IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 30, 2021, accepted April 16, 2021, date of publication April 23, 2021, date of current version June 28, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3075385

Context-Aware Software Vulnerability Classification Using Machine Learning

GRZEGORZ SIEWRUK[1,2] AND WOJCIECH MAZURCZYK[1,3], (Senior Member, IEEE)

[1]Faculty of Electronics and Information Technology, Warsaw University of Technology, 00-661 Warsaw, Poland
[2]Orange Poland S.A., 02-326 Warsaw, Poland
[3]Chair of Parallelism and VLSI, FernUniversität in Hagen, 58097 Hagen, Germany

Corresponding author: Grzegorz Siewruk (g.siewruk@ii.pw.edu.pl)

This work was supported by the SIMARGL project from the European Union's Horizon 2020 research and innovation programme under Grant 833042.

ABSTRACT Managing the vulnerabilities reported by a number of security scanning software is a tedious and time-consuming task, especially in large-scale, modern communication networks. Particular software vulnerabilities can have a range of impacts on an IT system depending on the context in which they were detected. Moreover, scanning software can report thousands of issues, which makes performing operations, such as analysis and prioritization, very costly from an organizational point of view. In this paper, we propose a context-aware software vulnerability classification system, Mixeway, that relies on machine learning to automatize the whole process. By training a model using known and analyzed vulnerabilities together with Natural Language Processing techniques to properly manage the information that the vulnerability description contains, we show that it is possible to predict the class that defines how severe the detected vulnerability is. The experimental results obtained on a real-life dataset collected by Mixeway for about 12 months from the infrastructure of one of the major mobile network operators in Poland prove that the proposed solution is useful and effective.

INDEX TERMS IT security, devsecops, machine learning, classification, vulnerability classification.

I. INTRODUCTION

The introduction of DevOps methodology to the software development industry has led to various changes [1]. Project management changes enables applications to be developed in a more flexible way. IT system architecture changed from a monolith to MicroService design [2] and the time to deploy a new version of an application into a production environment accessible for end-users is expected to be as short as possible. The last aspect is caused by the rapid growth of interest in software that automates operations related to software development. Such automation is a set of operations triggered by particular events (like pushing source code to the code repository) defined in a structured format called a *pipeline* [3]. When simplifying the whole process, steps like 'application building' and 'application deployment' are executed without human interaction and triggered by a specific event, and they are preceded by a specific set of tests that confirms that the new version of the developed software functions as expected.

A credible pipeline should include, in addition to mechanisms for verifying the correct behavior of the application, security verification mechanisms, which are one of the factors determining the software quality. The mentioned mechanisms should consist of an operation that verifies if a new version of an application does not contain security vulnerabilities (a deficit that may lead to unauthorized access or other threats that can compromise the system) [4]. In a model where dozens or hundreds of deployments are introduced daily, it is no longer possible to perform a manual review created by an IT security professional. Such verification should be replaced by tools allowing for the automatic verification of application security – vulnerability scanners. Unfortunately, the information obtained from such data sources may cause potential problems. One of which could be false positives (FP) that are reported in a given context but for which they may not be a real threat to the application. According to research published in [5] the ratio of all detected vulnerabilities to FPs is even 50% for some particular vulnerability scanners. This means

The associate editor coordinating the review of this manuscript and approving it for publication was Jiafeng Xie.

that one of two reported deficits should never be announced to development teams. Thus, it is hard to rely on such results when trying to design an automated process that will include a security checkup.

Currently, software vulnerability classification is an interesting research topic that has been raised by several researchers who have proved that machine learning algorithms can be useful for this purpose. The work of [6] demonstrates that the Neural Network algorithm gives the best results (compared to Naive Bayes, Support Vector Machine, and K-Nearest Neighbors) in the category prediction of reported vulnerabilities (Cross-Site Scripting, SQL Injection, etc.) based on a description of the deficit in the software. A large number of researchers focus on techniques that allow the detection of vulnerabilities, which leads to the classification of the source code as a vulnerability. The papers [7] and [8] introduce the concept of the *Vulnerability Extrapolation*, which is a multistep process that aims to *identify unknown vulnerabilities based on programming patterns observed in the familiar security vulnerabilities* [9]. The research presented in [10] and [11] describes the concept of *PhpMinerI*, which allows to predict if a specific statement in the source code of applications created in PHP technology is vulnerable either to Cross-Site Scripting or SQL Injection attacks. Unfortunately, none of these proposals could be used in large-scale real-life software development and security assurance processes. Current IT system architectures support a number of various technologies, so restricting the classifier in scope of the source code language (only for PHP or C/C++ applications) is the factor that makes it impossible to use such a solution. The predicted outcome is also important. Thus, rather than the defect category, which is often provided, it is information about the severity and impact of a software vulnerability what is desired instead.

That is why, in this paper we would like to address these deficits by proposing a *context-aware software vulnerability classification system*, called Mixeway, that uses machine learning techniques to make the process automatic. Based on a real-life dataset of software and network vulnerabilities, we train a model and we utilize Natural Language Processing techniques on the vulnerabilities' descriptions. Using such an approach we are able to successfully predict the severity of the vulnerability and assign it a proper label, called later a *grade*, so it can be efficiently mitigated. To the authors' best knowledge, such an approach has not been proposed nor evaluated previously in the literature.

In this paper, the term context-aware software vulnerability classification is defined as a classification that allows to distinguish if a given software vulnerability (reported by the automated scanning software) is relevant, thus should be fixed, or irrelevant when taking the context of an asset (affected by the reported deficit) into account.

Considering the above, the main contributions of this paper include:

- proposing a novel context-aware software vulnerability classification system, called Mixeway, which relies on machine learning and NLP techniques and allows to automatically predict the vulnerability category (i.e., whether it is necessary to be fixed or it is not relevant in the given context) and assign it a proper label.
- evaluating the effectiveness of the proposed system on a real-life dataset consisting of more than 50,000 software and network vulnerabilities gathered from the infrastructure of one of the major mobile operators in Poland.
- utilizing Natural Language Processing techniques on vulnerabilities' descriptions which have not been previously used in this context in the literature.
- making Mixeway publicly available as an open-source software at GitHub (https://github.com/mixeway).

The proposed solution is used to analyze and classify vulnerabilities detected by various sources to help security teams sift out the ''noise'' (irrelevant findings or false positives) contained in reports from automated testing tools (evaluation of the performance of the individual scanners is out of scope in this research).

The rest of the article is structured as follows. Section II contains a review of the related research, while Section III describes how the software delivery automation is currently being adopted by programming teams and how the security scanning software to support Continuous Integration and Continuous Deployment (CICD) process can be implemented. Next, in Section IV the basics of the selected algorithms used during the experimental evaluation are provided. Then, in Section VI the experimental test-bed, the methodology, and dataset are outlined, while Section VII presents the obtained results. Finally, Section VIII summarizes our work and indicates potential future research directions.

II. RELATED WORK

Machine learning algorithms are widely used in cyber security research. Currently, several works exist in the literature that demonstrate how machine learning algorithms can be utilized to analyze software vulnerabilities. They can be roughly grouped into two categories: detection of vulnerabilities in software and software vulnerability classification. The most relevant works from both groups are briefly reviewed below. There is a third group worth mentioning, that is network traffic classification. It corresponds to the solutions that use ML algorithms to detect cyber-attacks based on the network traffic incoming to a particular asset [12]. There is also a wide range of research regarding self-protection methods in mobile computing [13]. Those groups, however, will not be further described, as presented research is focused on software vulnerabilities rather than network vulnerabilities.

A. SOFTWARE VULNERABILITIES DETECTION USING MACHINE LEARNING

The first group is related to the detection of vulnerabilities in software. The solutions described in [14] and [15] investigate multiple types of neural networks to analyze the source code written in C/C++. The results enclosed in both

papers confirm that Convolutional Neural Networks (CNN) are promising solutions for this purpose. Next, in paper [16] the authors analyze the use of the Support Vector Machine algorithm with the Bag of Words representation of the source code. The obtained results prove the high accuracy of the developed model, although the training set was limited to just one Java application. Finally, the research in [17] analyzed multiple algorithms and showed that the CNN algorithm is the most suitable for vulnerability detection in the source code written in C/C++.

B. SOFTWARE VULNERABILITIES CLASSIFICATION USING MACHINE LEARNING

The second category is software vulnerability classification. The solutions within this group are most closely related to the topic of this paper. The research described in [18] evaluated multiple classifying algorithms, such as Random Forest, C4.5 Decision Tree, Naïve Bayes, and Logistic Regression for this purpose. The authors proposed a classifier that can detect if the vulnerability (from bug trackers like Bugzilla[1]) is a type of BV (BohrVulnerability, i.e., easy to find and easy to fix) or MV (MandelVulnerability, i.e., hard to find and complex to fix). Based on the obtained experimental results, it turned out that Random Forest and Decision Tree gave the best results. Next, in [19] the National Vulnerability Database (NVD), which contains information about publicly disclosed vulnerabilities, was utilized. This solution proposed a classifier using Support Vector Machines (SVM) to build a model that would predict the category of vulnerability, e.g., Cross-Site Scripting (CSS). Another work [20] in which NVD and CVE databases are used, proposes a classifier that can predict vulnerability severity based on its description. According to the presented results, the Neural Network algorithm obtained slightly better results than SVM. Research presented in [21] investigates the possibility of creating a model able to predict a number of vulnerabilities to be disclosed in the future (operating system affected). Extensive research in the scope of prediction when the next vulnerability for the particular software (e.g., a version of the operating system) will be published, is presented in [22]. The authors state that the poor quality of NVD makes it difficult to build an accurate model for such prediction. Both of the last-mentioned researches ([21] and [22]) use NVD database as a dataset. Then the papers [18] and [19] both deal with the software vulnerability classification problem. Their training set contained information on confirmed vulnerabilities (gathered from the bug tracking system and the NVD database). However, both solutions lack context for the environment that the application is working in. For example, CVE-2017-7529, which describes the integer overflow vulnerability in the Nginx web application server (affected versions from 0.5.6 to 1.13.2), is detected by the automated scanning software based on a banner (by default the server header may contain information about the version of the WebServer). If an administrator manipulates this header, then the detected vulnerability may be correctly reported as well as treated as a false positive. Without the application or environmental context, it is impossible to classify such vulnerabilities in a correct manner.

There are also commercial products available. For example, MicroFocus released an application called Audit Assistant that takes as the input the list of vulnerabilities in the source code reported by the MicroFocus Fortify software.[2] As a result, the classification of an analyzed vulnerability is provided with information if the deficit could be exploited (the vulnerability can be used to get, for example, unauthorized access) or whether it is not an issue (false positives reported by a scanner). Unfortunately, the software vendors do not provide any information about the algorithms or mechanisms used within the Audit Assistant.

C. SUMMARY

The comparison of the related research in the field of vulnerability classification is presented in Table 1. It refers to the possibility of using the proposed solutions for security assurance within the CICD process.

TABLE 1. Comparison of related research.

Research	Classification Type	Vuln. Source	Possible to be used in CICD
[18]	BV or MV	Bug Trackers	Possibly yes
[19]	Vulnerability category	NVD	No
[20]	Vulnerability severity	NVD	No
[21]	Number of vulnerabilities in future	NVD	No
[22]	Time to next vulnerability	NVD	No
Audit Assistant	Confirmed and not confirmed vulnerability	Fortify	Yes
Presented solution	Confirmed and not confirmed vulnerability	All Vuln. Scanners	Yes

To summarize, when using state-of-the-art methods, it is clear that machine learning algorithms can be used to classify software vulnerabilities. The classification described in [18] can be utilized as a solution that will accelerate the process of prioritizing any identified security vulnerabilities, where issues of the BV type could be mitigated first, as they are easy to fix. Unfortunately, no available models or solutions allow to confirm whether the identified vulnerability can be exploited in the context of a given application, which is crucial in terms of building a security quality gateway or at least meets the organizational security policy. Consider, for example, an 'SSL/TLS Untrusted Certificate' vulnerability reported by the scanning software. Each time a vulnerability occurs in the scope of an application, it will be marked as BV. This kind of weak spot could violate the policy in some cases (e.g., when the application is accessed by end users directly) while in others, it might not (e.g., when the application is accessible via a proxy or load-balancing where

[1] https://www.bugzilla.org

[2] https://www.microfocus.com/media/data-sheet/fortify-audit-assistant-ds.pdf

SSL offloading is performed). On the other hand, commercial solutions like Audit Assistant need specific input. In this particular example, the input consists only of vulnerabilities detected by MicroFocus Fortify, which is a major limitation.

In contrast to existing approaches, the solution proposed in this paper uses potential vulnerabilities (gathered from automation software, such as vulnerability scanners) and utilizes neural network algorithms and the dataset processed using a Natural Language Processing (NLP) approach (where typically NNs yield the best results [23]) to classify the vulnerabilities as confirmed/unconfirmed. As an input for the introduced algorithm, any pre-compiled list of software vulnerabilities (with no limitation to a specific scanning solution) can be used. As a result, the software vulnerabilities will be classified either as marked to be fixed or not based on fields such as vulnerability description and application context.

III. SECURING THE SOFTWARE DELIVERY CHAIN

While developing modern IT systems, more and more software development teams use agile techniques that aim to expedite the software delivery processes. Thanks to the change in mindset, tooling ecosystem and application architecture, it is possible to significantly shorten the time for testing and delivery of applications. All this combined enables an increase in the number of deployments of a new version of software in the production environment. In 2011, Amazon was able to deploy a new version of its source code every 11.6 seconds (on average) thanks to adopting agile techniques.[3] By deployment, we mean the whole process from source code creation, testing, building an application from the source code, and then releasing it to the end customer.

A. SOFTWARE DEVELOPMENT LIFE CYCLE IN DEVOPS

DevOps methodology did not change the Software Development Life cycle [24], which consists of the following steps: *(i)* setting requirements where the development team gathers information about how the software or a feature should work; *(ii)* designing particular interfaces and their integration; *(iii)* software development to create source code that meets the criteria prepared earlier; *(iv)* implementation in a particular environment, so the developed software can be run there; and *(v)* verification where the development team thoroughly tests the software.

Agile and DevOps made a change in the way SDLC blocks are executed (see Figure 1) [25]. The ability to automate some steps made it possible to deliver new features on a scale that was never possible a few years ago. The tool ecosystem for such a purpose consists of the following elements:

- *Source code repository*: that contains the source code in a versioned manner. Modern solutions like GitHub and Gitlab [26] can also be used as bug tracking systems in which the user gathers information about bugs in the software or other deficits in the functionality of the developed solution.

[3]Velocity 2011: Jon Jenkins, ''Velocity Culture''

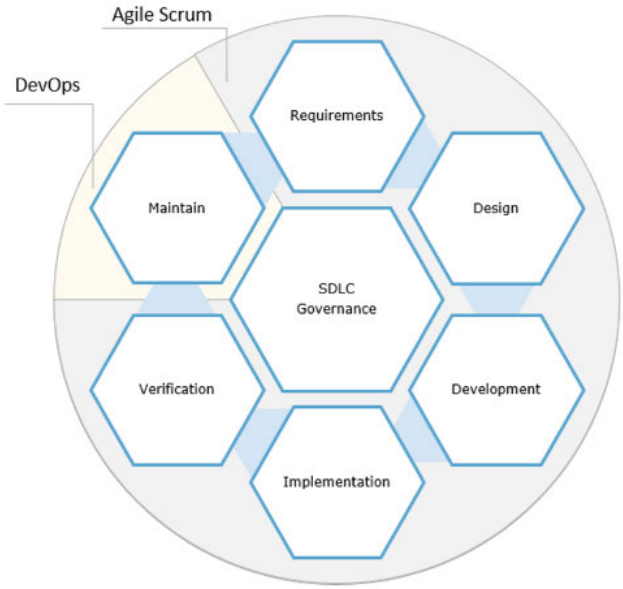

FIGURE 1. SDLC adoption in [23].

- *Continuous integration and continuous deployment*: tools that store the set of created pipelines in the form of a set of instructions that will be executed (to build ready to use applications) and when it should be executed (the scheduled event or triggered by a predefined action).
- *Testing*: for bug and problem-free applications, it should only be allowed to be deployed in the production environment where the end customers are dealing with the delivered software. To create an automated pipeline, there should be a complete test suite prepared for each application. Such a test should cover the areas of unit testing, functional acceptance testing, integration tests, performance tests, and more if needed.
- *Monitoring*: each event which is generated inside or by the infrastructure on which the application is working should be logged and stored properly. If the monitoring is properly configured, then the software developers can find the problems and bugs faster.

The simplified process, including tools and methodology, is presented in Figure 2. Each loop starts with the requirement issued to the developer who prepares a change within the code base and then sends it to the repository. The event of merging a newly delivered source code with the one which is already stored is triggered by a set of test suites. If the tests are completed successfully, the change is accepted and then the action of building application is launched. The generated application (if it passes the following test suite) is delivered for further testing in the proper environment. By design, everything from the start to the end is automatized. Note that what this process lacks is *security validation*. Many developers tend to forget that software security is one of the major factors that describe the quality of the solution. Unfortunately, the security inside a software delivery chain is often ignored, which can lead to major breaches.

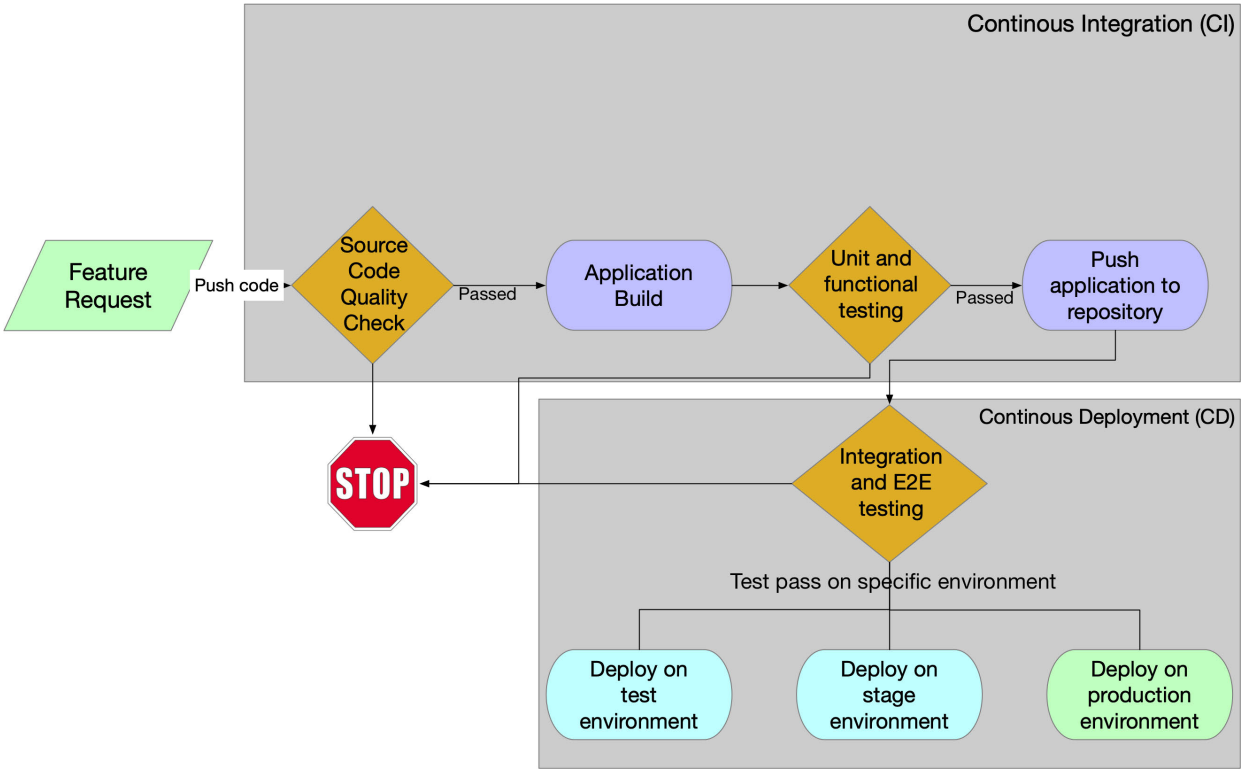

FIGURE 2. Automated software delivery chain.

B. SECURITY INSIDE A SOFTWARE DELIVERY CHAIN

The whole process described in the previous subsection does not cover the application security aspect of the software delivery automation. This phase should be described separately, as it is often ignored by the software development teams. There are a few reasons for that. Firstly, it is commonly believed that security verification inside a pipeline is complicated and the execution takes a large amount of time. Another reason is that the security findings obtained in an automated manner may contain many false positives and thus this can stop a pipeline from execution without a reason. For business owners, it is unacceptable to delay the implementation of a new feature due to a vulnerability that would later occur to be non-existing or unimportant. Both claims may be true, especially in places where the organizational culture is not at an appropriate level. Using properly integrated security tools, it is possible to ensure security during different stages of the delivery chain. The requirements for such tools are automatic functioning (e.g., using a predefined list of rules or policies), the ability to configure the scope and to run the security test remotely (e.g., via a REST API). Moreover, the execution time of such a scan should be as short as possible.[4]

There are several areas to be covered by the automated vulnerability testing during the described process:

- *Static Application Security Testing (SAST)* [27]: this kind of test requires access to the source code of an application. A vulnerability scanner analyzes the source code line by line looking for predefined patterns. Matches are reported as potential vulnerabilities. A SAST scan should be triggered by an event, which is the merging of a new code with the main source code repository.
- *Dynamic Application Security Testing (DAST)* [28]: this kind of test requires an application to run in a specific environment. The test logic executes several HTTP requests to verify if the application is vulnerable to well-known web attacks (e.g., from OWASP Top 10 list[5]). The DAST scan should be triggered by the event of deployment (of a new version of application) in the first test environment. In most cases, it is the development environment.
- *OpenSource Vulnerability Scanning*: requires access to the source code. In the first step, the test suite looks for OpenSource dependencies used within the codebase (e.g., using integration frameworks like *Apache Maven* or *npm*). The second step validates if there are known vulnerabilities already reported within a particular package. The Open Source Vulnerability scan should be triggered at the same time as the SAST scan as they both rely on the source code of an application.

[4]Note, that for each application, the allowed execution time is different, nevertheless, it should generally be counted in minutes rather than hours.

[5]https://owasp.org/www-project-top-ten/

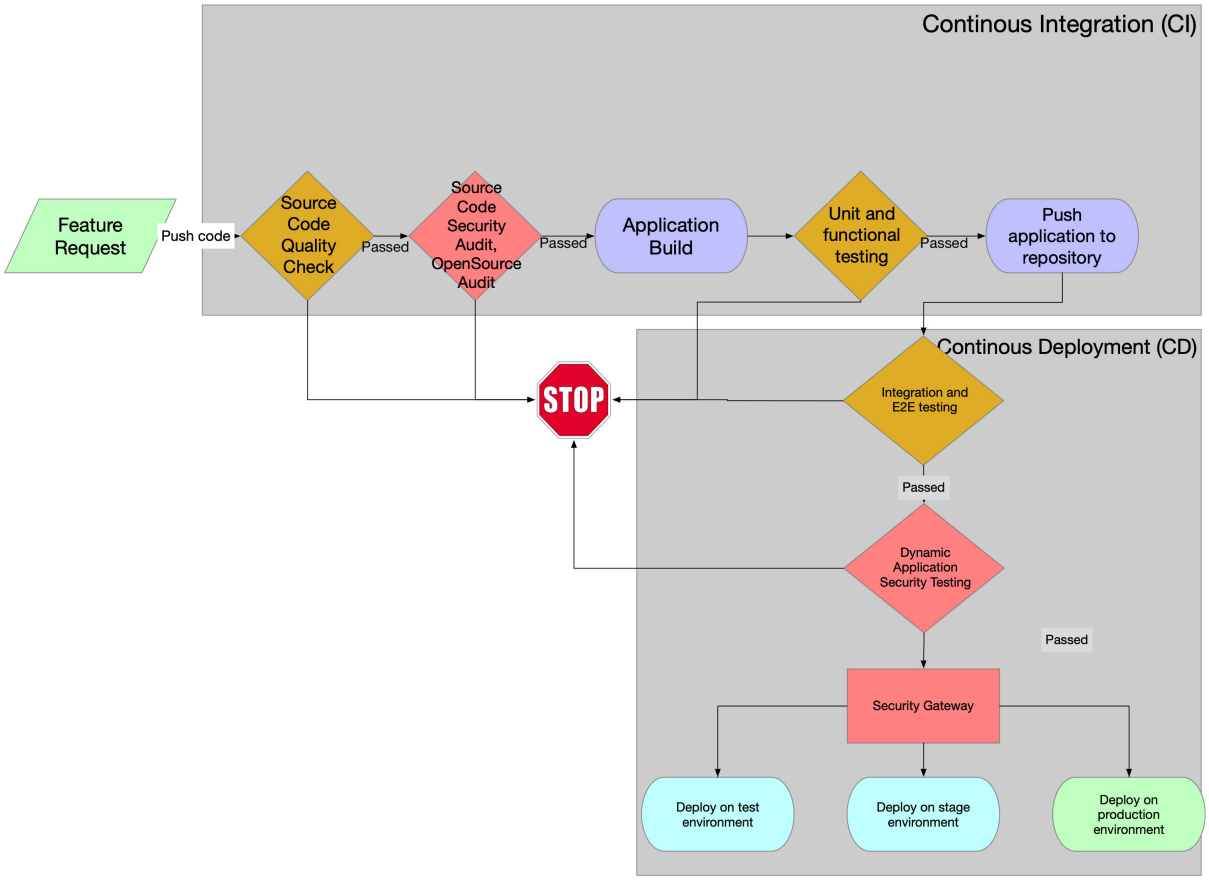

FIGURE 3. Software delivery chain with the security verification steps.

- *Network Vulnerability Scanning*: in most cases, this type of scanning is performed on assets with network interfaces. In general, such a scanner enumerates open ports on the tested assets. Then, it uses the obtained information to detect the service which is listening on a given port to verify whether there is any published vulnerability related to it, e.g., in the NVD database. The network vulnerability scan should be triggered when a new resource is added to the project or to the system infrastructure (e.g., when a new virtual machine is launched).
- *Image Vulnerability Scanning*: in most cases, the result of the Application Build step shown in Figure 2 is a docker image [29], which is built from a number of layers (one of those is an application that has just been built) that are executed using a shared host kernel. These layers can be created by particular services or even whole operating systems that use software packages. Image vulnerability scanning verifies if the layers that the image consists of have no security vulnerabilities. It should be triggered before pushing the newly built image to the repository.

All above-mentioned components combined can be utilized to modify the Automated Software Delivery Chain (Figure 2) to enable security within the Software Delivery process. Figure 3 illustrates their location in the automated pipeline that describes the software delivery chain where a particular security verification should be placed. SAST, OpenSource, and Image verification should be executed right after the source code has been sent to the repository. A decision about merging the newly created code with the existing one should be based on the results of a security test. Then it should be performed in the testing environment where the already built application is run. The decision to deploy the application to the production environment should consider the results of the security vulnerability testing.

Unfortunately, the information about the vulnerabilities required to be fixed has to be obtained manually by the IT security professionals. The outcome of the automated vulnerability scanning software could result in hundreds of potential issues that have to be analyzed. It is not always possible to rely on the severity, which is reported by an automated tool. For instance, the vulnerability ''SSL Certificate is issued by an untrusted authority'', which is detected in the context of the application and works in the internal ecosystem (available only to a specific number of users) is not as important as the same vulnerability discovered in an application available for external customers. We would like to

use this, as well as the fact that machine learning techniques have already been proven to be accurate for software vulnerability classification, to provide the context-aware software vulnerability classification. For this purpose, a concept of ''grading'' will be presented, i.e., the possibility to mark an issue as ''Confirmed and Relevant Vulnerability'' (CRV), which is a security issue reported by a vulnerability scanner and confirmed as relevant to be fixed, and ''Detected but Not Relevant Vulnerability'' (DNRV), which is a security issue discovered by a vulnerability scanner that, however, cannot be confirmed or is not relevant in an application context. The Authors strongly believe that by automating the process of labeling software or network vulnerabilities as ''CRV'' or ''DRNV'', they will significantly affect the level of security in the automated software delivery pipeline.

IV. CLASSIFICATION WITH MACHINE LEARNING

Text classification is used in various applications, e.g, in social media analysis [30], online advertisements (where to put proper ad) [31] or chatbots [32]. In the scope of Software Vulnerability Classification where deficit's description is an essential field to analyze (as the content may vary a lot based on the vulnerability source – see Section III-B, but the structure remains the same) there are several techniques which allow to classify the vulnerability. Text Mining (TM) [33] is commonly used when the goal of the task is to extract particular information from a large amount of text content. In this case, semantics in the text is not considered. Natural Language Processing (NLP) techniques are applied to teach the algorithm to understand the given text including semantics. There are several algorithms introduced to NLP techniques to prepare data before passing it to the training, such as Bag Of Words and tokenization. Since the goal of this work is to propose the context-aware vulnerability classification semantics is essential, thus only NLP techniques will be considered in this paper. Moreover, classification algorithms can be categorized in various ways, e.g., as supervised or unsupervised learning [34], binary or multiclass [35], and sentiment or content classification [36]. Considering the problem described in Section I, we consider supervised (the training dataset will be labeled), binary (output will be the grade of one of the two classes), and content (decision will be made based on the content of given sentences) classification. Furthermore, combining all above with the results obtained by other researchers in the field of software vulnerability classification (see Section II)it leads to a conclusion that algorithms such as Neural Networks (NN), Support Vector Machine (SVM), and Random Forrest (RF) [37] achieve significantly better results than other algorithms, e.g., Naive Bayes and k-Nearest Neighbors. Considering the above, we chose these three techniques (NN, SVM, and RF) for further analysis and experimental evaluation. These three techniques are also briefly described below:

- *Neural Network*: consists of three main elements. Input layer which is built of features selected for the algorithm to use. This layer does not perform any operations. Hidden layers are trying to put proper weights for a given input. In general, the output of each node inside a particular hidden layer can be described by $(y_j = \phi(\sum_i w_{ji}x_i + b_j))$ where (y_j) is an output of the neuron, (x_i) is the input, (b_j) is the bias, and (ϕ) is the activation function. The final layer is the output layer. In most cases, it can be treated just like the other hidden units while it uses a different activation function. Besides parameters like bias, weight or activation function, there are a number of hyper-parameters that include the layer size, number of nodes in a layer, learning rate for optimizing function, optimizing function, regularization or activation function. There are a number of elements that can be configured and tuned for an algorithm to work better, which make Neural Networks the most suitable for text classification purposes.
  There are multiple variants of Neural Networks, for instance, Recurrent Neural Network (RNN) [38], Convolutional Neural Network (CNN) [39], or hybrid [40]. RNN introduces the Memory Unit, which passes results from units within the same layer (see Figure 4). There are two types of Memory Units – Gated Recurrent Unit (GRU), which introduces two gates into an algorithm – reset and update. Through these gates, GRU decides whether to pass the calculated value to another unit or not. It just exposes the full hidden content without any control. The second type of Memory Unit is Long Short Term Memory (LSTM), which besides reset and update, introduces the output gate. This type of RNN, by design, performs better when memorizing longer sentences [41].
  CNN is commonly used for image classification due to the extracting possibilities. The usage of CNN in NLP problems has been recently proven to be efficient [42]. In this type of Neural Network, each input neuron is not connected to the output neuron of the next layer (see Figure 4). Instead, the convolution of the input is used to compute the output.
- *Random Forrest*: is an algorithm that uses a large number of uncorrelated decision trees [43]. The results of the decision for each decision tree are taken into consideration during the final decision making for the classifier.
- *Support Vector Machine*: is an algorithm that aims to find the margin-maximizing hyper-plane between classes. The main purpose for the SVM classifiers was binary classification, but making use of kernel functions allows SVM to perform operations of multiclass classification [44].

V. DATA COLLECTION

As hinted earlier, to address the shortcomings described in the previous section, a security quality gateway that would be able to provide reliable and useful information for the CICD tools about an application's security is desired. That is why, in response to these deficits, in this paper we design and develop a solution called *Mixeway*, which has been publicly

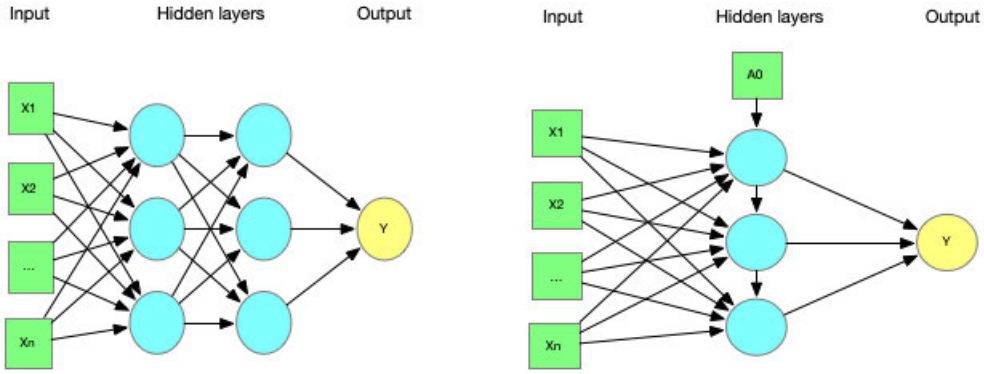

FIGURE 4. Overview of the general architecture for the simple neural network (left) and recurrent neural network (right).

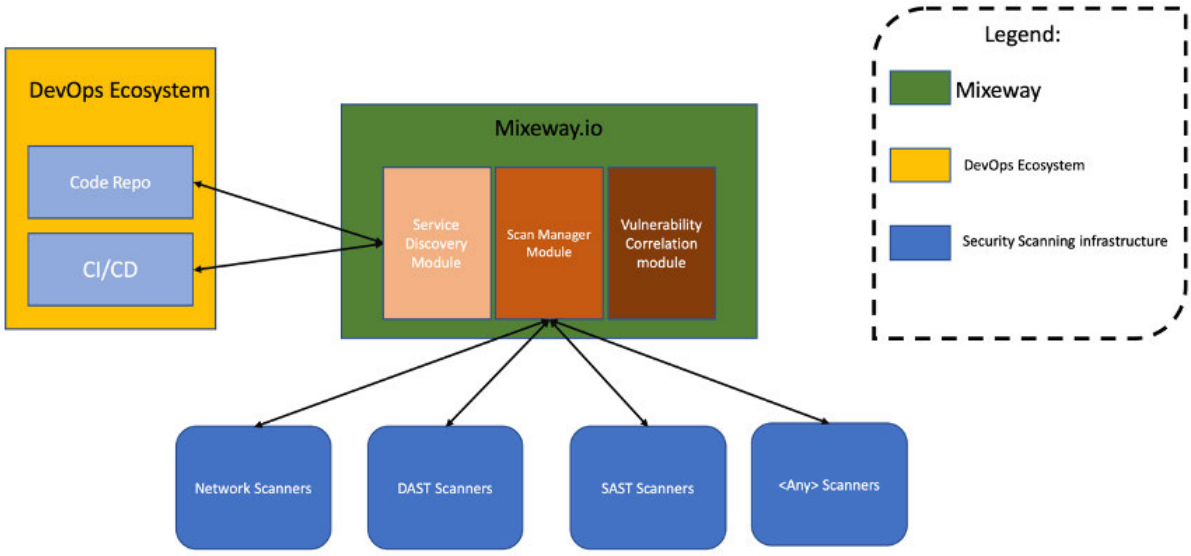

FIGURE 5. Architecture of the proposed solution.

shared with a community as an opensource on the GPL-3.0 license at GitHub.[6]

A. MIXEWAY STRUCTURE

The Mixeway architecture is illustrated in Figure 5 and it consists of three main modules.

- *Service Discovery Module*: this module is responsible for gathering information about the IT system in general. It enables integration with IaaS (Infrastructure as a Service) [45], i.e., whenever a new Virtual Machine (VM) is deployed in the scope of a particular IaaS tenant, the plugin obtains information, such as IP address, hostname, routing rules, or local firewall permissions. This data is used at a later stage for the configuration of the Network Vulnerability scanners and audit tools, which verify the network policy.
- *Scan Manager Module*: this module is responsible for interaction with various vulnerability scanners. Each time the CICD or any other tool (using REST API) requests a security test with a given scope (SAST, DAST, OpenSource, or Network Scanners), the Scan Manager Module uses the defined plugin to connect with a specific scanner, configure the scope of the test and later load vulnerabilities. At this stage, vulnerabilities are linked to the proper resources gathered from the Service Discovery Module.
- *Vulnerability Correlation Module*: as previously described, the vulnerability list discovered by automated tools is likely to contain a number of false positives. Additionally, different security scanners can report the same vulnerabilities. As a result, the complete report can be significantly lengthened and can contain many redundancies. That is why the Vulnerability Correlation Module is responsible for removing duplicates and performing classification to verify if

[6]https://github.com/Mixeway/MixewayHub

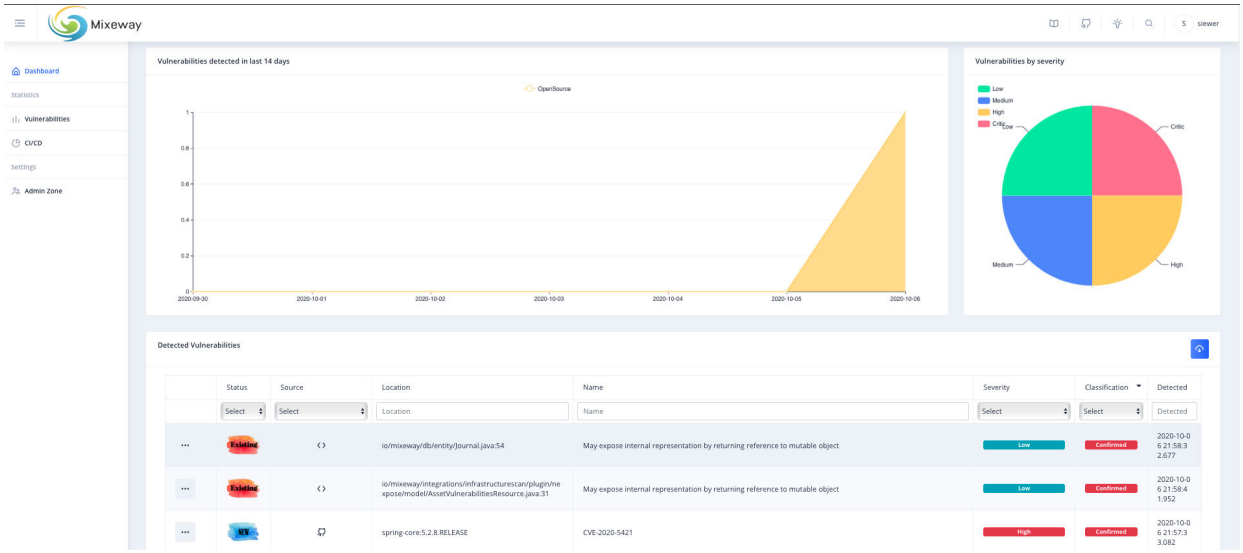

FIGURE 6. Graphical user interface of the mixeway project view (source: hub.mixeway.io).

a reported deficit is a software vulnerability or a false positive.

Note that Mixeway is also equipped with a graphical user interface (GUI) which, from the user point of view, makes it easy to use and configure (see Figure 6).

B. MIXEWAY IN THE AUTOMATED SOFTWARE DELIVERY PROCESS

Given that Mixeway is an orchestration tool for the functionality of the security quality gateway, thus it is strongly recommended to place it within the secured software delivery pipeline (see Figure 3). The main advantages of using the described software include:

- The software delivery process is independent of the Vulnerability Scanner used within the organization. The change of the scanning software will not affect the configuration of the CICD pipeline.
- Vulnerabilities from multiple types of scanners are stored in one place. Software developers and IT Security team members do not need to browse multiple locations to generate reports.
- Authorized team members are allowed to change the status of each vulnerability to "CRV" or "DNRV". As a result, software development teams are only notified about confirmed and important issues.
- Security Quality Gateway can be configured to fit the individual needs.

C. MOBILE NETWORK OPERATOR CASE STUDY OF MIXEWAY IMPLEMENTATION

The Mixeway software has been implemented in the ecosystem of a large mobile network operator to manage and accelerate IT security inside an automated software delivery chain. A variation in the workflow described in Figure 3 was designed and used. To be able to verify if a particular change could be merged with the main repository branch, static tests such as SAST and Open Source scans are being performed (Scan Manager module). DAST scans are executed during smoke tests [46]. Infrastructure scanning is executed in a scheduled manner (every 8 hours) or every time a new VM is being deployed in the name space of an IT system. To obtain information about newly created assets, the Service Discovery module is used. Finally, the Security Gateway serves information if the modification of the software code is secure (Vulnerability Correlation Module).

This way, hundreds or even thousands of vulnerability scans could be performed daily (the overall number depends on new feature releases).

VI. UTILIZED DATASET AND EXPERIMENTAL METHODOLOGY

In this section, we first present the datasets that we used during the performed evaluation. Moreover, we also present a feature extraction process. Finally, we introduce the methodology that we applied during the experimental phase.

A. UTILIZED DATASET AND FEATURE EXTRACTION

The developed solution, i.e., Mixeway, which was described in Section V-A, has been implemented and deployed in one of the largest Mobile Network Operators in Poland. Data related to vulnerabilities was collected for 12 months (between 2019 and 2020). This allowed to gather information about more than 50,000 software and network vulnerabilities. Security flaws were discovered in the e-commerce class IT systems. The detected vulnerabilities were verified by the authors during manual inspection, which resulted in the labeled dataset where each flaw has the "CRV" or "DNRV" tag. The data structure of the collected vulnerabilities is

TABLE 2. Data structure related to vulnerabilities as gathered by the mixeway.

Field Name	Separation Token	Description
Application Name	XXAN	Name of application in which context a vulnerability was detected. It could be an IP address when the deficit is affecting network object, URL when the issue was reported by the DAST scanning software and the name of the code repository if the vulnerability is present in the source code.
Application Context	XXAC	Context of the application. This field contains information about the source of a vulnerability, authentication type, network zone (internal or external) and the type of end user (employee or customer).
Vulnerability Name	XXVN	Name of the detected vulnerability taken from the scanning software. It could be the CVE name or a direct phrase like "Cross-Site Scripting Reflected".
Vulnerability Description	XXVD	This field contains a description of a vulnerability and information about why it is detected in the specific context.
Severity	XXSEV	Severity as reported by a scanner. The possible values are "Critical", "High", "Medium", "Low" and "Info".
Analysis	N/A	Verification performed by the IT security professional. It contains information whether a vulnerability has been confirmed ("1") or not ("0").

presented in Table 2. As the prepared dataset contains sensitive information about vulnerabilities of the existing software and network systems, the authors were not allowed to share it and thus it is not included in the Mixeway GitHub repository. However, it must be noted that the performed experiments can be repeated using any dataset which contains software and network vulnerabilities in the format described by MixewayVulnAuditor[7] in the Mixeway documentation. The choice of such a dataset results from the need to combine three aspects: a detected vulnerability, a specific web application (e.g., an URL, where an error has been detected,) and the context in which the application operates (who it is intended for and what type of data it processes). NVD database contains detailed information about vulnerabilities detected in a particular software that may be used in various contexts. For example, a particular version of the Content Management System (CMS) WordPress[8] may be used as an organization internal system for a few users to process irrelevant data. On the other hand, it may also be used as a landing page for customers. A vulnerability detected in such a CMS will be treated the same, while a possible attack on the application available for end-users might be much more severe.

[7]https://github.com/mixeway/mixewayvulnauditor
[8]https://wordpress.org

TABLE 3. Statistics of the utilized dataset.

Dataset attribute	No.
Dataset size	55665 records
Applications analyzed	109
SAST Vulnerabilities	38675
DAST Vulnerabilities	2282
OpenSource Vulnerabilities	755
Network Vulnerabilities	11953
CRV	19890
DNRV	33913

Thus, the NVD database cannot be used for the described purpose. Unfortunately, no organization is sharing such data on a scale which would enable us to prepare a valid dataset to be used for building an ML model.

Data containing long sentences, such as descriptions of vulnerabilities or application context, cannot be passed directly to the classification. There is a sequence of operations that need to be performed to prepare such a dataset. The vulnerability description contains information that is intended for the human operator to understand. It contains data about the identified issue (with explanation why it was reported) as well as details how to reproduce and fix it. To make the algorithm understand this piece of information the NLP preprocessing technique called ''tokenization'' is used. The tokenization process (which was proved to be efficient in previous research [47]) transforms a sentence into a numeric vector where each number corresponds to the word location in the prepared dictionary (see step 3 in Figure 7), which contains each word in a given dataset. Note that the input is multicolumn data and the output is a fixed length, one-dimensional vector of the tokenized words. The sequence of the described operations is presented in Figure 7. The first step is data collection. Such a table can be loaded from an exported file or a database directly. The classifier is not able to recognize the table columns, so specific tokens will act as a column separator. The second step is to add such separators (described in Table 2) with the addition of XXBOS and XXEOS, which are the ''beginning of a sentence'' and the ''end of a sentence'' tokens. Data prepared in this way is used to create a dictionary of words (note: the dictionary contains words detected in the dataset and corresponding numbers). In the next step, each column is tokenized using the prepared dictionary. The term ''tokenized'' means that the text is transformed into a sequence of numbers of corresponding words in the dictionary. Each column has to possess the same length (X_{column} which is the length of a particular column), so when the sequence length is shorter than X_{column} it is filled with 0 to match the proper dimension. Next, each prepared column is concatenated, so each example returns a one-dimensional sequence of numbers. Finally, such a prepared set of sequences is split into training and validation sets.

More information about the dataset is enclosed in Table 3 and Figure 8. As it can be seen, the majority of vulnerabilities in the dataset come from the SAST scanner, i.e., 72%, while the OpenSource Vulnerabilities represent only 2%.

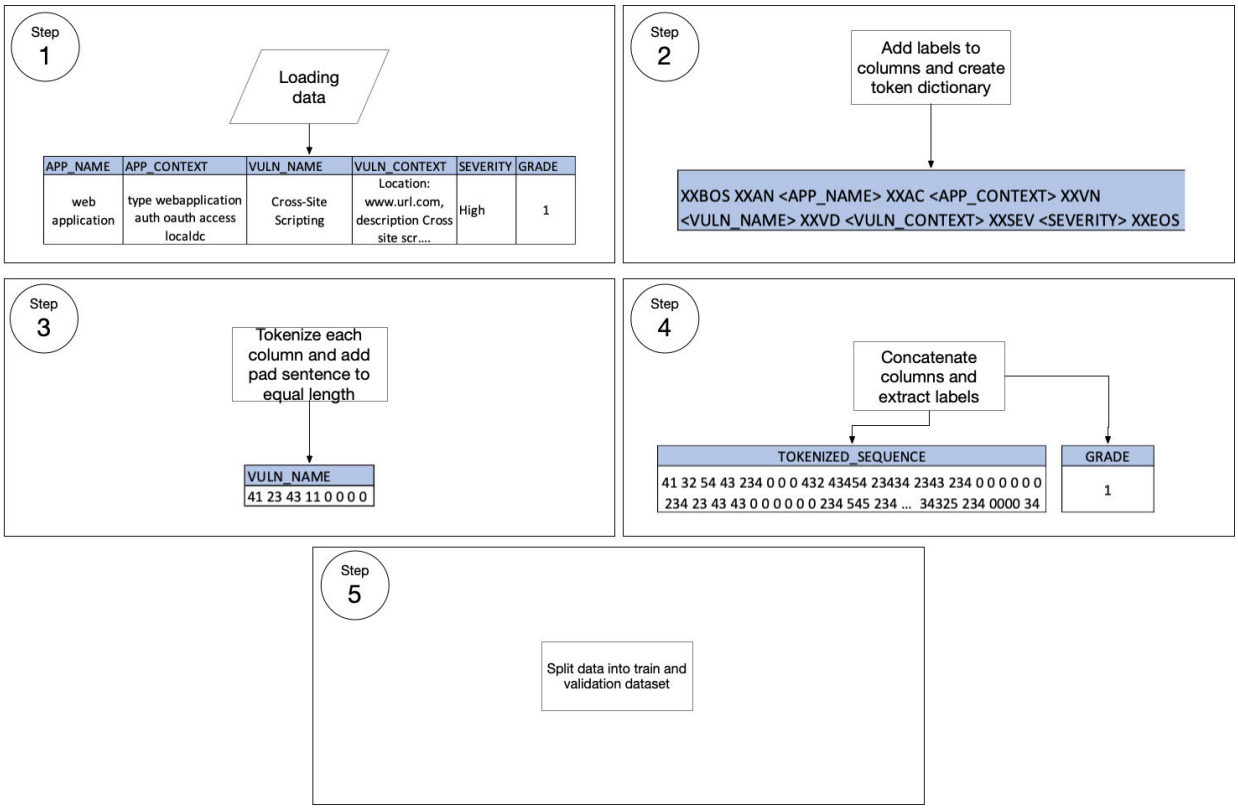

FIGURE 7. The sequence of operations to prepare data for classification).

Such uneven values should not be an issue for the implemented classifier. Further analysis of the data leads to the conclusion that the DAST scanners have greater accuracy than the SAST scanners, as more detected issues are classified as confirmed (62% vs. 31%).

B. EXPERIMENTAL METHODOLOGY

The experiments were performed on a machine with a CPU Intel Core i7-7700HQ, 32GB RAM, GPU NVIDIA GeForce GTX 1050. The experimental application was prepared using Python 3 with TensorFlow[9] and scikit-learn[10] library.

The five main steps performed during the experiments are illustrated in Figure 9. As already mentioned, the dataset used for experimental evaluation was gathered for over 12 months from the Mobile Network Operator's ecosystem by the Mixeway software (see section V-A). Next, the discovered software vulnerabilities were exported to be used for ML algorithms. Before using the data to train the model, preprocessing was performed (see section VI-A). The prepared data was then passed to the configured model (with parameters and hyper-parameters set to the values described below). The results obtained by the chosen classifiers are described in Section VII. Steps 4 and 5 are repeated for each classifier (NN, RNN, CNN, RF, and SVM). The created and trained model predicted a grade for the given sample. The outcome is validated and marked as "True Positive" (TP) – correct prediction of the CRV, "True Negative" (TN) – correct prediction of the DNRV, "False Positive" (FP) – incorrect prediction of CRV, and "False Negative" (FN) – incorrect prediction of the DNRV based on the provided "grade" label (supervised learning). These values (TP, TN, FP, and FN) are used to calculate metrics, such as accuracy (Equation 2), which expresses the probability of the classifier to detect the TP, precision (Equation 1) which is the ability of the classifier to reduce the number of detected FP values, recall (Equation 3) which is defined as an ability of the classifier to reduce the number of detected FN values and F1 (equation 4) which is the harmonic mean of recall and precision.

Configurations of the NN, which were tested to verify the software vulnerability classification, are enclosed in Table 4.

The values (such as the learning rate, regularization function, and a number of layers) described in Table 4 were obtained experimentally. For RNN, adding more layers resulted in a decrease in the classifier accuracy (and other metrics).

For the given experiment, RF with 300 decision trees were used from the scikit-learn library available in Python. During the experimental evaluation, we used SVM with a poly kernel and degree of eight. The parameters for RF and SVM were picked experimentally. A lower number of decision trees in

[9]https://tensorflow.org
[10]https://scikit-learn.org

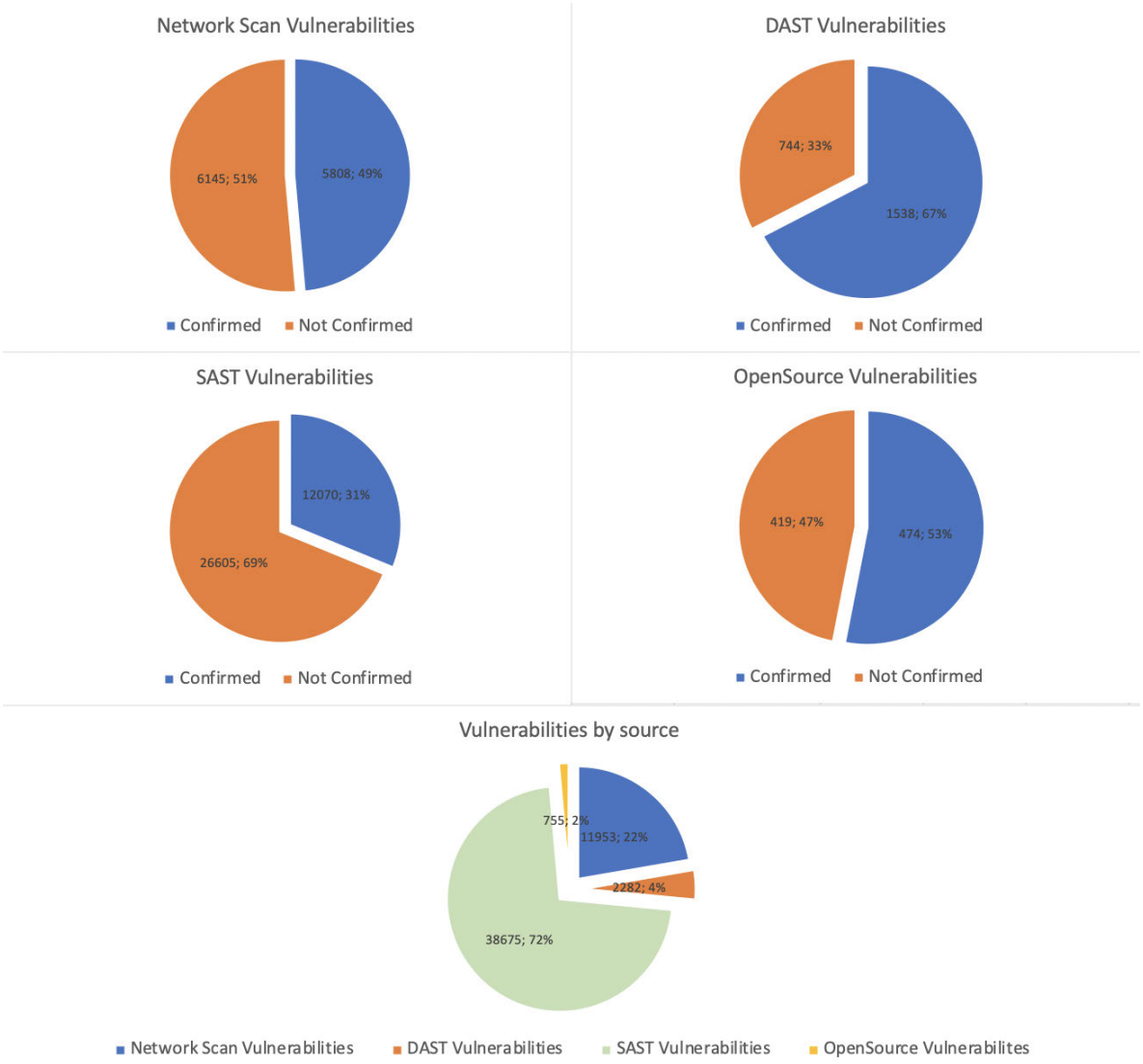

FIGURE 8. Overview of the dataset used for training and testing of the proposed model.

FIGURE 9. The structure of the performed experimental evaluation.

the RF algorithm resulted in a significant decrease in the metrics, such as accuracy, while using a number higher than 300 does not further improve the results. While examining the SVM algorithm, different kernel functions were tested. The polynomial kernel obtained the best results (while having the degree set to eight).

Cross validation was not used, as the applied libraries provided accurate and customizable functions, which ensures randomness in splitting the dataset into training and validation data.

TABLE 4. Configuration parameters for the ML algorithms.

Type	Layers	Activation function	Regularization
Basic NN	7	Hidden Layers - ReLU, Output - Sigmoid	L2 (0.1)
CNN	6	Hidden Layers - ReLU, Output - Sigmoid	L2 (0.1)
RNN with LSTM	5	Hidden Layers - ReLU, Output - Sigmoid	L2 (0.1)
RNN with GRU	3	Hidden Layers - ReLU, Output - Sigmoid	L2 (0.1)

To compare ML algorithms, we utilized the standard detection metrics calculated using the following equations:

$$precision = \frac{TP}{TP + FP} \tag{1}$$

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{2}$$

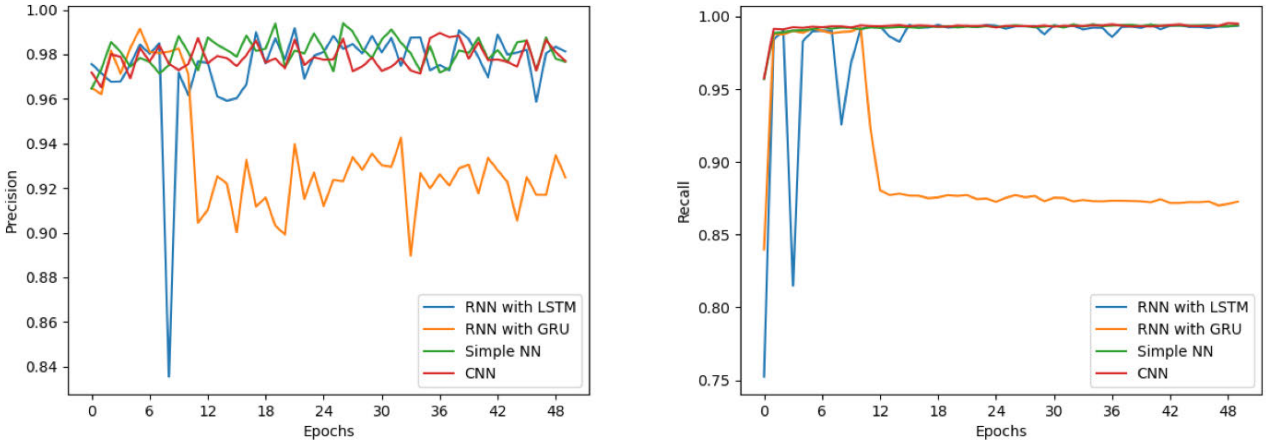

FIGURE 10. Precision (left) and recall (right) metrics obtained during the experiment – results are corresponding to metrics on the validation dataset.

$$recall = \frac{TP}{TP + FN} \tag{3}$$

$$F1 = 2\frac{precision * recall}{precision + recall} \tag{4}$$

VII. RESULTS

When combining the prepared input described in Section VI-A with the properly configured algorithms reviewed in Section VI, the experimental evaluation was performed to determine whether the ML algorithm can be used to effectively classify vulnerabilities in an application-context-aware dataset.

The obtained results are presented in Table 5. They show how the classifier deals with the validation dataset.

TABLE 5. Experimental results of the software vulnerability classification using different ML algorithms.

ML algorithm	Accuracy [%]	Precision [%]	Recall [%]	F1 score [%]	Learning time [s]
NN	0.98	0.96	0.99	0.98	16
RNN LSTM	0.99	0.98	0.98	0.98	390
RNN GRU	0.92	0.87	0.92	0.89	375
CNN	0.99	0.98	0.99	0.98	27
RF	0.98	0.99	0.98	0.99	10
SVM	0.85	0.77	0.99	0.87	644

In most cases, the evaluation of a classifier can be limited to comparing the accuracy score (especially for image classifiers). However, taking into consideration the nature of IT security, the values of FP and FN have to be carefully analyzed as well. Considering the use case described in Section III-B, i.e., the usage of classifiers in the evaluation of changes within the software inside the CICD pipeline, a large number of False Positive vulnerabilities may result in failing the whole pipeline for no reason. This kind of behavior is not acceptable for business owners. On the other hand, marking a vulnerability as DNRV when a serious flaw is present in the software, may result in a security breach, which is even more dangerous for the organization. For this reason, to compare the mentioned algorithm metrics, precision and recall have been used.

During the conducted experiment, we chose multiple variants of Neural Networks (RNN with LSTM and GRU, CNN, and basic NN), Random Forrest and SVM. The selected algorithms have already proven to be efficient in terms of software vulnerability classification in the existing literature [14], [15], [18], [19]. The SVM algorithm obtained the worst recall result (0.85 vs. 0.99) when compared to the other algorithms. A slightly better result was achieved in terms of precision for the Random Forest when compared to any variant of the Neural Network implementation. Neural Networks obtained better recall results (however, the difference of 0.01 is practically negligible). In terms of model preparation and learning time, the best results were obtained using Random Forrest – 10s. For NN, the basic implementation was achieved in 16s (for a single epoch) when compared to 390s for RNN LSTM, which is a considerable difference.

Figure 10 illustrates how the values of the recall and precision change in the NN model for a specific number of epochs (which is a single process of model training for the whole dataset). Note that the optimal number of epochs is 15 for each type of NN. More epochs do not increase the efficiency of the given models. On the other hand, if the number of epochs is lower than 15, this results in classifier instability (the fluctuations between the epochs at this stage are significant – 2%). The results for the RNN with GRU memory behaved in the opposite way and reached its best outcome below 15 epochs, while for the higher value both precision and recall dropped by 6% to 10%.

Regarding the precision and recall, the values obtained by the CNN, simple NN and RNN with LSTM were similar and the satisfying levels obtained accordingly were 98% and 99%, respectively. RNN with GRU obtained noticeably worse results for both metrics. Such behavior was expected due to the fact that the GRU unit does not perform well with long sentences (the training dataset was padded to a vector

TABLE 6. Results for TP, FP, FN, TN values for the selected ML algorithms.

ML algorithm	TP [No./%]	FP [No./%]	TN [No./%]	FN [No./%]
NN	5797 / 35.8	181 / 1.1	10185 / 63.1	10 / 0.1
RNN LSTM	5733 / 35.4	72 / 0.4	10294 / 63.6	74 / 0.5
RNN GRU	5187 / 32.1	760 / 4.1	9806 / 60.6	420 / 2.6
CNN	5763 / 35.6	109 / 0.7	10257 / 63.4	44 / 0.7
RF	10321 / 63.8	85 / 0.5	5658 / 34.9	109 / 0.7
SVM	8106 / 50.1	2324 / 14.3	5700 / 35.2	43 / 0.3

of 843 words). The best recall value (and the most stable) was obtained for the RNN with LSTM.

Considering the values that are used to calculate the metrics, e.g., precision (Table 6) it is possible to observe that in terms of FN – the LSTM, RF and SVM algorithms performed best. However, to properly choose the classifier to be used within the CICD pipeline (see Section III), both FN and FP should be considered. The SVM with FP count of 2324 is far less effective than the LSTM and RF (72 and 85).

The results for the RF and SVM classifiers are presented in Table 5. The presented outcomes contain average values for 10 epochs (single run of model training for the whole dataset). The difference between the particular runs is less than 1%, so the metrics obtained for these classifiers are quite stable. This proves that the data has been appropriately randomly divided into training and verification sets.

To ensure the validity of the presented research we deliberately manipulated the dataset. We chose examples with the same vulnerability and context but different applications. Labels for 50% of these examples were switched to the opposite value. This caused the dataset to contain contradictions that should not have occurred. With 10% of all examples affected, metrics for all classifiers deteriorated drastically. Such change in the dataset resulted in accuracy and precision drop by 13% for RNN, CNN, and RF and by 20% for SVM. Unfortunately, the mentioned dataset contains information about vulnerabilities identified in applications that work in the mobile network operator environment. It is possible that the prepared model would not be so accurate when used for the classification of deficits detected in applications which origin from, for example, banking industry. This issue would be a subject of our future work.

VIII. CONCLUSION

In this paper, we proposed a novel context-aware software vulnerability classification system – Mixeway – which relies on machine learning and NLP techniques and allows to automatically predict the vulnerability category. Moreover, we shared Mixeway with the security community as an open-source software available at GitHub.

In more detail, in this research, we investigated the efficiency and effectiveness of various machine learning algorithms for software vulnerability classification. If proved successful, such a classifier can be utilized as a security component inside a software delivery chain to make the whole CICD process more secure and to accelerate the work of both security and software development teams within organizations. The dataset used during the experimental evaluation of the proposed solution contained information about both the vulnerability (name, description) and the software in which the security issue was detected. To use such data, NLP techniques were applied. The efficiency of several types of Neural Network, Random Forrest and Support Vector Machine algorithms was verified.

The obtained experimental results prove that the ML algorithm can be used to classify software vulnerabilities in a context-aware dataset. Accuracy and recall values that were obtained for NN and RF (98%) are enough to configure the software delivery chain with confidence in the quality of the output from the classifier. When taking into consideration both metrics and the time needed for training, then the Random Forrest algorithm yielded the best results. However, the dataset gathered for the evaluation purpose contained software vulnerabilities for only 100 applications (and the majority are from e-commerce platforms). If the dataset would have contained more diverse software vulnerabilities, it is expected that the NN (especially RNN with LSTM variant) would obtain better outcomes due to a number of hyper-parameters that are possible to configure during the model preparation process. The results described in Section VII prove the efficiency of using the ML techniques for software vulnerability classification. Introducing grades, such as CRV and DNRV, could significantly facilitate the security assurance within the automated software delivery process. Using the proposed prototype as a component of the implemented Mixeway software could be desired as a part of a Vulnerability Correlation module (see Figure 5). Integration in the scope of exporting the stored software vulnerability to fit the learning process has already been designed.[11]

Trying to implement such a process in any organization may require bidirectional integration (between the described classifier and Mixeway) as once prepared, the model has to be continuously updated for several reasons. Firstly, more and more new types of software vulnerabilities are discovered each day. The other reason is that the application context can change.

Future work will be devoted to preparing ready to use scenarios for CICD processes (GitHub actions[12] and

[11] https://github.com/Mixeway/MixewayVulnAuditor – the source code of the proposed prototype

[12] https://github.com/features/actions

GitLab-CI[13]). The goal of such an action is to create a standard for a Security Quality Gateway inside an automated software delivery chain (presented in Figure 3) where the decision is based on a result provided by the presented classifier. In parallel, further samples from the vulnerability scanners will be collected (in particular from applications different than e-commerce), which will be later used in the training phase and could be utilized to help improve the model efficiency. Possibility of using vulnerabilities reported in the Hackerone[14] program will also be investigated, as reports available on the mentioned platform contain information about vulnerabilities detected in scope of specific web applications.

REFERENCES

[1] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "Relationship of DevOps to agile, lean and continuous deployment," in *Proc. Int. Conf. Product-Focused Softw. Process Improvement*. Cham, Switzerland: Springer, 2016, pp. 399–415.

[2] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," in *Proc. 24th Asia–Pacific Softw. Eng. Conf. (APSEC)*, Dec. 2017, pp. 466–475.

[3] S. A. I. B. S. Arachchi and I. Perera, "Continuous integration and continuous delivery pipeline automation for agile software project management," in *Proc. Moratuwa Eng. Res. Conf. (MERCon)*, May 2018, pp. 156–161.

[4] G. B. Simpson, "CI/CD software security automation," Sandia Nat. Lab, (SNL-NM), Albuquerque, NM, USA, Tech. Rep. SAND2018-11338PE, 2018.

[5] M. Qasaimeh, A. Shamlawi, and T. Khairallah, "Black box evaluation of Web application scanners: Standards mapping approach," *J. Theor. Appl. Inf. Technol.*, vol. 96, no. 14, pp. 4584–4596, 2018.

[6] G. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng, and X. Zhao, "Automatic classification method for software vulnerability based on deep neural network," *IEEE Access*, vol. 7, pp. 28291–28298, 2019.

[7] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning," in *Proc. 5th USENIX Workshop Offensive Technol. (USENIX Assoc.)*, 2011, p. 13.

[8] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2012, pp. 359–368.

[9] 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, p. 56, Aug. 2017.

[10] L. K. Shar and H. B. K. Tan, "Predicting common Web application vulnerabilities from input validation and sanitization code patterns," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Sep. 2012, pp. 310–313.

[11] L. K. Shar and H. B. K. Tan, "Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1767–1780, Oct. 2013.

[12] M. Zolanvari, M. A. Teixeira, L. Gupta, K. M. Khan, and R. Jain, "Machine learning-based network vulnerability analysis of industrial Internet of Things," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 6822–6834, Aug. 2019.

[13] M. Sajjad, A. Ahmad, A. W. Malik, A. B. Altamimi, and I. Alseadoon, "Classification and mapping of adaptive security for mobile computing," *IEEE Trans. Emerg. Topics Comput.*, vol. 8, no. 3, pp. 814–832, Jul. 2020.

[14] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, E. Antelman, A. Mackay, M. W. McConley, J. M. Opper, P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," 2018, *arXiv:1803.04497*. [Online]. Available: http://arxiv.org/abs/1803.04497

[15] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," 2018, *arXiv:1807.06756*. [Online]. Available: http://arxiv.org/abs/1807.06756

[13]https://docs.gitlab.com/ee/ci/

[14]https://www.hackerone.com

[16] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden, "Software vulnerability prediction using text analysis techniques," *Proc. 4th Int. Workshop Secur. Meas. Metrics (MetriSec)*, 2012, pp. 7–10.

[17] R. L. Russell, L. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, and M. W. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.

[18] M. D. M. Zulkernine and F. Jaafar, "An automatic software vulnerability classification framework," in *Proc. Int. Conf. Softw. Secur. Assurance (ICSSA)*, Jul. 2017, pp. 44–49.

[19] B. S. H. Li, M. Li, Q. Zhang, and C. Tang, "Automatic classification for vulnerability based on machine learning," in *Proc. IEEE Int. Conf. Inf. Automat. (ICIA)*, Yinchuan, China, Aug. 2013, pp. 312–318.

[20] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Sep. 2017, pp. 125–136.

[21] O. H. Alhazmi and Y. K. Malaiya, "Prediction capabilities of vulnerability discovery models," in *Proc. Annu. Rel. Maintainability Symp. (RAMS)*, 2006, pp. 86–91.

[22] S. Zhang, D. Caragea, and X. Ou, "An empirical study on using the national vulnerability database to predict software vulnerabilities," in *Proc. Int. Conf. Database Expert Syst. Appl.* Cham, Switzerland: Springer, 2011, pp. 217–231.

[23] P. Frijns, R. Bierwolf, and T. Zijderhand, "Reframing security in contemporary software development life cycle," in *Proc. IEEE Int. Conf. Technol. Manage., Oper. Decisions (ICTMOD)*, Nov. 2018, pp. 230–236.

[24] N. Alhirabi, O. Rana, and C. Perera, "Designing security and privacy requirements in Internet of Things: A survey," 2019, *arXiv:1910.09911*. [Online]. Available: http://arxiv.org/abs/1910.09911

[25] M. Virmani, "Understanding DevOps & bridging the gap from continuous integration to continuous delivery," in *Proc. 5th Int. Conf. Innov. Comput. Technol. (INTECH)*, May 2015, pp. 78–82.

[26] J. C. C. Ríos, K. Kopec-Harding, S. Eraslan, C. Page, R. Haines, and C. Jay, "A methodology for using GitLab for software engineering learning analytics," in *Proc. IEEE/ACM 12th Int. Workshop Cooperat. Hum. Aspects Softw. Eng. (CHASE)*, May 2019, pp. 3–6.

[27] A. Masood and J. Java, "Reframing security in contemporary software development life cycle," in *Proc. IEEE Int. Symp. Technol. Homeland Secur. (HST)*, Waltham, MA, USA, Nov. 2015, pp. 230–236.

[28] Y. Pan, "Interactive application security testing," in *Proc. Int. Conf. Smart Grid Elect. Automat. (ICSGEA)*, Xiangtan, China, Aug. 2019, pp. 558–561.

[29] S. M. Biradar, R. Shekhar, and A. P. Reddy, "Build minimal docker container using golang," in *Proc. 2nd Int. Conf. Intell. Comput. Control Syst. (ICICCS)*, Madurai, India, Jun. 2018, pp. 1–4.

[30] B. Y. Pratama and R. Sarno, "Personality classification based on Twitter text using naive Bayes, KNN and SVM," in *Proc. Int. Conf. Data Softw. Eng. (ICoDSE)*, Nov. 2015, pp. 170–174.

[31] X. Jin, Y. Li, T. Mah, and J. Tong, "Sensitive Webpage classification for content advertising," in *Proc. 1st Int. Workshop Data Mining Audience Intell. Advertising (ADKDD)*, 2007, pp. 28–33.

[32] P. Muangkammuen, N. Intiruk, and K. R. Saikaew, "Automated thai-FAQ chatbot using RNN-LSTM," in *Proc. 22nd Int. Comput. Sci. Eng. Conf. (ICSEC)*, Nov. 2018, pp. 1–4.

[33] M. Allahyari, S. Pouriyeh, M. Assefi, S. Safaei, E. D. Trippe, J. B. Gutierrez, and K. Kochut, "A brief survey of text mining: Classification, clustering and extraction techniques," 2017, *arXiv:1707.02919*. [Online]. Available: https://arxiv.org/abs/1707.02919

[34] J. C. Ang, A. Mirzal, H. Haron, and H. N. A. Hamed, "Supervised, unsupervised, and semi-supervised feature selection: A review on gene selection," *IEEE/ACM Trans. Comput. Biol. Bioinf.*, vol. 13, no. 5, pp. 971–989, Sep. 2016.

[35] G. Ou and Y. L. Murphey, "Multi-class pattern classification using neural networks," *Pattern Recognit.*, vol. 40, no. 1, pp. 4–18, Jan. 2007.

[36] T. Chen, D. Borth, T. Darrell, and S.-F. Chang, "DeepSentiBank: Visual sentiment concept classification with deep convolutional neural networks," 2014, *arXiv:1410.8586*. [Online]. Available: https://arxiv.org/abs/1410.8586

[37] J. Hartmann, J. Huppertz, C. Schamp, and M. Heitmann, "Comparing automated text classification methods," *Int. J. Res. Marketing*, vol. 36, no. 1, pp. 20–38, Mar. 2019.

[38] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Proc. 15th Annu. Conf. Int. Speech Commun. Assoc. (INTERSPEECH)*, 2014, pp. 338–342.

[39] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," 2014, *arXiv:1404.2188*. [Online]. Available: http://arxiv.org/abs/1404.2188
[40] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 2267–2273.
[41] W. Yin, K. Kann, M. Yu, and H. Schutze, "Comparative study of CNN and RNN for natural language processing," 2017, *arXiv:1702.01923*. [Online]. Available: https://arxiv.org/abs/1702.01923
[42] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification," 2016, *arXiv:1606.01781*. [Online]. Available: http://arxiv.org/abs/1606.01781
[43] L. Breiman, *Random Forests*, no. 1. Cham, Switzerland: Springer, 2001.
[44] F. Sebastiani, "Machine learning in automated text categorization," *ACM Comput. Surv.*, vol. 34, no. 1, pp. 1–47, Mar. 2002.
[45] N. K. Singh, S. Thakur, H. Chaurasiya, and H. Nagdev, "Automated provisioning of application in IAAS cloud using Ansible configuration management," in *Proc. 1st Int. Conf. Next Gener. Comput. Technol. (NGCT)*, Sep. 2015, pp. 81–85.
[46] N. Zhao and M. Shum, "Technical solution to automate smoke test using rational functional tester and virtualization technology," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, 2006, p. 367.
[47] X. Sun, X. Liu, J. Hu, and J. Zhu, "Empirical studies on the NLP techniques for source code data preprocessing," in *Proc. 3rd Int. Workshop Evidential Assessment Softw. Technol. (EAST)*, New York, NY, USA, 2014, pp. 32–39.

GRZEGORZ SIEWRUK was born in Siedlce, Poland, in 1990. He received the B.Sc. and M.Sc. degrees in telecommunications and became Ph.D. candidate in computer science from the Warsaw University of Technology, in 2017.

Since 2013, he has been working as an IT Security Expert with Orange Poland S.A. During the professional career, he implemented solutions related to source code security, audit, and orchestration of software vulnerability scanners during CICD pipeline, which leads to detecting and mitigating over 50 000 software vulnerabilities. Since 2017, his research has been concerned with the machine learning algorithms in vulnerability classification to provide faster and more accurate results for CICD processes.

WOJCIECH MAZURCZYK (Senior Member, IEEE) received the B.Sc., M.Sc., Ph.D. (Hons.), and D.Sc. (Habilitation) degrees in telecommunications from the Warsaw University of Technology (WUT), Warsaw, Poland, in 2003, 2004, 2009, and 2014, respectively. He is currently a Professor with the Institute of Computer Science, WUT. He is also working as a Researcher with the Parallelism and VLSI Group, Faculty of Mathematics and Computer Science, FernUniversität, Germany. His research interests include bio-inspired cybersecurity and networking, information hiding, and network security. He is involved in the technical program committee of many international conferences and also serves as a reviewer for major international magazines and journals. Since 2016, he has been the Editor-in-Chief of an open access *Journal of Cyber Security and Mobility*. He has been serving as an Associate Editor for the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY and a Mobile Communications and Networks Series Editor for *IEEE Communications Magazine*, since 2018.