN for the generation of graph-level embeddings as well as the binary classification.

A. Data Preprocessing

Data preprocessing is necessary for the analysis of source code samples existed in a large multi-class dataset and make proper preparations before using them for the training of ML models. Since there exists different types of software vulnerabilities and their frequencies of occurrences are different, the numbers of available samples in the different vulnerability types are varying. This means, the dataset is imbalanced. An ML model trained with imbalanced dataset has a biased behavior, performing better for classes with more samples and worse for classes with fewer samples. There are various approaches to deal with this problem, e.g. via undersampling, oversampling or generating synthetic data [17]. We use undersampling to create balanced datasets of vulnerable and non-vulnerable samples. The basic idea of it is to randomly eliminate instances of the majority class so as to match the number of instances with the minority class.

B. Extracting AST

The AST is a tree type data structure that represents the semantic structure of a source code written in a formal language. Each node of the tree denotes a construct occurring in the text [18]. By using a parser called Clang available in the clang index library (Cindex) [19], one can generate ASTs from source codes.

[1]Transductive methods compute embeddings for a fixed set of graphs in form of a lookup table or an embedding matrix. From a deep learning perspective, these embedding methods are "shallow" in that they are not deep models and optimise the output vectors directly [16].

[2]Inductive methods are more complex models that represent functions of neighborhood aggregation to embed arbitrary graphs without the need of reducing dimensionality or changing the structure of the graphs [16].

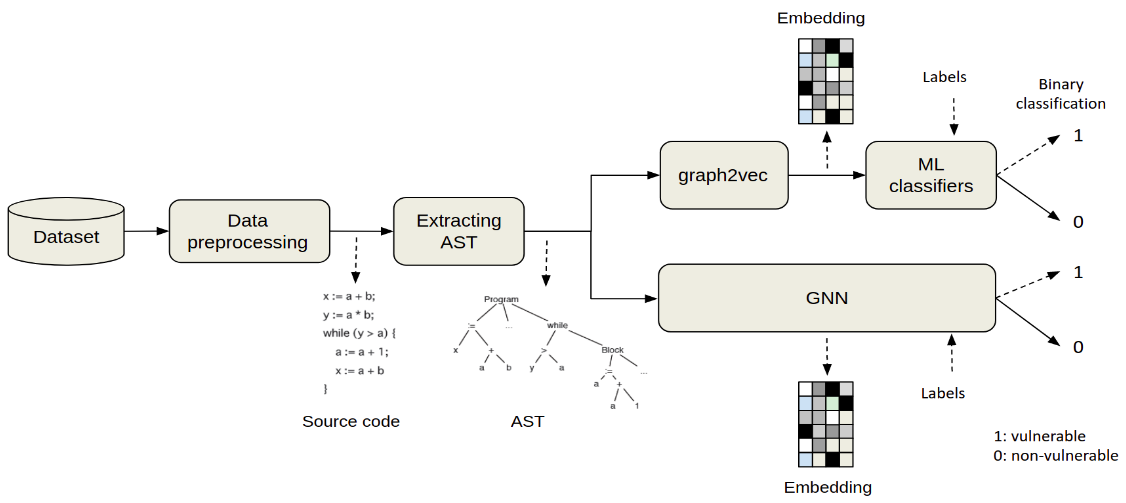


Fig. 1. Project pipeline

Listing 1 shows an example of source code taken from the Draper VDISC dataset [4]. The first three lines of it are converted into a sequence of tokens, as is shown in Listing 2.

Listing 1. A source code from Draper VDISC dataset

```
GetFileType(gpointer handle)
{
    WapiHanldeType type;
    if(!_WAPI_PRIVATE_HAVE_SLOT (handle)) {
        setLastError (Error_INVALID_HANDLE);
        return(FILE_TYPE_UNKNOWN);
    }
    type = _wapi_handle_type (handle);
    if (io_ops[type].getfiletype == NULL) {
        SetLastError (ERROR_INVALID_HANDLE);
        return(FILE_TYPE_UNKOWN);
    }
    return(io_ops[type].getfiletype ());
}
```

Listing 2. Example of a tokenized source code

```
"GetFileType": IDENTIFIER
"(": PUNCTUATION
"gpointer": IDENTIFIER
"handle": IDENTIFIER
")": PUNCTUATION
"{": PUNCTUATION
"WapiHandleType": IDENTIFIER
"type": IDENTIFIER
";": PUNCTUATION
```

After this step, the tokens are converted into a data structure that yields an AST representation of the given source code, as is shown in Fig. 2. Here, each graph node stands for a corresponding functional construct existed in the C programming language. It starts with the name of the code and then defines a function declaration ”FUNCTION_DECL” , which is the main function of that code sample. This main function has two children, the first one is a parameter declaration ”PARM_DECL” and the second one is a compound statement ”COMPOUND_STMT”, which is a combination

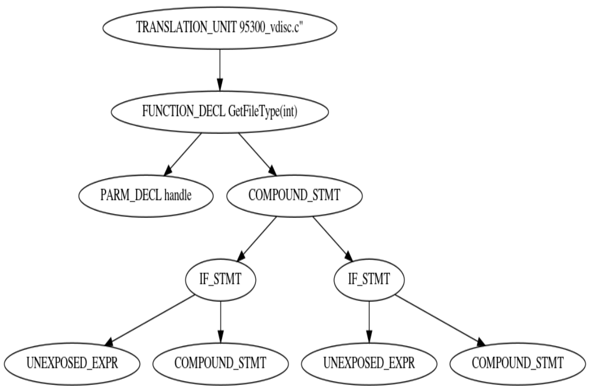


Fig. 2. An extracted AST using Clang

of two or more simple statements. IF statements are denoted in this AST as ”IF_STMT”. Unexposed expressions ”UNEXPOSED_EXPR” refer to expressions whose location, information, and children could be extracted, but the specific kind operations is not exposed.

A further step is to obtain structural features from each extracted AST, i.e., information about connections, indices and degrees of each node in the graph. Here, the degree indicates the depth of the node, i.e., the number of children it has. The format for describing these structural features is shown in Listing 3.

Listing 3. Structural representation of AST

```
edge= [start node, destination node]

feature= {"node index": degree}
```

For the AST shown in Fig. 2, its final AST with structural representation is visualized in Listing 4:

Listing 4. The final structure of an extracted AST of a source code

```
{"edges": [[1, 2], [2, 3], [2, 4], [4, 5], [5, 6],
[5, 7], [4, 8], [8, 9], [8, 10]],
"features": {"1": 1, "2": 2, "3": 0, "4": 2, "5": 2,
"6": 0, "7": 0, "8": 2, "9": 0, "10": 0}}
```

C. *graph2vec*

Graph2vec is a neural embedding framework proposed by Narayanan et al. in [20], where graph embeddings are learnt in an unsupervised manner. The learning method is data-driven. Hence it requires a fairly large set of graphs to learn graph representations. It has the ability to capture structural similarity, i.e., vectors representing structurally similar graphs lie closely in space.

In order to build embedding vectors, the graph2vec model of [21] has been used. It takes input files in JSON format. Each file represents a graph and consists of two keys. The first key is called ”edges” which corresponds to the edge list of the graph, while the second key is ”features” which corresponds to the node features. In this way, each file is organized in a similar way as our structural AST representations, as is shown in Listing 4.

Using a training dataset, the graph2vec model is hence able to learn a vector embedding for each AST graph. This means, each AST graph can be represented as a corresponding embedding vector of size 128.

D. *ML Classifiers*

The generated embedding vectors can hence be classified by an ML model for binary classification. Two models have been studied. The first model is a linear SVM , which can deliver a ”best fit” decision to provided data [22]. The sklearn library is used to build and train a linear support vector classifier with a default regularization parameter $C = 1$ and a standard SVM loss function called ”hinge”.

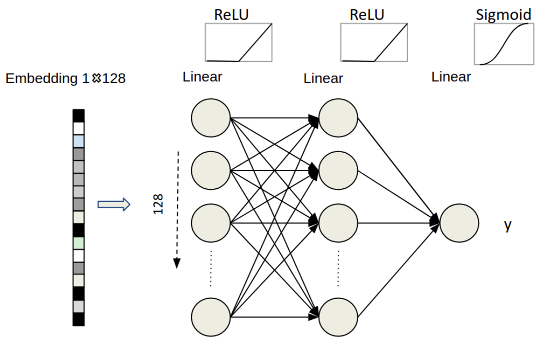


Fig. 3. MLP model

The second model is an MLP, which is an artificial neural network trained with back-propagation. The architecture of our MLP model is shown in Fig. 3. Training of the MLP is done by using the PyTorch library. It contains three layers. The input layer has 128 neurons which can take an embedding vector with the size of 128. The hidden lay has a similar structure, also with 128 neurons. The output lay has a single neuron. The activation functions used are ReLU in the first two layers and Sigmoid in the last layer.

E. *GNN*

Alternative to the graph2vec method, we have applied a state-of-the-art GNN framework for node-, edge- and graph-level embedding and prediction. As a deep learning model, it uses a form of neural message passing in which vector messages are exchanged between nodes [23]. Our GNN model is built using the PyTorch Geometric library. Its architecture can be seen in Fig. 4.

The GNN takes directly a graph as input and computes its node-level embedding with an embedding size of 128. It consists of three graph convolutional layers [24] with ReLU as the activation function. In this way, each node can learn features from its three-hop neighborhood and creates its embedding vector. In order to create embedding at the graph-level, two pooling layers are applied. The first layer is a global mean pool layer, which returns graph-level-outputs by averaging node features across the graph. This means, for a single graph G_i, its output is computed as follows:

$$r_i = \frac{1}{N_i}\sum_{n=1}^{N_i} X_n, \tag{1}$$

where X is the node feature matrix and N represents the number of nodes.

The second layer is a global max pool layer, which returns graph-level-outputs by taking the channel-wise maximum across the node dimension. This means, for a single graph, its output is computed as follows:

$$r_i = max_{n=1}^{N_i} X_n. \tag{2}$$

As the output of the above illustrated embedding process, an embedding vector of size 256 is created. For the purpose of performing vulnerability detection based on this embedding vector, the GNN model attaches three full-connected layers on its top and they act as a binary classifier. This classifier has the same structure of the MLP model described in Section III-D. The only difference is that its input size is 256. During the back-propagation process, all embeddings as well as the weights of upper layers in the GNN model are trained toward achieving the best prediction results on the training dataset.

For the training of both the GNN and MLP models, we use the Binary Cross Entropy (BCE) as loss function and the Adam optimizer is initialized with a learning rate of 0.01.

IV. Experimental Evaluations

For the experimental evaluation of the proposed approaches, we use a public dataset called Draper VDISC [4], which contains 1.27 million of synthetic and real function-level samples of C and C++ code mined from open source software. The samples have been labelled via static analysis for potential vulnerabilities. The dataset contains various vulnerability types

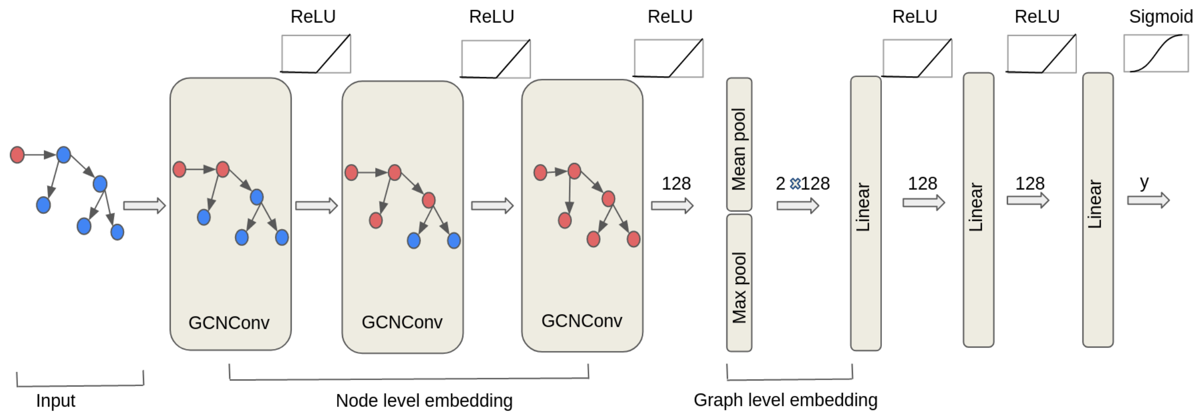


Fig. 4. GNN model

defined by CWE (Common Weakness Enumeration). Based on the CWE types, we split the whole dataset into five groups. Table I shows the distributions of samples in all five groups of CWE vulnerabilities.

As can be observed from the table, the Draper VDISC dataset is highly imbalanced as the number of positive (i.e. vulnerable) samples is far less than the number of negative (non-vulnerable) samples due to the fact that they are collected from real projects. Using the undersampling method described in Section III-A, this problem is fixed by making two configurations. One of them is to create per-vulnerability models to be trained only on samples containing the same vulnerability type so as to measure the detectability of different vulnerability types in a consistent manner. The second configuration is to create a new category (named as CWE-COMBINED) which contains vulnerable samples of all vulnerability types and non-vulnerable samples of any vulnerability type. In this setting, a sample code will be classified as being vulnerable or not without differentiating the exact type of vulnerability.

As a consequence, we have now arrived at six balanced datasets. They are CWE-119, CWE-120, CWE-469, CWE-476, CWE-OTHERS, CWE-COMBINED, with each having the same number of positive and negative samples. For example, the CWE-120 category contains 47660 ”vulnerable” and the same amount ”non-vulnerable” samples after performing under-sampling. Table II shows the amount of samples belonging to each of the six datasets after performing under-sampling.

For the training of the MLP and GNN models, each dataset has been divided into three disjoint subsets for training, validation and test purpose. The distribution is 80% for training, 10% for validation, 10% for test.

We use the learning curve as an estimate of the generalization capability of a model. During the training process, we apply a form of early-stopping [22]. The purpose is to prevent over-fitting. If there is no improvement in the validation loss after five epochs, training will be stopped.

Fig. 5 shows the learning curves resulted from the training of MLP and GNN on the balanced CWE-120 dataset. As can be observed, for the MLP model, both training loss and validation loss start at high values ($train\_loss = 0.694$, $val\_loss = 0.685$) and go down until the validation loss starts to increase again and does not improve anymore (at the 50th epoch). So the training stops at 55th epoch. As contrast, the GNN model tends to over-fit the data faster than the MLP does and it stops after 6 epochs, since the validation loss already starts to increase from 0.564 after the first epoch and continues to swing later until it stops at 0.599.

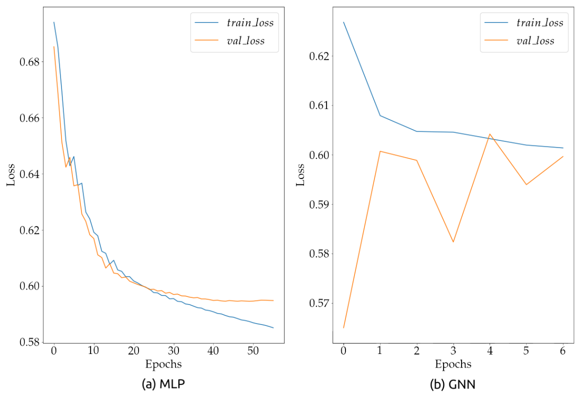


Fig. 5. Learning curves of (a) MLP and (b) GNN models on CWE-120

A similar behavior happens when training both models on the rest of vulnerability categories, which can be seen in Fig. 6, 7, 8, 9 and 10 in the Appendix. These figures show that the MLP takes more epochs to obtain the lowest possible $train-loss$ compared to the GNN model. The $val-loss$ starts to fluctuate in the case of GNN, whereas it is more stable by in the case of MLP. This indicates that GNN can be trained faster and generalizes better, while it tends to over-fit the training data sooner (i.e. within fewer epochs) than MLP.

TABLE I
THE DISTRIBUTION OF SAMPLES TO EACH VULNERABILITY TYPE IN DRAPER VDISC DATASET

CWE ID	Frequency %	Description
CWE-120	38.2%	Buffer copy without checking size of input ('Classic Buffer Overflow')
CWE-119	18.9%	Improper restriction of operations within the bounds of a memory buffer
CWE-469	2.0%	Use of pointer subtraction to determine size
CWE-476	9.5%	NULL pointer dereference
CWE-Other	31.4%	Improper input validation, use of uninitialized variable, buffer access with incorrect length value, etc.

TABLE II
NUMBER OF POSITIVE AND NEGATIVE SAMPLES BELONGING TO EACH CATEGORY AFTER UNDER-SAMPLING

Class	Vulnerable	Non-vulnerable
CWE-120	47660	47660
CWE-119	24157	24157
CWE-469	2625	2625
CWE-476	12094	12094
CWE-OTHERS	35028	35028
CWE-COMBINED	82411	82411

In order to evaluate how well the models can detect vulnerability, we use the following performance metrics: classification accuracy, precision, recall and F_1-score, all calculated using only the test samples of each CWE dataset. The results are shown in Table III.

TABLE III
SOURCE CODE VULNERABILITY CLASSIFICATION PERFORMANCE ACROSS EACH VULNERABILITY CLASS AND BASED ON DIFFERENT ML MODELS

Model	Accuracy	Precision	Recall	F_1
CWE-120				
SVM	0.665	0.667	0.677	0.672
MLP	0.681	0.682	0.686	0.684
GNN	0.671	0.637	0.779	0.701
CWE-119				
SVM	0.704	0.697	0.715	0.706
MLP	0.719	0.721	0.723	0.722
GNN	0.731	0.716	0.779	0.746
CWE-469				
SVM	0.693	0.732	0.631	0.678
MLP	0.660	0.634	0.776	0.698
GNN	0.746	0.752	0.761	0.756
CWE-476				
SVM	0.559	0.582	0.423	0.490
MLP	0.562	0.573	0.574	0.574
GNN	0.543	0.532	0.660	0.589
CWE-OTHERS				
SVM	0.621	0.627	0.601	0.614
MLP	0.640	0.663	0.598	0.629
GNN	0.625	0.620	0.657	0.638
CWE-COMBINED				
SVM	0.631	0.636	0.615	0.625
MLP	0.643	0.637	0.675	0.655
GNN	0.638	0.621	0.708	0.661

Looking at the obtained F_1-scores in Table III, it can be observed that SVM, MLP and GNN models all have obtained better F_1-scores on the detection of vulnerability tpyes of CWE-120, CWE-119 and CWE-469 than on the rest CWE types. The GNN model wins by achieving the best F_1-score on each CWE dataset.

Shown in Table IV is the comparison of our approach with [5]. Both use the Draper VDISC Dataset for model learning. The only difference lies in the number of samples available. Because they use only codes written in C language, whereas we use both C and C++ codes, our dataset is larger. Although the absolute number of samples used by us is bigger, the distribution of training, validation and test data is the same, i.e., 80%:20%:20%. Shown in Table IV is the performance comparison of vulnerability detections based on F_1-scores. An obvious increment of F_1-score for nearly all CWE datasets has been achieved by our approach, with the only exception on the CWE-476 dataset.

TABLE IV
PERFORMANCE COMPARISON BASED ON F_1-SCORES

	Related work [5]	Our work		
Model	CNN	SVM	MLP	GNN
CWE-120	0.427	0.672	0.684	0.701
CWE-119	0.509	0.706	0.722	0.746
CWE-469	0.090	0.678	0.698	0.756
CWE-476	0.598	0.490	0.574	0.589
CWE-OTHERS	0.270	0.614	0.629	0.638

V. CONCLUSION

In this work, we have used several ML algorithms to learn vulnerability patterns in C/C++ source code based on Draper VDISC dataset. First we generate ASTs which converts source code as a graph representation with detail structural information. Then we have proposed two approaches for the learning of graph embeddings, leading to vector-based representations of the input space. The first approach uses graph2vec as the embedding method, which takes a graph as input and transforms it into a lower dimension embedding vector. The embedding vectors can then be classified using either SVM or MLP for vulnerability detection. The second approach uses a GNN model to achieve a graph-level embedding as well as a binary classification. The second approach converges faster and performs better detection than the first one. Based on experimental results and comparative studies, it has been shown that the proposed approaches are able to effectively generate graph embeddings, detect vulnerability in source codes, and achieved improvement in detection accuracy over the state-of-the-art research work.

References

[1] X. Sun, Z. Pan, and E. Bertino, *Artificial Intelligence and Security*, 1st ed. Essex, England: Springer, Cham, 2019, eBook ISBN 978-3-030-24268-8.

[2] K. Filus, P. Boryszko, J. Domańska, M. Siavvas, and E. Gelenbe, “Efficient feature selection for static analysis vulnerability prediction,” *Sensors*, vol. 21, no. 4, 2021.

[3] M. Siavvas, E. Gelenbe, D. Kehagias, and D. Tzovaras, “Static analysis-based approaches for secure software development,” in *Security in Computer and Information Sciences*, E. Gelenbe, P. Campegiani, T. Czachórski, S. K. Katsikas, I. Komnios, L. Romano, and D. Tzovaras, Eds. Cham: Springer International Publishing, 2018, pp. 142–157.

[4] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, “Automated vulnerability detection in source code using deep representation learning,” 12 2018, pp. 757–762.

[5] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, “Vulnerability prediction from source code using machine learning,” *IEEE Access*, vol. 8, pp. 150 672–150 684, 2020.

[6] I. H. Sarker, “Machine learning: Algorithms, real-world applications and research directions,” *SN Computer Science*, vol. 2, no. 12, 03 2021. [Online]. Available: https://doi.org/10.1007/s42979-021-00592-x

[7] S. Halder and S. Ozdemir, *Hands-On Machine Learning for Cybersecurity*. Packt Publishing, 2018. [Online]. Available: https://books.google.de/books?id=2bj7uQEACAAJ

[8] S. M. Ghaffarian and H. R. Shahriari, “Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey,” *ACM Comput. Surv.*, vol. 50, no. 4, Aug. 2017. [Online]. Available: https://doi.org/10.1145/3092566

[9] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, “Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, p. 772–787, Nov. 2011. [Online]. Available: https://doi.org/10.1109/TSE.2010.81

[10] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, “Identifying the characteristics of vulnerable code changes: An empirical study,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 257–268. [Online]. Available: https://doi.org/10.1145/2635868.2635880

[11] Y. Shin and L. Williams, “An empirical model to predict security vulnerabilities using code complexity metrics,” in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’08. New York, NY, USA: Association for Computing Machinery, 2008, p. 315–317. [Online]. Available: https://doi.org/10.1145/1414004.1414065

[12] H. Dam, T. Tran, T. Pham, S. Ng, J. Grundy, and A. Ghose, “Automatic feature learning for predicting vulnerable software components,” *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 67–85, 2021, cited By 17. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85056740888&doi=10.1109%2fTSE.2018.2881961&partnerID=40&md5=14503f1ab61cc464c87cf150bafa5bb1

[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, “Vuldeepecker: A deep learning-based system for vulnerability detection,” *Proceedings 2018 Network and Distributed System Security Symposium*, 2018. [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23158

[14] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, “Learning to map source code to software vulnerability using code-as-a-graph,” 2020.

[15] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, “Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks,” vol. 32, 2019, cited By 36. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85089940996&partnerID=40&md5=360e4ac4b80f50a82fbee08b6e8712d6

[16] M. Grohe, “Word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data,” in *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–16.

[17] J. Brownlee. (2020, 01) Random oversampling and undersampling for imbalanced classification@ONLINE. Last accessed 18 September 2021. [Online]. Available: https://machinelearningmastery.com/random-oversampling-and-undersampling-for-imbalanced-classification/

[18] P. D. Thain, *Introduction to Compilers and Language Design*, 2nd ed., 2021, iSBN 979-8-655-18026-0.

[19] (2021, 04) Clang indexing library bindings. Last accessed 5 July 2021. [Online]. Available: https://libclang.readthedocs.io/en/latest/index.html?highlight=expression#clang.cindex.CursorKind.is_expression

[20] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, “graph2vec: Learning distributed representations of graphs,” *CoRR*, vol. abs/1707.05005, 2017. [Online]. Available: http://arxiv.org/abs/1707.05005

[21] B. Rozemberczki, “A parallel implementation of ”graph2vec: Learning distributed representations of graphs” (mlgworkshop 2017).” 2017. [Online]. Available: https://github.com/benedekrozemberczki/graph2vec

[22] A. Geron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 1st ed. O’Reilly Media, Inc., 2017.

[23] J. Gilmer, S. Schoenholz, P. Riley, O. Vinyals, and G. Dahl, “Neural message passing for quantum chemistry,” vol. 3, 2017, pp. 2053–2070, cited By 387. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85045254838&partnerID=40&md5=2752da1248c5ee0e4ce7c3d3d0b7c0f3

[24] T. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” 2017, cited By 2519. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85086180249&partnerID=40&md5=de472b43f496c76073ce3493990482e3

Appendix

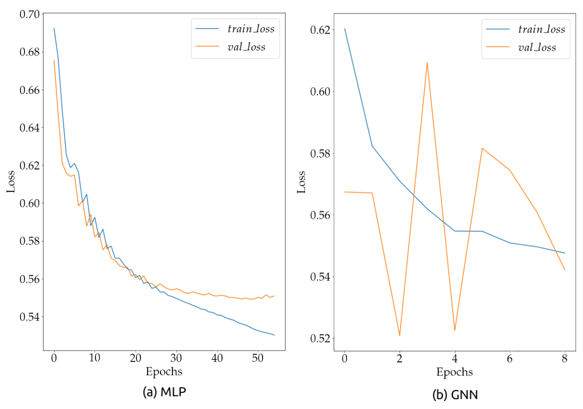


Fig. 6. Learning curves of (a) MLP and (b) GNN models on CWE-119

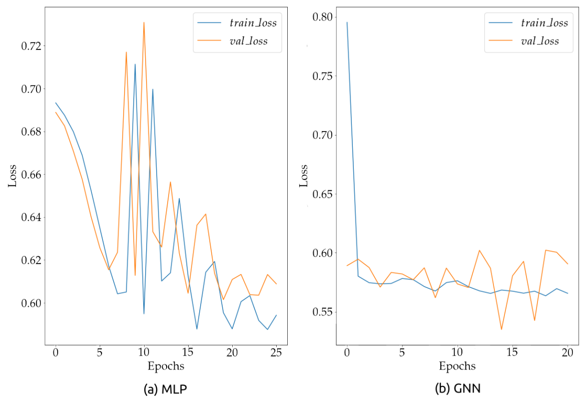


Fig. 7. Learning curves of (a) MLP and (b) GNN models on CWE-469

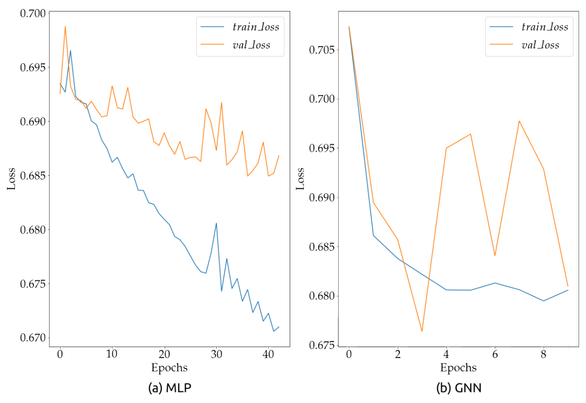


Fig. 8. Learning curves of (a) MLP and (b) GNN models on CWE-476

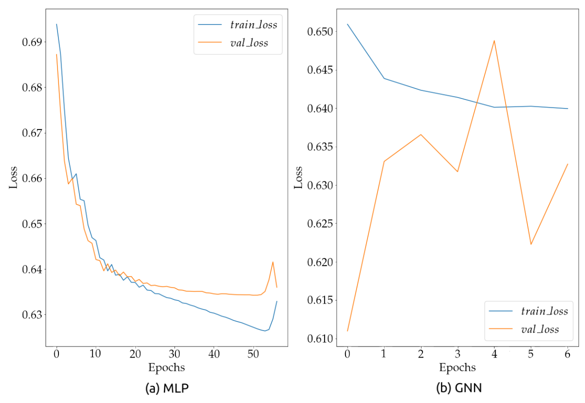


Fig. 10. Learning curves of (a) MLP and (b) GNN models on CWE-COMBINED

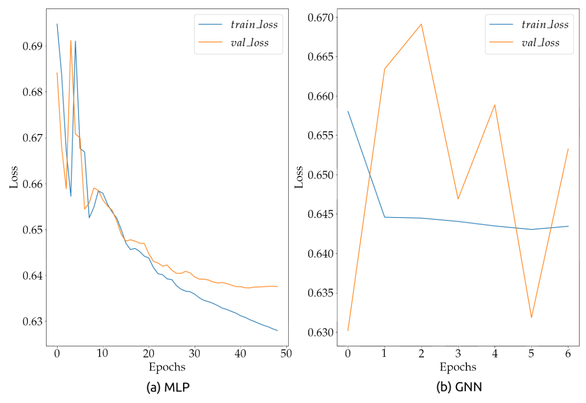


Fig. 9. Learning curves of (a) MLP and (b) GNN models on CWE-OTHERS

Design and Evaluation of a Quality Measurement Framework for News and Broadcasting Applications

Lukas Bluhmki
Matric Number: 11135342
Dept. of Comm. Sys. & Networks
University of Applied Sciences Köln
Cologne, Germany
Lukas.Bluhmki@smail.th-koeln.de

In Cooperation With:

Westdeutscher Rundfunk
Department of Program Delivery
Appellhofplatz 1, 50667
Cologne, Germany

***Abstract*— In order to cope with the growing demands of commissioning, maintaining, and monitoring large network infrastructures, modern broadcasting companies such as Westdeutscher Rundfunk need tools for rapid troubleshooting. Using the software-based measurement system vNet Fusion, a measurement framework could be designed that provides a tool for rapid troubleshooting. To control the software by means of an additional web framework, the NETCONF and YANG API was leveraged. The front and back end stack was designed using the Django framework. Thus, the basis for a powerful tool could be created, which will be available for all employees of Westdeutscher Rundfunk.**

Keywords*— *Network Measurements, vNet Fusion, LMAP, NETCONF, YANG, Django

I. INTRODUCTION

The increasing influence of digitization with the roll-out of Ethernet and IP infrastructures is also gaining the upper hand in the field of broadcasting. A modern broadcasting company is faced with the task of developing large-scale troubleshooting methods in addition to the in-depth knowledge required to operate these structures. Distributing audio & video (AV) streams over large networks, locations and infrastructures requires trained personnel and good preparation.

While modern companies are pioneers in moving to routed, packet-switched networks, WDR still operates many hybrid solutions of conventional transmission methods and IP transmissions. This is due to the fact that technical approaches for software-defined networking are only gradually finding their way into the many special solutions used by broadcasters. However, with developments in the area of "Everything as a Service" or cloud environments, the shift to virtual environments in broadcasting is also being driven forward.

This is where the software-based measurement system *vNet Fusion* [1] developed by Viavi Solutions comes in. It is based on the concept of distributing virtual machines and instances to build networks of measurement points. With the help of a controller and collector, these are orchestrated and managed. The architecture of this system is based on the Large-Scale Measurement of Broadband Performance (LMAP) framework [2]. This means that the measurement points act autonomously as individual instances and can be activated by instructions from the controller. In addition, the collected measurement data is communicated to the collector, which then processes it.

VIAVI's technical solutions are aimed at network engineers and technicians who want to solve complex problems with great expertise. In this work, however, the goal was to develop a tool that offers numerous employees from different areas a way to troubleshoot. This should support the daily work and simplify processes. Both the specifics and peculiarities of a broadcasting company were to be taken into account. In order to meet these requirements, an analysis of the environment had to be carried out before development could start. The basis for this project was the software *vNet Fusion*, the provided NETCONF & YANG API and the web framework *Django*.

II. FUNDAMENTALS

A. *VIAVI vNet Fusion*

VIAVI's vNet Fusion enables software-based lifecycle management on different OSI layers. This is done on the basis of the standards Y.1564 (EtherSAM) [3], RFC 6349 (TrueSpeed) [4] or RFC 5357 (TWAMP) [5]. Fusion seamlessly integrates into the rest of the environment of the VIAVI devices and can thus be seen as a platform for measurements of various kinds.

The virtual machines provided by Fusion act as measurement points and can communicate with each other as well as with hardware devices. This approach is clearly illustrated in Fig. 1. Here, the architecture is divided into a so-called Control Plane and a Data Plane. While the control plane contains all processes and instances for configuration, orchestration and management, the actual measurements take place on the data plane.

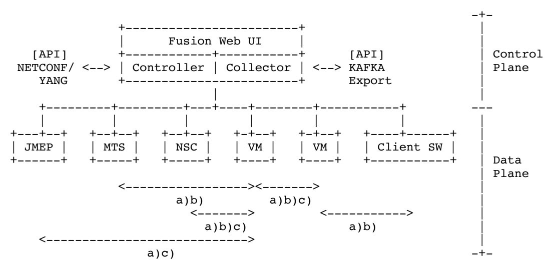


Fig. 1. The vNet Fusion Framework With Test Support Regarding to a) Y.1564 b) RFC 6349 c) RFC 5357 [1]

Fig. 1 also shows that the controller and collector can be accessed via the NETCONF & YANG and the KAFKA API. The *Fusion* internal Web UI enables all processes to be controlled and managed. The virtual machines (VMs) can be found in the data plane. In addition, the system provides a client software that can be used as a measuring point on one of the common operating systems. The left side of the data plane contains the hardware-supported devices: Beside the intelligent SFP, which was titled as JMEP, in addition the devices MTS and NSC. The relationships listed at the bottom of the figure show the interoperability of the instances with respect to their measurement methods.

B. *NETCONF & YANG*

The Network Configuration (NETCONF) protocol [6][7] is designed to meet complex requirements for the configuration, management, and operation of network devices. The communication protocol is based on a layer principle. At the top level, information is packed into a secure transport layer initiated by SSH or similar protocols. The remaining layers (messages, operations, and content) are filled with information according to an XML structure. The content can be transported in the form of XML or JSON.

The communication is initiated between the client (initiator) and the server (responder). The NETCONF server is implemented on the device to be controlled. The data exchange takes place via Remote Procedure Calls (RPCs). In addition to the status display, these also enable the transmission of parameters to be configured or tasks to be performed. The server responds to an RPC request either with a reply, an error or a simple OK.

For the communication with the *vNet Fusion* System, VIAVI has defined a special RPC operations. These include *Login, renewToken, startMATest, getReports or Logout.* [8]

In order to have a standardized structure specification for the control and configuration data to be transmitted, the modeling language YANG [9][10] is used. In YANG, data is organized in a hierarchical tree structure whose individual components are referred to as nodes. Each component can contain further nodes. The YANG models are defined by the manufacturer of the devices API, so that they can be controlled according to prescribed standards.

C. *Django Web Framework*

The *Django* Web Framework is python-based and was released as free and open-source software. It provides a framework that simplifies the development of web applications. It is possible to start the application via the integrated web server. To make the process of development efficient, the software provides classes, functions, and various tools [12].

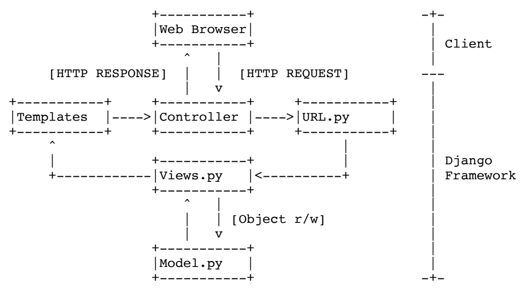


Fig. 2. Django Framework Model-View-Controller Architecture [11] [modified]

To understand how *Django* works, the Model-View-Controller (MVC) architecture can be considered (Fig. 2) [12]. The client calls the web application by sending an HTTP request. The called URL is evaluated by the *URL.py* script and forwarded to the backend logic in *Views.py*. Communication to dynamic content in databases can also take place there. This communication is done via an object-relational mapping (*Model.py*). Once processing is complete, the information is forwarded to the templates. These contain information about the front end (HTML, JavaScript, etc.) and return the collected, produced data to the client via the HTTP Response.

III. Requirements Engineering

A suitable requirements engineering is indispensable for the success of a modern software project. In this context, it is useful to apply a model for the orderly execution of these processes. In this case, the Twin Peaks model [13] was chosen, which is characterized by the fact that both requirements and architecture are in an agile relationship to each other. Continuous immersion in the subject matter results in a more pronounced picture of both the requirements and the architecture of the system. This increase in the level of detail leads to the completion of the system.

In order to clearly define the requirements to be considered, they were divided into four topics: Test Procedure, Results Handling, User Interface Design and Workflow Integration. The identification of these requirements was gathered through a variety of field reports and analysis of problem scenarios.

A. *Test Procedures*

The examination of the test requirements revealed that common troubleshooting scenarios can be located in the area of service provisioning as well as monitoring. In addition, the tests should take into account the analysis of a committed information rate (CIR) as well as the measurement parameters packet loss, latency and jitter. In addition to the comparison of common protocols of the Ethernet, IP, TCP and UDP stack, it would also be desirable to be able to analyze newer developments such as UDT [14] and QUIC [15]. Since the development should be clearly optimized for use in a broadcast environment, an implementation of measurements of AV protocols like SIP, RTP, RTMP, SRT, or RIST would be appreciated.

B. *Results Handling*

Another important point is the processing of the determined results in the form of data sets. These should be stored in a suitable format and permanently available. In addition, this storage should allow both the display of detailed information in a user interface and be exportable for other purposes. This could be ensured with an API. The data sets should also be available in a multi-user environment and be able to be provided with meta data.

C. *User Interface Design*

The success of the tool to be developed will be largely determined by how intuitive and easy to use the user interface is. For the wide range of different users, it would therefore be conceivable to provide different modes (Beginner and Expert) that vary in the complexity of the available functions. As shown in Fig. 3, the user interface should provide visual feedback on the measurement that is easy to understand.

```
  WDR Netztest             [Run Test] [Results] [Templates]
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

      ^ Values                         +--Feedback--+
      |         _/\                    |  _         |
      |   /\   /   \  /\               | (_) green  |
      |  /  \_/     \/  \        ==>   | (_) yellow |
      | /                \             | (_) red    |
      |                                |            |
      +-------------------> Time       +------------+
```

Fig. 3. Schematic Sketch of the User Interface

The data should be visualized in a diagram and an easy-to-understand signal should provide information about the suitability of the measured service. As can be seen in the upper right of Fig. 3, the user interface could use various links in a navigation to forward to the results and the creation of predefined templates. It would also be desirable if the user interface could be found in the corporate design of Westdeutscher Rundfunk (WDR), in order to enable seamless integration into the existing environment.

D. *Workflow Integration*

WDR has a technical infrastructure that places certain requirements on the new tool to be developed. Since the internal network extends over many locations, providers, and even national borders, it is necessary to develop an easily scalable and distributable system. In the best case, the user should get along without the use of additional technology or devices. This ensures mass compatibility. In addition, the measurement tool should be universally and decentralized applicable. Since the average employee owns both a company cell phone and a laptop, it would make sense to use these devices to run tests.

IV. Design

A. *Overview*

To get an overview of how the different components were put together to form a new measurement framework, Fig. 4 provides information. The Quality Measurement Framework results from from the LMAP Framework, which is represented by *vNet Fusion*, and the *Django* Web framework. The Quality Measurement Framework (QMF) Orchestrator can be seen as an instance that combines the different services into a composition. These services include e.g. views, templates and models of the Django framework, the RPC & Netconf library and the web server functionality.

The control protocols that are responsible for the exchange of configuration data can also be identified. Looking at the framework from an SDN perspective, the communication via NETCONF between Orchestrator and vNet Fusion can be seen as a northbound API. The internal data exchange between Controller & Collector and test points via RESTCONF, on the other hand, represents the southbound API. The transported configurations according to the YANG modelling models are identical. Communication between the access device and the orchestrator takes place by default using HTTP. As described in Fig. 3, the test point can be a VM, client software, as well as a hardware device such as JMEP, MTS, or NSC.

```
                                                          -+-
+-----------------------------+        +--------------+    |
|      QMF Orchestrator       +--------+    Access    |    | Django
| (Django, RPC & Netconf Lib) |  HTTP  |    Device    |    | Web
+-----------------------+-----+        +--------------+    | Framework
               NETCONF  |                                  |
       (Northbound API) |                                 ---
+-----------------------+-----------------------------+    |
|                Controller & Collector               |    |
+-----------------------+--+--+-----------------------+    |
              RESTCONF  |  |  |                            | LMAP
       (Southbound API) |  |  |                            | Framework
     +------------------+  |  +-------//---------+         | (vNet
     |                     |                     |         | Fusion)
+----+----+           +----+----+           +----+----+    |
| Test    |           | Test    |     ..    | Test    |    |
| Point 1 |           | Point 2 |           | Point n |    |
+---------+           +---------+           +---------+    |
                                                          -+-
```

Fig. 4. The Composition of the Quality Measurement Framework From Fusion and Django Including Used APIs and Protocols

The web application to be designed will henceforth be given the working title "Netztest", engl. net(work) test. With *vNet Fusion* pre-integrated into the environment, the focus was on developing the back and front end using *Django*. The decision to use the *Django* web framework came about for a variety of reasons. The server-side framework offered the greatest compatibility with the requirements of the tool.

B. Back End Stack

The central link between the frameworks is the library *Netconf-Client [16]*. It was selected after evaluating many different libraries from different programming languages. Since *vNet Fusion* uses its own NETCONF operations, these also had to be configurable via the library. The most intuitive way to implement this was via the *Netconf-Client* library. An SSH connection is established via the *connect_ssh* function. This can transfer customizable RPCs using the *manager()* class.

To be able to call the RPC operations provided by Fusion such as *Login*, *renewToken*, *startMATest*, *getReports* or *Logout* in the backend [8], a library was designed. The architecture of this library can be seen in Fig. 5. The upper half of the image describes the built of the RPC frame, while the lower half contains operations, content, and results processing.

```
  RPC             +-----------------------------+  +-------------+
  Framework       |        rpcRequest()         |  |  Test XML   |
  Build           +-----------------------------+  |    File     |
. . . . . . . . . . .|. .|. .|. .|. .|. .|. .|. . .|. . . . . . .|. .
                     |   |   |   |   |   |   |     |             |
                  +--+---+---+---+---+---+---+--+  |             |
  Operations,     | rpcLogin(), rpcLogout(),    |  +-------------+
  Content         | getTokenDetails(),          |   |   |   |   |
  & Results       | stopTest(), getMAList(),    |  +-------------+
  Processing      | getLatestReport(),          |  | startTest() |
                  | getFullReport()             |  |             |
                  +-----------------------------+  +-------------+
```

Fig. 5. Architecture of the Designed RPC Library

Most functions use the function *rpcRequest()* to build the RPC. These include *rpcLogin()*, *rpcLogout()*, *getTokenDetails()*, *stopTest()*, *getMAList()*, *getLatestReport()*, and *getFullReport()*. An exception applies to the startTest() operation. Since this has a certain complexity due to the transfer of numerous configuration parameters and varies greatly depending on the test type (Y.1564 or RFC 6349), it was decided not to build the RPC using *rpcRequest()*. Instead, XML files are available for each test type, which represent the basic structure of the call. The configuration parameters are written into these files and sent as a whole. Listing 1 shows how a valid RPC is assembled from the operation and then submitted via the manager (*mgr*) and the *dispatch()* function.

```
def rpcRequest(operation):

    rpc = """
    <action xmlns="http://tail-f.com/ns/netconf/actions/1.0">
        <data>
            <""" + operation[0] + """
xmlns="http://viavisolutions.com/ns/yang/vtest">"""

    for i in operation[1]:
        rpc = rpc + "\n<" + i[0] + ">" + i[1] + "</" + i[0] + ">"

    rpc = rpc + "\n</" + operation[0] + """>
        </data>
    </action>"""

    return mgr.dispatch(rpc)
```

Listing 1. RPC Request Function Design Code Example from rpc.py

The central framework logic can be found in the *Django* Views. This is where the processes for the individual pages are processed and the content is generated. The views start RPCs, accept HTTP POST & GET values, read & write the dynamic data of the databases via models and pass the results to the templates. The developed framework has seven different views and corresponding templates.

- The *Login & Logout View* uses the authentication tools of the *Django* framework. It compares the user data sent by the template via HTTP POST with database entries and manages sessions to log the user in or out.

- The *Home View* has no task except to output the associated template and provide static information. No processing of dynamic data takes place.
- With the *Templates View* the user has the possibility to create and edit templates for recurring troubleshooting scenarios. This is the basis to start network tests quickly and easily without having to make recurring configurations. Besides the CIR, there is the possibility to specify a duration and to name the test. The configuration options were deliberately kept simple.
- The network measurements are started and performed using the *TestAdd, Test, and TestRefresh Views*. The functionality includes the selection of the network connection to be measured, as well as the selection of a template. Once the test is started, the data is visualized live and refreshed using a jQuery command.
- The Result View allows the user to view or evaluate measurements that have already been performed. In addition to the overview, a detailed view with extended information is also provided.

All views have corresponding templates which contain the frontend for user interaction. These have been designed and developed to be easy to use without the need for in-depth network engineering knowledge.

The final link in the Django architecture (Fig. 2) is formed by the so-called models, which represent tables in a database. Two tables have been created.

The *Templates* table contains the configuration data generated by the user to start tests. For this purpose, the fields *Title, Type, CIR,* and *Duration* have been created. The more complex *Results* table contains several fields. Besides date, template parameters, and selected measurement agents, fields for the measurement data were created. These include the bidirectional values for the data rate and the Round Trip Time (RTT). Furthermore, the model contains functions that calculate the mean of the measured values and determine a KPI for the visual feedback of the suitability of the connection. The functions are also mapped to the objects and can be used like field entries.

To understand the interaction between Model, Template, View, RPCs and Static Content, the flowchart in Fig. 6 can be consulted.

C. *Front End Stack*

A useful tool needs a well-designed user interface. Even if the focus of this work is based on the development of the backend functionality, care was taken to meet the requirements of the user interface.

Django Templates offer the possibility to link the HTML, CSS, JavaScript and dynamic content using the *Django Template Language* (Python) [17]. The front end was created with the help of the CSS framework *Bootstrap* [18]. The framework offers a responsive design by default, which meets the requirements of device compatibility. In addition, care was taken to take into account the CI of WDR. In the test and results templates, the Javascript library Chart.js [19] was applied to visualize the results. Also Javascripts that were already integrated in the Bootstrap framework have been used to simplify the interaction.

D. *Full Stack*

Fig. 6 illustrates the interaction of the frameworks components. If the flowchart is viewed from bottom to top, the models, the various RPC functions, the views, the templates and the static content can be identified.

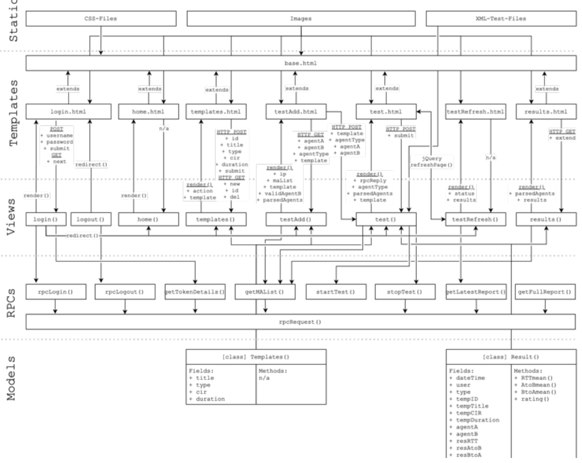


Fig. 6. Model, View, Template, RPC and Static Content Interaction of the Quality Measurement Framework

It can be seen that the two models are used in the views "templates()", "test()" and "results()". The implemented functions of the models are also listed in the right-hand area of the fields. In addition, it can be seen in the RPC Library that, with the exception of "startTest()", all functions refer to the basic function "rpcRequest()". Additionally, the information exchanged between the templates and views via HTTP POST, GET and the function parameters is illustrated. It is also made clear that the templates are all based on the "base.html" template, which provides the framework for the frontend. The CSS files, images and test configuration XML files can be considered static content.

V. Evaluation

In order to evaluate the suitability and consideration of previously established requirements, a test environment was built. This was mainly used to perform series of measurements to verify the functionality of the test procedures. Other requirements like the result handling, the user interface design or the workflow integration could be evaluated in a different way.

Fig. 7 shows that the measurement was performed between two buildings of the headquarters in Cologne. The first virtual test agent (vTA) was located on the host server, which also hosted both the *vNet Fusion* Controller and the Netztest VM. The second VM was hosted on a mini PC in another building. Both management and test connections passed through the WDR network and were not logically separated. Since the focus was on triggering the functions implemented in the backend and not on the measurement itself, there was nothing preventing this approach.

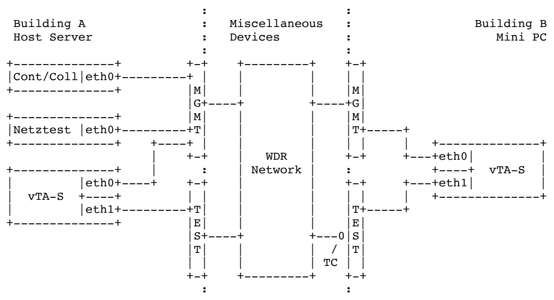


Fig. 7. Test Environment for the Prototype Evaluation

The central link of the Measurement Framework is the visual feedback within the measurement process. This is done via diagram and in a simplified form, after completion of the test as a traffic light. The measurement process checks the connection for the previously set CIR and provides the user with feedback as to whether this has been achieved or not. To test this function a traffic control (Fig. 7: TC) was implemented at the physical ingress interface of the Mini PC using the Linux package Iproute2 [20]. Three test runs took place:

1. No Traffic Shaping
   Max Bandwidth: As provided
2. Light Traffic Shaping
   Max Bandwidth: 5 Mbps
3. Heavy Traffic Shaping
   Max Bandwidth: 100 kbps

Listing 2 shows how the traffic control (tc) was applied. For this purpose, a classless queuing discipline (qdisc root) was assigned to the virtual bridge "Test" (virbrTE) as a substitute for the physical interface eth1 of the Mini PC. The token bucket filter (tbf) was applied with a maximum data rate of 5 Mbps and a burst size of 32 kbit. In addition, all packets remaining in the queue for more than 400 ms are discarded.

```
tc add dev virbrTE \
   root qdisc \
   tbf \
   rate 5mbit \
   burst 32kbit \
   latency 400ms
```

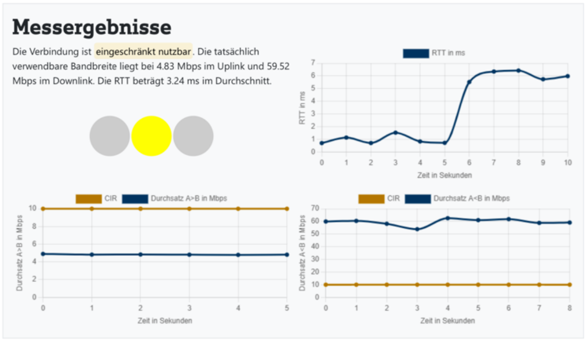
Listing 2. Light Traffic Control for virbrTE with 5 Mbps Traffic Shaping

The various test runs were performed using the TrueSpeed test according to RFC6349. A template was selected which specifies 10 Mbps as the CIR. The results showed that the visual feedback was displayed as desired. As long as the traffic shaping was switched off, the traffic light was green. This meant that the actual throughput was above 10 Mbps. When light traffic shaping was enabled, the traffic light turned yellow. This meant that the connection was functional, but offered less than the agreed 10 Mbps. With heavy traffic shaping, the traffic light was red. Since the available throughput was 100 kbps here, this was considered an unusable connection. Once the throughput is below 5% of the CIR (10 Mbps * 0.05 = 500 kbps), this is indicated.

Fig. 8 gives an overview of the visualization of the measurement results. The case of light traffic shaping can be seen, where the traffic light turns yellow. Also shown are the curves for the RTT, and the throughputs in both directions. Thus, this figure also confirms that many requirements have been taken into account. Both the visualization of the determined measurement data and the simple comprehensibility of this are propagated.

A closer look reveals irregularities in the duration of the tests. The test is configured to last 15 seconds. It can be seen that the throughput of

A>B is measured for 5 seconds and the throughput of A<B for 8 seconds. This error is due to the northbound API via NETCONF and does not correspond to the actual measurement result. Inspection of the measured data in the Fusion UI indicates that all data has been successfully collected via the RESTCONF API. However, the transport via NETCONF could not be implemented successfully until the end of the project, despite assistance from the manufacturer.



Messergebnisse

Die Verbindung ist eingeschränkt nutzbar. Die tatsächlich verwendbare Bandbreite liegt bei 4.83 Mbps im Uplink und 59.52 Mbps im Downlink. Die RTT beträgt 3.24 ms im Durchschnitt.

Fig. 8. Measurement Results With Traffic Light Visualization and Diagrams

With regard to the test procedures specified in the requirements, a mixed summary can be drawn. The available measurement procedures are limited to the functional scope provided by *vNet Fusion*. In addition to Y.1564 (EtherSAM) and RFC6349 (TrueSpeed), this also includes RFC 5357 (TWAMP). These measurements support troubleshooting on layer 2-3 and the usual protocols of the Ethernet, IP, UDP, and TCP stack. Unfortunately, no application protocols for media applications are supported (RTP, SRT, RIST) or the API required for this is not provided (SIP). Modern developments such as UDT and QUIC are also not taken into account in *vNet Fusion* at the current time. Therefore, it was not possible to integrate measurement methods in the network test application for these protocols.

Measurements using TrueSpeed are reliably implemented in the Measurement Framework, whereas EtherSAM still requires further bug fixes. The consideration of possible network parameters such as packet loss, delay and jitter varies depending on the test method. At the current state, the assessment of the connection is based only on the parameter of the CIR, which can be further expanded in the future.

The determined measurement data are stored in the form of lists in the SQLite database table. This enables further processing. In addition, it is conceivable to implement an API that can export this data.

With regard to the requirements for the user interface design, almost all requirements could be met. The user can easily navigate through the page and finds explanations at numerous points to be introduced to the topic. For the full validation of usability, extensive usability testing with numerous participants could still be done in the future.

In addition, emphasis was placed on seamlessly integrating the framework into the WDR environment. To this end, the decentralized nature of availability was taken into account. This means that the web application can be accessed from any device with a web browser as long as it is in the WDR network.

In addition, both *vNet Fusion* and the web application developed with *Django* are hosted on virtual machines, which allows numerous possibilities for system integration. A scalable and easily deployable system has been developed. Currently, however, it operates only in the DMZ and cannot yet be accessed outside via a proxy. Especially for connections that find access to the WDR via an external DSL, this feature would be recommendable.

VI. Conclusion & Outlook

The idea of designing a Quality Measurement Framework had the goal in mind to simplify the work of a broadcast engineer and to expand areas of competence with the help of a tool. The aim was to combine the two worlds of broadcasting and the ever-increasing spread of IP environments. The developed framework and the resulting tool fulfill almost all of the previously identified requirements. At the same time, however, it forms the basis for future developments and enables expansion into many different areas.

With the integration of tests according to Y.1564 and RFC 6349, powerful troubleshooting approaches are available to the user. Nevertheless, it would be desirable if in the future these techniques could be adapted to broadcast-specific use cases. This could be done by implementing AV transport protocols or codecs. However, since these are also transported on the common layers of the Ethernet, IP, UDP, TCP stack, the measurement methods already implemented are both informative and powerful.

The manufacturer VIAVI also gave an outlook on further developments in the direction of containerization, the full implementation of RESTCONF and the simplification of existing structures. Furthermore, the functionality is extended by 10 Gbps and even 100 Gbps support. These transmission rates are especially interesting for the WAN area. Future developments will thus also have a significant influence on Netztest's own development and offer new possibilities. This could also happen with respect to new protocols and test methods.

During the development, the progress was partly hampered by the incomplete documentation, as well as repetitive errors and bugs in the *vNet Fusion* software and API. It remains to be hoped that these vulnerabilities will be optimized in the future so that communication via the API can be more reliable.

In summary the quality measurement framework forms the basis for the future optimization of troubleshooting processes. WDR now has a tool that takes the company to a new level of maintenance and support of large network structures. Future success will depend not only on the technical development but also on the dissemination among colleagues and the creation of awareness for this tool.

References

[1] Viavi Solutions Inc. Virtual Service Activation and Performance Management. 2018. url: https://www.viavisolutions.com/ru-ru/literature/nitro-vnet-fusion-product-solution-briefs- en.pdf (visited on 11/05/2020)

[2] P. Eardley et al. A Framework for Large-Scale Measurement of Broadband Performance (LMAP). RFC 7594. Sept. 2015. doi: 10.17487/RFC7594

[3] ITU-T. “Ethernet service activation test methodology”. In: ITU-T Y.1546 (Feb. 2016). url: https://www.itu.int/rec/T-REC-Y.1564/en (visited on 07/13/2021)

[4] R. Schrage et al. “Framework for TCP Throughput Testing”. In: Request for Comments 6349 (Aug. 2011). doi: 10.17487/RFC6349

[5] J. Babiarz et al. A Two-Way Active Measurement Protocol (TWAMP). RFC 5357. Oct. 2008. doi: 10.17487/RFC5357

[6] R. Enns. NETCONF Configuration Protocol. RFC 4741. Dec. 2006. doi: 10.17487/RFC4741. url: https://rfc-editor.org/rfc/rfc 4741.txt

[7] R. Enns et al. Network Configuration Protocol (NETCONF). RFC 6241. June 2011. doi: 10.17487/RFC6241. url: https://rfc-editor.org/rfc/rfc6241.txt.

[8] Viavi Solutions. Controller and Collector APIS - Version 6.0. 1-844-468- 4284. Nov. 2019.

[9] M. Bjoerklund. YANG - A Data Modeling Language for the Network Con- figuration Protocol (NETCONF). RFC 6020. Oct. 2010. doi: 10.17487/ RFC6020. url: https://rfc-editor.org/rfc/rfc6020.txt.

[10] M. Bjoerklund. The YANG 1.1 Data Modeling Language. RFC 7950. Aug. 2016. doi: 10.17487/RFC7950. url: https://rfc-editor.org/rfc/rfc7950.txt.

[11] D. Savić et al. “Simulation Data Exchange - Web Interface for CostGlue Application”. In: (Mar. 2008), p. 8. url: https://www.researchga te . net / publication / 297264757 _ Simulation _data_exchan ge_-_web_interface_for_CostGlue_application (visited on 07/14/2021).

[12] N. George. Django’s Structure—A Heretic’s Eye View. July 2021. url: h ttps://djangobook.com/mdj2-django-structure/ (visited on 07/15/2021).

[13] B. Nuseibeh. “Weaving together requirements and architectures”. In: Com- puter 34.3 (2001), pp. 115–119. doi: 10.1109/2.910904.

[14] Y. Gu. UDT: UDP-based Data Transfer Protocol. Internet-Draft draft-gg- udt-03. Work in Progress. Internet Engineering Task Force, Apr. 2010. 19 pp. url: https://datatracker.ietf.org/doc/html/draft-gg-udt-03 (visited on 06/12/2021).

[15] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. May 2021. doi: 10.17487/RFC9000.

[16] I. ADTRAN. netconf-client - A NETCONF client for Python 2.7 and 3+. Jan. 2019. url: https://github.com/ADTRAN/netconf_client (visited on 06/02/2021).

[17] D. S. Foundation. Django Documentation - Release 3.2.6.dev. July 2021. url: https://buildmedia.readthedocs.org/media/pdf/django/3.2.x/django.pdf (visited on 07/15/2021)

[18] Mark Otto et al. Bootstrap – Release 5.1, July 2021 . url: https://getbootstrap.com/ (visited on 07/02/2021)

[19] E. Timberg. Chart.js - Simple HTML5 Charts using the <canvas> Tag. Mar. 2013. url: https://github.com/chartjs/Chart.js (visited on 07/27/2021).

[20] L. Foundation. Iproute2 - A collection of Utilities for Controlling TCP / IP Networking and Traffic Control in Linux. Apr. 1999. url: https://wiki.linuxfoundation.org/networking/iproute2 (visited on 07/29/2021).