Computers & Industrial Engineering 196 (2024) 110453

Contents lists available at ScienceDirect

Computers & Industrial Engineering

journal homepage: www.elsevier.com/locate/caie

System-wide vulnerability of multi-component software☆

Erol Gelenbe [a,b,c,*], Mert Nakıp [a], Miltiadis Siavvas [d]

[a] Institute of Theoretical and Applied Informatics, Polish Academy of Sciences, Baltycka 5, Gliwice, 44–100, Poland
[b] King's College London, Department of Engineering, WC2R 2LS, London, UK
[c] Université Côte d'Azur, CNRS I3S, 06100, Nice, France
[d] CERTH-ITI, Thermi, 570 01, Thessaloniki, Greece

ARTICLE INFO

Keywords:
Cybersecurity
Vulnerability prediction
Systems of interconnected software components
Associated Random Neural Network
Deep learning

ABSTRACT

In software systems comprised of many interconnected components, the vulnerability of each component will affect the vulnerability of other components and of the system as a whole. Existing techniques allow the quantification of the vulnerability of individual components taken singly, but the assessment of their vulnerability when they are interconnected or interdependent remains a challenge. The present work addresses this problem with a novel System-Wide Vulnerability Assessment (SWVA) framework for interconnected software components, based on an Associated Random Neural Network (ARNN) that estimates the system-wide vulnerability of all software components from known local vulnerabilities of individual components, and from their interconnections. The ARNN uses a problem-specific weight initialization, and learns from existing software system examples with a gradient-based deep learning algorithm. The ARNN is then used to assess the vulnerability of hitherto unseen software systems. The performance of the proposed ARNN-based SWVA framework is evaluated and compared against several well-known machine learning techniques on 13 different versions of a real-world software system with up to 11 components. The experimental results show the superior performance of the ARNN achieving above 85% median accuracy and good high scalability with respect to the number of connected software components.

1. Introduction

Secure information and communication technologies (ICT) may be difficult and expensive to build, but the cost of not meeting security requirements can also be extremely high (cisco, 2019; ciscopriv, 2019). Since software is pervasive across all ICT systems, there has been much interest in methods that can automatically inspect software to detect and reduce the security vulnerabilities that may result from design and implementation errors, from defects in the programming languages and tools that are used to develop software (Walden, Stuckman, & Scandariato, 2014), or from common programming mistakes (Siavvas, Gelenbe, Kehagias, & Tzovaras, 2018).

Thus, useful Static Code Analyzers exist to seek out vulnerabilities in software (sonarqube, 2024; verastat, 2020), and testing techniques (Pang, Xue, & Wang, 2017) have been suggested as a way to detect vulnerabilities during software production. Security is also a concern in programming languages, and vulnerabilities in the languages can be exploited by potential attackers. Thus several languages, e.g. OCaml, Java, and C# use static analysis and dynamic checks (Salka, 2005), while Rust is a system programming language specifically designed for safety-critical systems (Ding et al., 2017). While several organizations offer guidelines to enhance software security (cert, 2020; owasp, 2020; owaspguide, 2020; sans, 2020), it appears that some 85% of applications may contain vulnerabilities, while 10% or more of them may contain critical flaws that can lead to security breaches in software (Veracode, 2018).

Since the analysis of software vulnerability is time consuming (Jackson & Bennett, 2018), recent research has applied machine learning to this area (Dam et al., 2017; Nafi, Roy, Roy, & Schneider, 2020) using various methods (Neuhaus, Zimmermann, Holler, & Zeller, 2007; Shin, Meneely, Williams, & Osborne, 2010; Walden et al., 2014), including neural networks (Catal, Akbulut, Ekenoglu, & Alemdaroglu, 2017; Zhang, de Carnavalet, Wang, & Ragab, 2019), and deep learning (Filus, Boryszko, Domańska, Siavvas, & Gelenbe, 2021; Li et al., 2018).

Most of this prior work focuses on the vulnerability of individual programs or software components. Yet in a system where many individual components are logically interconnected as a system, the

☆ This research has been supported by the European Commission Horizon Europe – the Framework Programme for Research and Innovation (2021–2027) DOSS Project under Grant Agreement No: 101120270.

* Corresponding author at: Institute of Theoretical and Applied Informatics, Polish Academy of Sciences, Baltycka 5, Gliwice, 44–100, Poland.
E-mail addresses: gelenbe.erol@gmail.fr (E. Gelenbe), mnakip@iitis.pl (M. Nakıp), siavvasm@iti.gr (M. Siavvas).

https://doi.org/10.1016/j.cie.2024.110453
Received 13 May 2024; Received in revised form 30 July 2024; Accepted 3 August 2024
Available online 8 August 2024

vulnerability of each component can be modified by the vulnerability of other individual components in the system. Indeed, if only a few components have security vulnerabilities, then other components which individually do not have vulnerabilities may be compromised due to the propagation of security breaches from other components.

Thus, to address this major challenge, the present paper proposes and tests a novel System-Wide Vulnerability Assessment (SWVA) method based on the recently developed Associated Random Neural Network (ARNN) (Gelenbe & Nakip, 2023), which offers an automated machine learning approach to vulnerability testing for multi-component software systems.

We consider a software system composed of n separate individual sequential software components $\mathbf{S} = \{S_1, \ldots, S_n\}$. We assume that all of these components are resident on a single physical shared memory computing platform, and some or all components are interconnected to each other through procedure calls from one component to another. In this context, the proposed SWVA method uses:

- The individual vulnerability V_i of each component of the system is assessed by existing methods that are discussed below, and
- The graph that represents procedure calls or message passing between components, is defined by its $n \times n$ adjacency matrix A.

For instance, a procedure call can result in the return of data, or the transfer of code, to the component that initiates the call, so that the vulnerability of the component that initiates the call can be affected by the vulnerability of the component that is called. Therefore, in the sequel we will use an undirected graph A, that is represented by an adjacency matrix whose elements satisfy $A(i,j) = A(j,i),\ A(i,i) = 0,\ \forall\ i,j = 1,\ldots,n$.

As shown in Fig. 1, the SWVA predicts the updated vulnerability Likelihood Level L_i of all the system components, using the data about each component's individual vulnerability level V_i, and the matrix A.about the direct or indirect interactions between the components of S.

The SWVA uses the recently introduced ARNN model for Machine Learning (ML), with a specific initialization procedure and gradient-based deep learning algorithm. In this paper, the approach is applied and tested on a real-world software system consisting of 11 components, with 13 different versions, where each version has distinct internal connections among its components.

The remainder of this paper is organized as follows. The abbreviations used in this paper are listed in Table 1.

Section 2 briefly reviews the recent related research. Section 3 details the problem and presents the SWVA framework with our approach to solving the problem using a particular Machine Learning (ML) model, the ARNN. Section 4 presents the adaptation and usage of ARNN for the SWVA framework, including the learning algorithm.

Section 5 and Section 6 respectively describe the experimental work we have performed and the results obtained. Conclusions are detailed in Section 7, where we also present suggestions and plans for future work.

Table 1
List of abbreviations in order of appearance.

Abbreviation	Definition
ICT	Information and Communication Technologies
SWVA	System-Wide Vulnerability Assessment
ARNN	Associated Random Neural Network
ML	Machine Learning
NLP	Natural Language Processing
BERT	Bidirectional Encoder Representations from Transformers
RNN	Random Neural Network
JVM	Java Virtual Machine
WORA	Write Once, Run Anywhere
MLP	Multilayer Perceptron
KNN	K-Nearest Neighbours
Lasso	Least Absolute Shrinkage and Selection Operator
TNR	True Negative Rate
TPR	True Positive Rate
CPU	Central Processing Unit
PC	Personal Computer
GB	Gigabyte
RAM	Random-Access Memory

2. Related work

In this section, we review recent research in the two areas that we use in this paper: (1) Vulnerability prediction, and (2) ML research using Random Neural Networks (RNNs, which is the core ML model in this paper.

2.1. Vulnerability prediction

The purpose of vulnerability prediction is to identify security hotspots, i.e., software components that are likely to contain vulnerabilities. It focuses on the construction of vulnerability prediction models, which are typically based on ML and are able to predict the existence of vulnerabilities in software components using information retrieved from the components themselves. Several vulnerability prediction models have been proposed over the years, with the vast majority of them using information retrieved from source code mainly through static analysis or text mining.

The use of software metrics to identify security flaws was first examined in Shin and Williams (2008a, 2008b), which showed that complexity metrics are weak indicators of vulnerabilities. In Chowdhury and Zulkernine (2011), a framework for automatically predicting vulnerabilities based on software metrics was proposed, showing that software metrics can be adequately used for vulnerability prediction. The capacity of using statically collected software metrics in predicting the existence of software vulnerabilities has further been verified in Moshtari and Sami (2016) and Zagane, Abdi, and Alenezi (2020).

In text mining methods, the code tokens (i.e., words) are used as input in training ML models, and in Hovsepyan, Scandariato, Joosen, and Walden (2012), ML-based predictors exploit textual terms and their corresponding frequencies, which are extracted from the source code of software components. In Li et al. (2018) the VulDeePecker system is described with the word2vec embedding tool (Mikolov, Chen, Corrado, & Dean, 2013) that converts each token into a vector, and identifies vulnerabilities through deep learning.

In recent studies, it was examined whether graphical code representations may be used in vulnerability prediction. For instance, Zhou, Liu, Siow, Du, and Liu (2019) presented Devign, which is a broad graph neural network-based model. In this work, several graphical code representation formats (e.g., Control Flow Graphs, Abstract Syntax Trees, etc.) were combined to enhance the performance of deep learning model for vulnerability prediction. Following the advances in the field of Natural Language Processing (NLP), researchers have utilized pretrained Transformer-based models for the downstream task of vulnerability prediction. In Kim, Choi, Ahmed, Nepal, and Kim (2022), the authors developed a vulnerability prediction tool, called VulDeBERT, using the well-known Bidirectional Encoder Representations from Transformers (BERT) model (Devlin, Chang, Lee, & Toutanova, 2018) with problem-specific fine-tuning. Their findings suggested that the performance of VulDeBERT is better than that of VulDeePecker (Li et al., 2018). In addition, in Hanif and Maffeis (2022), VulBERTa is developed by fine-tuning BERT to predict vulnerabilities using source code. Thus much research has been conducted on vulnerability prediction, and several vulnerability prediction models have been developed including text mining-based models with advanced text processing techniques (such as BERT), that have recently attracted the greatest attention (Devlin et al., 2018; Fu & Tantithamthavorn, 2022; Hanif & Maffeis, 2022; Vaswani et al., 2017) and demonstrated promising results (Kalouptsoglou, Siavvas, Kehagias, Chatzigeorgiou, & Ampatzoglou, 2022).

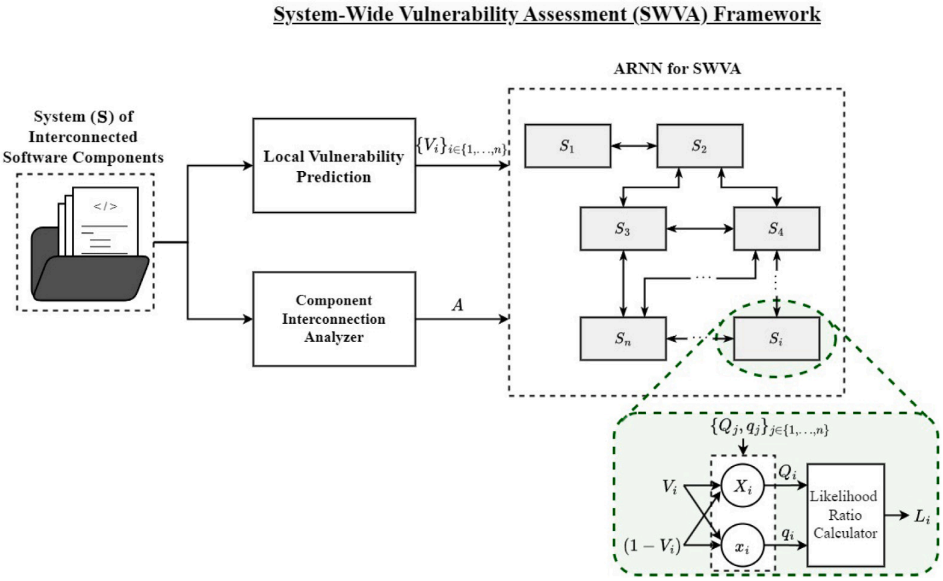

Fig. 1. The System Wide Vulnerability Assessment (SWVA) takes as inputs the individual component vulnerabilities V_i estimated with existing methods, as well as the interactions between different components expressed via an adjacency matrix A, and uses the ARNN to produce the updated vulnerability Likelihood Level metric L_i for each software component. SWVA learns from prior ground truth data with real software component vulnerability and interconnection data, and accurately predicts the updated Likelihood Level of the actual vulnerabilities L_i for (hitherto unseen)multi-component software systems that were not used in training the SWVA.

However, existing vulnerability prediction models detect the potential presence of vulnerabilities in a given software component based solely on information retrieved from the component itself, neglecting critical information from the wider system of interconnected components that affects the vulnerability status. That is, none of the current vulnerability prediction models considers the architecture of the wider software system (i.e. the interconnections between components) to make a prediction. In contrast, as a novel approach, we propose the SWVA framework, which collectively assesses the vulnerability of all interconnected components, considering both the intrinsic information from the components themselves and the architecture of the wider software system, together with the information from the interconnected components. To the best of the authors' knowledge, it is the first approach to performing a system-wide vulnerability assessment of interconnected components.

2.2. Random neural network

The Random Neural Network (RNN) (Gelenbe, 1989) was invented to mimic the spiking behaviour of mammalian brain neurons, and its gradient-based learning was designed both for feed-forward and recurrent structures (Gelenbe, 1993), and used for deep learning (Gelenbe & Yin, 2017). In addition, Levenberg–Marquardt learning (Basterrech, Mohamed, Rubino, & Soliman, 2011) and unique weights initialization (Timotheou, 2009) algorithms have also been proposed for the RNN.

In recent years, RNNs have been widely applied, and shown to perform better than other neural models in several areas (Aiello, Gaglio, Lo-Re, Storniolo, & Urso, 2005; Timotheou, 2010; Yin, 2018), such as vehicle classification (Hussain & Moussa, 2016), cyberattack detection and mitigation (Evmorfos, Vlachodimitropoulos, Bakalos, & Gelenbe, 2020; Latif et al., 2022; Latif, Idrees, Zou, & Ahmad, 2020), voice and video quality evaluation in the Internet (Ghalut & Larijani, 2014; Martínez, Moròn, Robledo, Rodríguez-Bocca, Cancela, & Rubino, 2008; Radhakrishnan & Larijani, 2011; Rubino, Tirilly, & Varela, 2006), the modulation of downlink traffic in LTE systems (Adeel, Larijani, & Ahmadinia, 2017), and optimal scheduling of video content (Ghalut & Larijani, 2018) in networks. Other applications have included the dynamic management of energy (Ahmad, Larijani, Emmanuel, Mannion, Javed, & Phillipson, 2017; Ahmad et al., 2020) and of the air comfort level (Javed, Larijani, Ahmadinia, Emmanuel, Mannion, & Gibson, 2017; Javed, Larijani, Ahmadinia, & Gibson, 2017) in smart buildings.

3. Problem statement and system design

We now present the problem statement that we consider in this paper and introduce the System-Wide Vulnerability Assessment (SWVA) framework. To this end, Fig. 1 displays the overall structure of the proposed approach, and presents the particular structure of an ARNN node.

The ARNN is a specific neural network architecture, that uses the mathematical structure of the Random Neural Network (Gelenbe, 1989). It is designed to represent the connectivity of a multi-component system, and the ARNN learning algorithm reflects the needs of SWVA. The neural weights of the ARNN represent the procedure calls and communications between individual software components, and the weights are learned so that they reflect the propagation of vulnerabilities among different components. Thus, the unique architecture of the ARNN enables it to learn the relationships between software components, and to assess the effects of the vulnerabilities of other components on each component. Earlier research using the ARNN for evaluating the vulnerability of interconnected Internet of Things (IoT) devices (Gelenbe & Nakip, 2023) has shown that the ARNN-based compromised device identification outperforms current state-of-the-art ML methods by achieving high average accuracy of the order of 95%.

In this paper we develop the recurrent ARNN architecture, with a specific weight initialization and gradient-based learning to address the SWVA problem.

As shown in Fig. 1, we consider each software component S_i – a part of a system **S** comprised of n components – whose *local* (individual) vulnerability V_i is known empirically. The local vulnerability V_i results from the presence of code constructs within S_i that cause vulnerabilities and is predicted via "Local Vulnerability Prediction" methods, as developed in Dam et al. (2018).

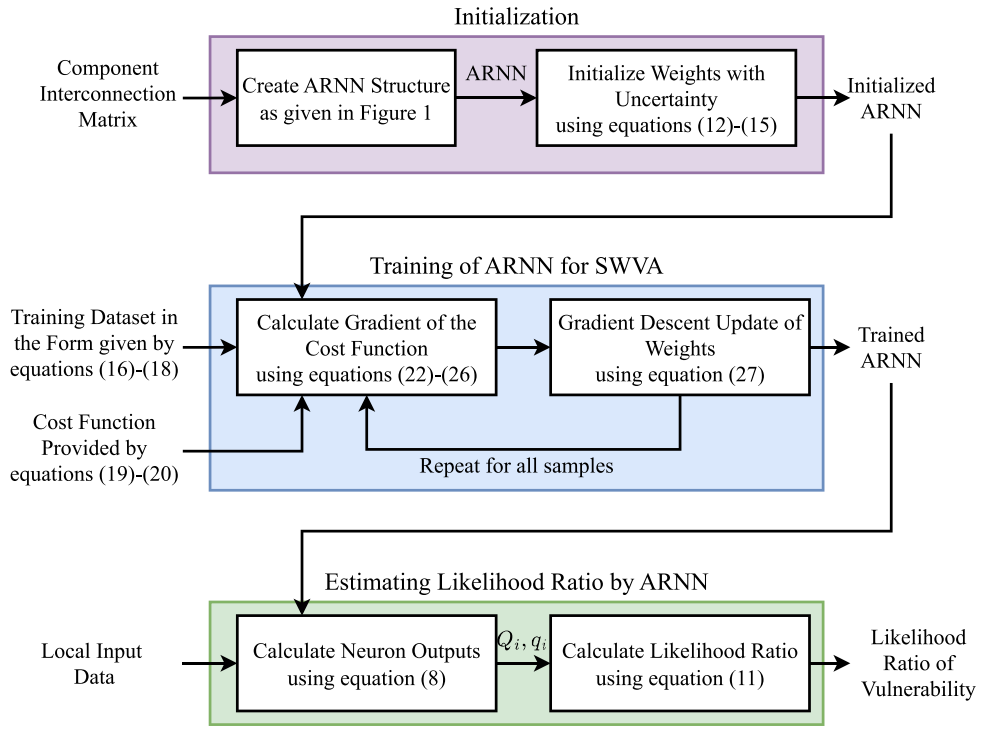

Fig. 2. Flowchart of the initialization, training, and estimation functionalities of ARNN.

On the other hand, the binary connectivity matrix $A = [A(i,j)]_{n\times n}$ represents the manner in which components can call each other via procedure calls, or transfer data to each other. By convention, we set $A(i,i) = 0$, while if $i \neq j$ and $A(i,j) = 1$, this means that during its execution, S_i can transfer data to S_j or activate it through a procedure call which may also require a data transfer, or the component S_j request results that are produced by S_i. Such exchanges can then transfer security vulnerabilities between components. In general, procedure call between two components will require a return of results from the called component to the calling component, so that transfers of data will typically be bilateral, and require that $A(i,j) = A(j,i),\ i \neq j$.

Thus, the SWVA method assesses whether any component in a software system is vulnerable, either due to its own local vulnerability, or due to the vulnerabilities of other components with which it exchanges data in the software system.

3.1. Local vulnerabilities

We use the variables $0 \leq V_i,\ v_i = 1 - V_i \leq 1$ to represent the local vulnerability of component S_i, which allows us to define the likelihood that it is *locally vulnerable*, by using the metric:

$$L_i^L = \frac{V_i}{v_i}. \tag{1}$$

We can also use a threshold value $0 < \alpha < 1$ so that we may consider that the software component S_i is locally vulnerable if:

$$\Big[V_i > \alpha \quad \text{or} \quad v_i < (1-\alpha)\Big],\ \textit{hence}\ L_i^L > \theta = \frac{\alpha}{1-\alpha}, \tag{2}$$

where θ is interpreted as the **threshold** of the local vulnerability likelihood calculator.

3.2. SWVA based on the ARNN

The SWVA method that we propose, is a ML-based approach to assess the impact of the local vulnerabilities of each component, together with the connection matrix A of procedure calls and communications between components, to assess the resulting overall vulnerability of each component and hence of the software system. Hence, the ARNN model uses the connectivity matrix A, and is trained with existing data to learn the mapping:

$$\textbf{ARNN}: \ \{V_i, v_i\}_{i\in\{1,\ldots,n\}} \rightarrow L = \{L_i\}_{i\in\{1,\ldots,n\}}, \tag{3}$$

where L is the system wide vulnerability likelihood regarding each component S_i of S.

4. ARNN for system-wide vulnerability assessment

We now detail the ARNN model that is used for the SWVA method. As shown in the right-hand part of Fig. 1, for each software component S_i, the ARNN has a pair of two competing RNN neurons X_i and x_i with internal states $K_i(t)$ and $k_i(t)$. X_i indicates that S_i is *vulnerable* while x_i indicates that S_i is *not vulnerable*. $V_i,\ v_i = 1 - V_i$ are the inputs to $X_i,\ x_i$, respectively. The stationary states:

$$Q_i = \lim_{t\to\infty} Prob[K_i(t) > 0],\ q_i = \lim_{t\to\infty} Prob[k_i(t) > 0], \tag{4}$$

serve as decision variables regarding the vulnerability of S_i, and are used to calculate the likelihood ratio L_i regarding the vulnerability of the software component S_i.

4.1. ARNN model building, initialization, and estimation

Figure 2 shows the manner in which the ARNN model is built. It is first constructed based on the component interconnection matrix provided by the software system program analyzer, as was shown schematically in Figure 1.

The ARNN weights are initialized as detailed in Section 4.1.2. Then, its weights are learned using the gradient descent algorithm that minimizes the cost function specific to the SWVA on a training dataset. The remainder of this section details the ARNN methodology, weight initialization, and learning algorithm.

4.1.1. The competing RNN neurons X_i and x_i

The ARNN architecture is a recurrent RNN, whose non-linear state equations are a given in Gelenbe (1993), with $2n$ neurons that are used for the n ARNN nodes. These state equations take the form:

$$\begin{aligned} Q_i &= \lim_{t\to\infty} Prob[K_i(t) > 0] \\ &= \frac{V_i + \sum_{j=1}^{n} W_{ji}^{+} Q_j}{v_i + \sum_{j=1}^{n} [W_{ij}^{+} + W_{ij}^{-}] + \sum_{j=1}^{n} w_{ji}^{-} q_j}, \end{aligned} \tag{5}$$

and similarly

$$\begin{aligned} q_i &= \lim_{t\to\infty} Prob[k_i(t) > 0] \\ &= \frac{v_i + \sum_{j=1}^{n} w_{ji}^{+} q_j}{V_i + \sum_{j=1}^{n} [w_{ij}^{+} + w_{ij}^{-}] + \sum_{j=1}^{n} W_{ji}^{-} Q_j}, \end{aligned} \tag{6}$$

where W_{ij}^{+} and W_{ij}^{-} are the excitatory and inhibitory connection weights from X_i to the neurons of ARNN node j, while w_{ij}^{+} and w_{ij}^{-} are the connection weights from x_i to neurons of node j. In the ARNN, these are restricted as follows:

$$W_{ij} = W_{ij}^{+} + W_{ij}^{-},\ w_{ij} = w_{ij}^{+} + w_{ij}^{-},\ \forall i, j \in \{1 \dots n\},\ i \neq j, \tag{7}$$

with constants $W_{ij},\ w_{ij} \geq 0$ representing the total firing rates of the neurons towards other connected neurons. This restriction has two advantages: it simplifies the learning algorithm since it only updates $W_{ij}^{+},\ w_{ij}^{+},\ i, j \in \{1, \dots, n\},\ i \neq j$, and it also avoids having very large or very small weights, resulting in better convergence.

Thus, we can then write:

$$\begin{aligned} Q_i &= \frac{V_i + \sum_{j=1}^{n} W_{ji}^{+} Q_j}{v_i + \sum_{j=1}^{n} W_{ij} + \sum_{j=1}^{n} [w_{ji} - w_{ji}^{+}] q_j}, \\ q_i &= \frac{v_i + \sum_{j=1}^{n} w_{ji}^{+} q_j}{V_i + \sum_{j=1}^{n} w_{ij} + \sum_{j=1}^{n} [W_{ji} - W_{ji}^{+}] Q_j}. \end{aligned} \tag{8}$$

4.1.2. Initializing the connection weights of the ARNN

Before applying the learning algorithm based on specific input data, we initialize the connection weights of the ARNN model in a manner which avoids the bias that may arise from a specific choice of the weights:

- The local vulnerability predictor provides "perfect ignorance" information, i.e. $V_i = v_i = 0.5$, for each component.
- The initial connection weights are chosen so that the ith node reflects "perfect uncertainty" with $Q_i = q_i = 0.5$.

To select the network weights that satisfy these two constraints, we use Eqs. (8) for Q_i and q_i:

$$\begin{aligned} Q_i &= 0.5 = \frac{V_i + 0.5 \sum_{j=1}^{n} W_{ji}^{+}}{V_i + \sum_{j=1}^{n} W_{ij} + 0.5 \sum_{j=1}^{n} [w_{ji} - w_{ji}^{+}]}, \\ or \quad & \sum_{j=1}^{n} W_{ij} = V_i - 0.5 \sum_{j=1}^{n} [w_{ji} - w_{ji}^{+} - 2W_{ji}^{+}], \end{aligned} \tag{9}$$

and

$$\begin{aligned} q_i &= 0.5 = \frac{V_i + 0.5 \sum_{j=1}^{n} w_{ji}^{+}}{V_i + \sum_{j=1}^{n} w_{ij} + 0.5 \sum_{j=1}^{n} [W_{ji} - W_{ji}^{+}]} \\ or \quad & \sum_{j=1}^{n} w_{ij} = V_i - 0.5 \sum_{j=1}^{n} [W_{ji} - W_{ji}^{+} - 2w_{ji}^{+}]. \end{aligned} \tag{10}$$

Since each pair of neurons $X_i,\ x_i$ is symmetric, we select $W_{ji}^{+} = w_{ji}^{+}$, $W_{ji} = w_{ji}$, and $W_{ij}^{+} = w_{ij}^{+} = 0.5\, w_{ij}$ so that:

$$\sum_{j=1}^{n} w_{ij} = V_i + 0.5 \sum_{j=1}^{n} [3w_{ji}^{+} - w_{ji}].$$

Due to the bilateral communication, we set $w_{ij} = w_{ji}$ when $A(i, j) = 1$. We then have:

$$\sum_{j=1}^{n} W_{ij} = \sum_{j=1}^{n} w_{ij} = \frac{4}{3} V_i,$$

so

$$\sum_{j=1}^{n} W_{ij}^{+} = \sum_{j=1}^{n} w_{ij}^{+} = \frac{2}{3} V_i \tag{11}$$

Let N_i denote the number of neighbours of component S_i, i.e. the number of components that are directly connected with S_i. We then initialize the ARNN weights as follows:

$$\begin{aligned} &When\ A(i, j) = 1\ and\ i \neq j:\ W_{ij}^{+} = W_{ij}^{-} = w_{ij}^{+} = w_{ij}^{-} = \frac{1}{3\, N_i}, \\ &Else\ if\ A(i, j) = 0\ or\ if\ i = j:\ W_{ij}^{+} = W_{ij}^{-} = w_{ij}^{+} = w_{ij}^{-} = 0. \end{aligned} \tag{12}$$

Note that the inter-neuron weights are zero if $A(i, j) = 0$, and we also have zero weights from a neuron to itself.

After initialization, for all neuron pairs $i, j,\ i \neq j$ with $A(i, j) = 1$, the weights $W_{ij}^{+},\ w_{ij}^{+}$ determined via the learning algorithm given in Section 4.2.

4.1.3. Computing the likelihood ratio L_i of the overall vulnerability S_i

We now show how the Likelihood Ratio L_i for the Overall Vulnerability of Component S_i can be computed from the ARNN. Using Gelenbe (1993), we know that if the solution to the $2n$ Eqs. (5) and (6) are such that $0 \leq Q_i,\ q_i < 1$ for $i \in \{1, \dots, n\}$, then the joint stationary distribution of network state satisfies:

$$\begin{aligned} &\lim_{t\to\infty} Prob[K_1(t) = C_1, \dots, K_n(t) = C_n, \\ &\qquad k_1(t) = c_1, \dots, k_n(t) = c_n] \\ &= \prod_{i=1}^{n} Q_i^{C_i} (1 - Q_i)\, q_i^{c_i} (1 - q_i), \end{aligned} \tag{13}$$

and the probability that ARNN predicts the component S_i is vulnerable, is:

$$P_i^{V} = \lim_{t\to\infty} Prob[K_i(t) > k_i(t)] = \frac{Q_i(1 - q_i)}{1 - q_i Q_i} \tag{14}$$

Similarly, we obtain that the probability that component S_i is not vulnerable:

$$P_i^{v} = \lim_{t\to\infty} Prob[K_i(t) < k_i(t)] = \frac{q_i(1 - Q_i)}{1 - q_i Q_i}, \tag{15}$$

Accordingly, one computes the Likelihood Ratio regarding the vulnerability of component S_i as:

$$L_i = \frac{P_i^{V}}{P_i^{v}} = \frac{Q_i(1 - q_i)}{q_i(1 - Q_i)} \in [0, +\infty)\ . \tag{16}$$

Thus for some threshold $\Theta > 0$, $L_i \geq \Theta$ supports the hypothesis that S_i is vulnerable in the presence of the interconnected components, while if $L_i < \Theta$, then the ARNN infers the opposite, while $L_i = \Theta$ would indicate that the ARNN is unable to reach a decision.

4.2. Training the ARNN for SWVA

The training dataset Δ contains a pair of n-vectors Ψ_z and G_z, for each data sample $z \in \{1, \dots, Z\}$ that corresponds to a version of the software system **S**:

$$\Delta = \{\Psi_z, G_z\}_{z \in \{1, \dots, Z\}}, \tag{17}$$

where Ψ_z is the vector of input pairs $(V_{i,z}, v_{i,z})$ for all components $i \in \{1, \dots, n\}$ for sample z such that:

$$\Psi_z = \Big[(V_{1,z}, v_{1,z}), \dots, (V_{i,z}, v_{i,z}), \dots, (V_{n,z}, v_{n,z})\Big], \tag{18}$$

and G_z is the vector of ground truths $G_{i,z}$ for the likelihood ratio of vulnerability for all components $i \in \{1, \dots, n\}$ for sample z such that:

$$G_z = \Big[G_{1,z}, \dots, G_{i,z}, \dots, G_{n,z}\Big]. \tag{19}$$

4.2.1. The learning algorithm

First, let $Q_{i,z}$ and $q_{i,z}$ respectively denote the values of Q_i and q_i for the input values in Ψ_z, i.e. $Q_{i,z} = Q_i(\Psi_z)$ and $q_{i,z} = q_i(\Psi_z)$. Then, the resulting likelihood ratio (denoted by $L_{i,z}$) for the sample z in the dataset Δ is calculated as

$$L_{i,z} = \frac{Q_{i,z}(1-q_{i,z})}{q_{i,z}(1-Q_{i,z})}, \tag{20}$$

and the cost function to be minimized takes the form:

$$E_z = \frac{1}{2}\sum_{i=1}^{n}(L_{i,z} - G_{i,z})^2. \tag{21}$$

The steps of the "Learning Algorithm" are then as follows:

- **Initialize the ARNN** as described in Section 4.1.2.
- **For** $z = 1$ **to** $z = Z$:
- **For** $i = 1$ **to** $i = n$:
- **Compute** $Q_{i,z}$, $q_{i,z}$, and $L_{i,z}$
- **Compute** E_z using (21)
- **Update** the connection weights of ARNN for components S_i using Gradient Descent with the cost function E_z:

$$W^+_{J,K} \leftarrow W^+_{J,K} - \eta\frac{\partial E_z}{\partial W^+_{J,K}}, \tag{22}$$
$$w^+_{j,k} \leftarrow w^+_{j,k} - \eta\frac{\partial E_z}{\partial w^+_{j,k}},$$

where $\eta > 0$ is the learning rate.

4.2.2. Gradient descent of the cost function

We first obtain the derivative of the cost function for each z with respect to the ARNN weights $W^+_{J,K}$ and $w^+_{j,k}$ as

$$E_z^{J,K} \equiv \frac{\partial E_z}{\partial W^+_{J,K}} = \sum_{i=1}^{n}(L_{i,z} - G_{i,z})\frac{\partial L_{i,z}}{\partial W^+_{J,K}}, \tag{23}$$
$$E_z^{j,k} \equiv \frac{\partial E_z}{\partial w^+_{j,k}} = \sum_{i=1}^{n}(L_{i,z} - G_{i,z})\frac{\partial L_{i,z}}{\partial w^+_{j,k}}.$$

In order to compute the corresponding terms, we can easily show that:

$$\frac{\partial L_{i,z}}{\partial W^+_{J,K}} = L_{i,z}\Big[\frac{Q^{J,K}_{i,z}}{Q_{i,z}(1-Q_{i,z})} - \frac{q^{J,K}_{i,z}}{q_{i,z}(1-q_{i,z})}\Big], \tag{24}$$
$$\frac{\partial L_{i,z}}{\partial w^+_{j,k}} = L_{i,z}\Big[\frac{Q^{j,k}_{i,z}}{Q_{i,z}(1-Q_{i,z})} - \frac{q^{j,k}_{i,z}}{q_{i,z}(1-q_{i,z})}\Big],$$

where

$$Q^{J,K}_{i,z} \equiv \frac{\partial Q_{i,z}}{\partial W^+_{J,K}}, \quad q^{J,K}_{i,z} \equiv \frac{\partial q_{i,z}}{\partial W^+_{J,K}}, \tag{25}$$
$$Q^{j,k}_{i,z} \equiv \frac{\partial Q_{i,z}}{\partial w^+_{j,k}}, \quad q^{j,k}_{i,z} \equiv \frac{\partial q_{i,z}}{\partial w^+_{j,k}}.$$

Now, let us define the vectors of size $[1 \times n]$:

$$L_z = \Big(L_{i,z}\Big)_{i\in\{1,\ldots,n\}}, \tag{26}$$
$$L_z^{J,K} = \Big(L^{J,K}_{i,z}\Big)_{i\in\{1,\ldots,n\}}, \quad L_z^{j,k} = \Big(L^{j,k}_{i,z}\Big)_{i\in\{1,\ldots,n\}},$$
$$Y_z^{J,K} = \left(\frac{Q^{J,K}_{i,z}}{Q_{i,z}(1-Q_{i,z})} - \frac{q^{J,K}_{i,z}}{q_{i,z}(1-q_{i,z})}\right)_{i\in\{1,\ldots,n\}},$$
$$Y_z^{j,k} = \left(\frac{Q^{j,k}_{i,z}}{Q_{i,z}(1-Q_{i,z})} - \frac{q^{j,k}_{i,z}}{q_{i,z}(1-q_{i,z})}\right)_{i\in\{1,\ldots,n\}},$$

so that if we denote the Hadamard product of vectors a and b by $a \circ b$, we can write:

$$L_z^{J,K} = L_z \circ Y_z^{J,K}, \quad L_z^{j,k} = L_z \circ Y_z^{j,k}. \tag{27}$$

The **gradient descent update** of $W^+_{J,K}$, $w^+_{j,k}$ for sample z then becomes:

$$W^+_{J,K} \leftarrow W^+_{J,K} - \eta(L_z - G_z)^\tau\, L_z^{J,K}, \tag{28}$$

$$w^+_{j,k} \leftarrow w^+_{j,k} - \eta(L_z - G_z)^\tau\, L_z^{j,k},$$

where M^τ is the Transpose of matrix M.

5. Experiments using a real-world software system

In order to evaluate the performance of the proposed SWVA framework, we consider a real-world software system, called GitHubCrawler, and collect data using this system. During the data collection using GitHubCrawler system, we first identify the interconnections between software components, i.e. extract the $[n \times n]$ matrix A_c, for each committed version c of this system. Then, for each component in software with version c, we utilize a local vulnerability detector to compute inputs and the ground truth.

5.1. The software system

The GitHubCrawler system has been developed as part of a European Union research project SDK4ED (*EU HORIZON Project SDK4ED*, 2020). This system, which is written in Java and a core element of the Dependability Toolbox (DepTool, 2024) of the SDK4ED Platform (Kehagias, Jankovic, Siavvas, & Gelenbe, 2021), is responsible for (1) cloning both public and private software repositories from GitHub, (2) compiling them with MAVEN (Maven, 2024), and (3) crawling GitHub, in order to retrieve software repositories that satisfy user needs based on submitted queries (programming language, stars, etc.). In earlier research, it has been used by the Quantitative Security Assessment (Siavvas, Kehagias, Tzovaras, & Gelenbe, 2021) and Vulnerability Prediction (Filus et al., 2021; Filus, Siavvas, Domańska, & Gelenbe, 2020) micro-services of the Dependability Toolbox to access the GitHub repositories for security analysis.

The GitHubCrawler system has 13 committed versions in its development history, and its latest version consists of 11 software components interconnected to perform its main functions. Note that earlier versions of this system consist of fewer (4 to 10) components. The selected software system has undergone a thorough security code review and actual vulnerabilities were identified in some of its components. This information was particularly useful for the present work for constructing the ground truth that is required for building and evaluating the ARNN model.

As already stated, for the purposes of the present study, the various components of the selected software system operate on the same local machine (i.e. on a single physical shared memory platform), where some or all of its components are interconnected to each other through procedure calls. Since the source code is written in the high level programming language Java, the analysis benefits from a higher level of abstraction, that focuses on the system's logic and functionality, rather than on low-level hardware specific operations. It should be also noted that, although the various components operate on the same local machine, the analysis is not restricted to a particular operating system, since Java applications operate on the Java Virtual Machine (JVM), which enables applications to run across various operating systems (cross-platform operation). Java adopts the "Write Once, Run Anywhere (WORA)" philosophy, which is achieved by compiling the source code into a bytecode that can be executed on any hardware platform or operating system that has compatible JVM. JVM acts as an abstraction layer between the application and the underlying OS, handling the application's execution in a way that makes it compatible across different platforms.

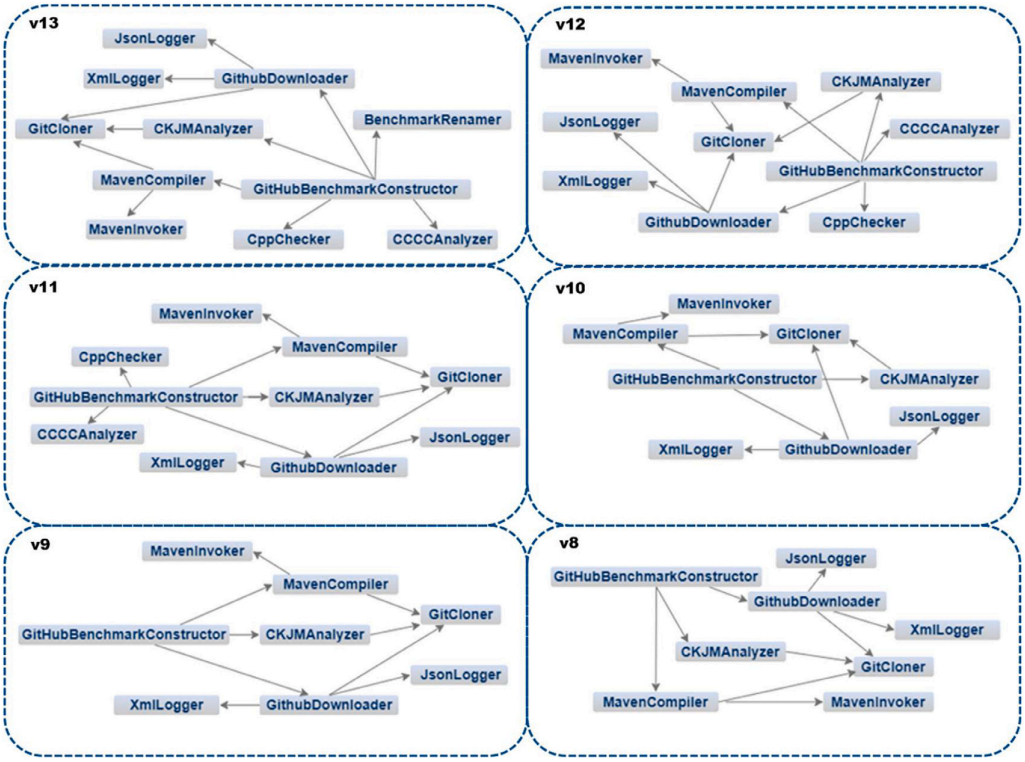

Fig. 3. The graphs with the interconnections between the components of the six latest versions of the GitHubCrawler software system, as produced by the Dependency Viewer plugin.

5.2. Interconnection of software components

Recall from Section 3 and Fig. 1, that it is first necessary to identify the interconnections between all components, i.e. to extract the connection matrix A_c in software version c, for either training or execution of the ARNN model. To this end, we use the Eclipse plugin named Dependency Viewer (*VIEWER*, 2024) to provide a graph of the interconnections between the components of the given software system.

During our experimental work, in order to construct the dataset that is used for performance evaluation, the Dependency Viewer was employed for each of the 13 versions of the GitHubCrawler system to obtain the interconnections between the components.

The component interconnection graphs of the 6 latest versions (v8–v13) of GitHubCrawler are illustrated in Fig. 3 . These graphs were used to construct the connection matrix of each committed software version c, denoted by A_c. Note that we manually inspected both the graphs and source code of each software version to verify the correctness of interconnections between components and to add links missed by the Dependency Viewer tool.

5.3. Local vulnerability prediction

In order to collect a dataset of the local vulnerability predictions within the SWVA framework, we use 13 different vulnerability prediction models developed as part of the IoTAC Software Security by Design Platform (Siavvas et al., 2024). That is, for each committed version c of the GitHubCrawler system, we collect a dataset comprised of 13 samples. The local component based vulnerability predictions are based on text mining of the word sequences, which have demonstrated promising results in recent work (Dam et al., 2018).

For each software component $S_{c,i}$ in version c of the GitHubCrawler, we first parse its source code to extract word sequences, and then calculate its local vulnerability score $V_{c,i,z}$ using the local vulnerability predictor that corresponds to the sample z. Recall that we set $v_{c,i,z} = 1 - V_{c,i,z}$.

5.4. Ground truth for the likelihood of vulnerability

For each sample $z \in \{1, \ldots, 13\}$ collected for component i in any software version c, we compute a ground truth likelihood $G_{c,i,z}$ for three different scenarios considering the vulnerability level of the set $(\mathcal{I}_{c,i})$ of software components, which is comprised of $S_{c,i}$ and its directly connected neighbours, i.e. $\mathcal{I}_{c,i} = \{i\} \cup \{j : A_c(i,j) = 1\}$.

Accordingly, these three different scenarios are as follows:

1. **Average:** In this instance, we take the ground truth vulnerability level of component i as the average of $S_{c,i}$ and of the corresponding values of the components to which component i is connected:

$$H^1_{c,i,z} = \frac{1}{|\mathcal{I}_{c,i}|} \sum_{j \in \mathcal{I}_{c,i}} V_{c,j,z} \tag{29}$$

2. **Maximum:** As the worst case scenario, we calculate the ground truth for the given component $S_{c,i}$ based on the local vulnerability level of its most vulnerable neighbour (i.e. connected component):

$$H^2_{c,i,z} = \max_{j \in \mathcal{I}_{c,i}} \left(V_{c,j,z} \right) \tag{30}$$

3. **Maximum-Standard Deviation (Max-Std):** We finally consider an intermediate scenario where the standard deviation is subtracted from the maximum vulnerability level among $S_{c,i}$ and its neighbours. In this way, we aim to avoid considering a component equally vulnerable with its most vulnerable neighbour. Then,

$$H^3_{c,i,z} = H^2_{c,i,z} - \sqrt{\frac{1}{|\mathcal{I}_{c,i}|} \sum_{j \in \mathcal{I}_{c,i}} \left(V_{c,j,z} - H^1_{c,i,z} \right)^2} \tag{31}$$

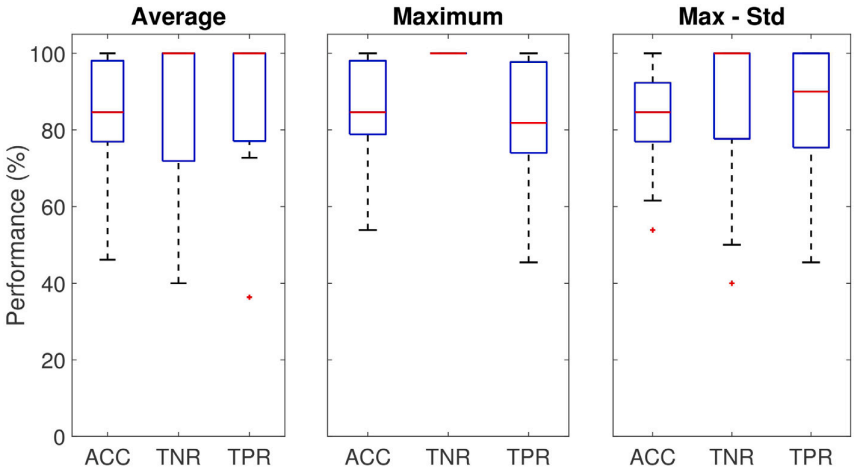

Fig. 4. Performance of the proposed ARNN-based SWVA framework for the latest version (v13) of the GitHubCrawler system using each of the three ground truth calculation methods, namely "Average", "Maximum", and "Maximum-Standard Deviation", with respect to percentage Accuracy, TNR, and TPR metrics.

Then, considering any scenario $m \in \{1, 2, 3\}$, the ground truth of the likelihood ratio is calculated as

$$G_{c,i,z} = \frac{H^m_{c,i,z}}{1 - H^m_{c,i,z}}. \tag{32}$$

5.5. Data collection

The data collection process that was followed for the present study is summarized in the following steps:

- Step 1: Initially, the selected software application, i.e., the GitHub Crawler, was cloned locally along with all its 13 versions, in order to undergo specific processing for generating the required data.
- Step 2: For each version of the selected software application, the adjacency matrix A was extracted. In particular, the Dependency Viewer tool was executed in order to extract the interconnection graph of each version of the selected software application (see Fig. 1). Subsequently, the graphs were manually processed to construct the adjacency matrix (A) for each version, following the process described in Section 5.2.
- Step 3: For each component of each version of the selected software application, the 13 local vulnerability detectors were executed in order to retrieve their vulnerability scores V and v. These scores reflect the local vulnerability level of each component, as provided by the local vulnerability detectors. All 13 versions of the GitHub Crawler were processed, and this process was repeated for each one of the 13 versions of the selected software application, leading to 169 samples.
- Step 4: Using (i) the adjacency matrices A for each version of the selected software application, and (ii) the local vulnerability scores/levels of their software components were computed, we constructed the ground truth, by computing the vulnerability ground truth for the best, worst, and intermediate case scenarios, by taking the average, maximum, and maximum-std vulnerability levels respectively of each component and of its neighbouring components in the adjacency matrix A.
- Step 5: Then, based on the vulnerability level ground truth we computed the vulnerability likelihood ground truth by using (32), for each one of the three considered scenarios. (see Section 5.4). In that way, for each one of the 169 samples we have three ground truth vulnerability likelihood values for each component that correspond to three different scenarios (i.e., best case, worst case, and intermediate case scenarios).
- Step 6: The final dataset was extracted and stored in a form suitable to train and evaluate the ARNN-based SWVA mechanism.

6. Performance of SWVA and its comparison with other methods

We now evaluate the performance of the proposed ARNN-based SWVA framework, using the GitHubCrawler system for each of the three given scenarios, and compare it against the well-known ML models, Multilayer Perceptron (MLP), K-Nearest Neighbours (KNN), and Least Absolute Shrinkage and Selection Operator (Lasso). Since we have 13 samples corresponding to each of the different lexicographic techniques that are used to detect the vulnerability of a software component, for each of the 13 software versions, this leads to a total of 169 samples. Therefore, we perform 5-fold cross-validation to obtain generalizable results using the relatively small dataset. Note that as 5-fold cross-validation trains a given model with 80% of the dataset and tests it with the remaining 20% iteratively changing the training and test data, it provides results that reflect the predicted performance of the model in practice. Also, during our experiments, we trained the ARNN for 100 epochs with learning rate of 10^{-12} for each cross-validation fold, and we used the best value of the decision threshold for each ML model. Note that a relatively small learning rate is selected to prevent large gradient updates, as the learning algorithm minimizes the squared error of the likelihood ratio.

6.1. Performance with different ground truth computations

We first evaluate the performance of the proposed ARNN-based SWVA framework using each of the three ground truth calculation methods given in Section 5.4, for the latest version – v13 – of the GitHubCrawler system. Accordingly, Fig. 4 displays a box-plot of the SWVA performance over software components with respect to percentage Accuracy, True Negative Rate (TNR), and True Positive Rate (TPR) metrics for each of "Average", "Maximum", and "Max-Std" methods.

For each ground truth calculation method, the results in this figure show that ARNN-based SWVA achieves 85% median Accuracy and 100% median TNR. We also see that:

- When **Average** is used, median performance is 100% with respect to both TNR and TPR. However, the lower whisker of the TNR box-plot shows that the average false detection rate for a minority of 25% of components is slightly high, such that the average TNR for 3 of the 11 components is below 85%. At the same time, SWVA successfully identifies vulnerable components achieving above 72% TPR for every component except one.

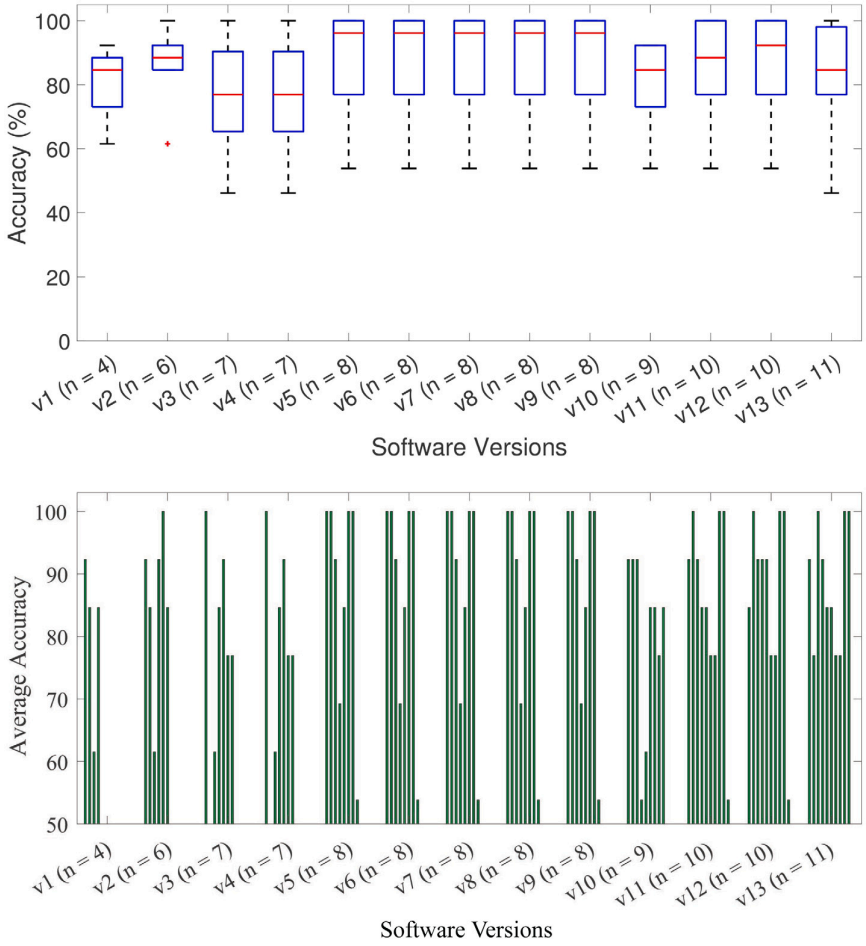

Fig. 5. Percentage accuracy of the ARNN model presented as a box-plot (top) and a bar chart showing the individual components (bottom) for all software versions.

- When **Maximum** is used, although the median of TPR is 82%, the TNR performance of SWVA is 100% i.e. SWVA provides no false detection, for all components except one. This result is expected, as this ground truth only labels each component as invulnerable if all neighbours are also invulnerable. On the other hand, this ground truth calculation significantly decreases median TPR, since our ARNN-based SWVA assesses the vulnerability based on the learned relationship between connected components instead of taking the maximum. That is, ARNN-based SWVA may indicate that a component is not vulnerable even if one of its neighbours is predicted to be locally vulnerable.
- When **Max-Std** is used, we obtain similar results with using Average. In contrast, SWVA now achieves slightly higher TNR for all components with slightly decreased TPR.

In order to avoid biased measurements that could result from taking the maximum vulnerability of connected components as ground truth, during the rest of our analysis we present the results for the **Average** ground truth calculation.

6.2. Detailed results using average-based ground truth

We now analyse the performance of ARNN-based SWVA for different versions of the considered software system by varying the number of components. Thus, in Fig. 5, we present the percentage accuracy of ARNN-based SWVA as a box-plot (top) presenting the performance statistics over all components in each version and as a bar graph (bottom) showing the performance for each component separately.

In Fig. 5, we see that the median accuracy is above 85% for all versions except v3 and v4. We also see that the overall accuracy of ARNN-based SWVA tends to be higher when the total number of components n is comparably larger. This may be due to the fact that ARNN-based SWVA considers interconnected components to make an accurate assessment. In addition, our results in this figure show that although the median accuracy is considerably high, the lower whisker is often between 45% to 60%.

In Fig. 6 (top), which presents the TPR results, we see that the median TPR equals 100% for all versions except v1 for which the median TPR is about 83%. Despite the high median TPR, there are outliers or low whisker values around 45%–50%. During our analysis of the dataset and the results, we observed that the TPR performance is low mainly for the single-connected components, which have a connection with only one of the other components, especially when the ground truth is slightly above 1. The ground truth of the likelihood ratio is observed to be just over 1 (between 1 and 1.5) when ARNN-based SWVA underestimated the likelihood indicating that the given component is invulnerable.

Next, Fig. 6 (bottom) displays the TNR performance of ARNN-based SWVA for each version of the considered software. Our TNR results – similar to the TPR – show that the median performance equals 100% for the majority of versions, and it is between 85% to 90% for only v1, v2 and v10. In addition, for each software version, we see that there are one or two components for which the TNR is less than 60%.

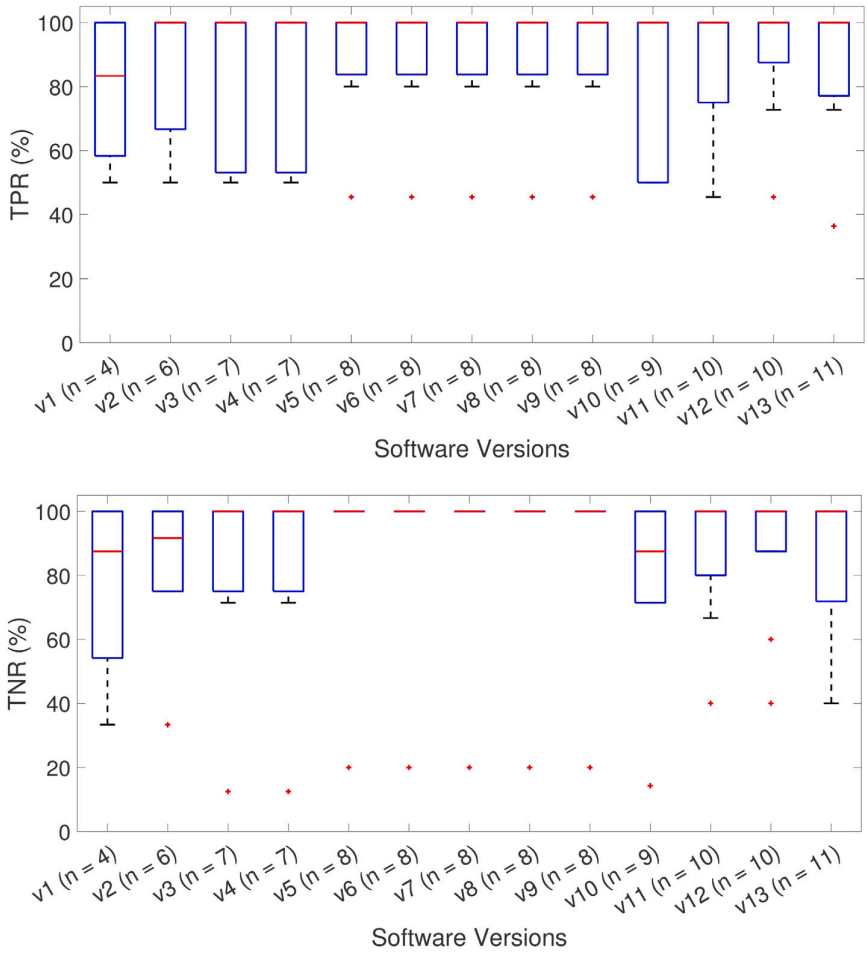

Fig. 6. Box-plot for the percentage TPR (top) and TNR (bottom) performance of the ARNN model for all software versions.

6.3. Comparison with other ML models

We further compare the performance of the ML model, ARNN, adapted specifically for SWVA in this work, with other well-known models, MLP, KNN, and Lasso. As these models are not structured to process local vulnerability predictions (V_i and v_i) separately, the input provided is $L_i^L{}_{i\in\{1,\ldots,n\}}$. That is, the ML models other than ARNN learn a mapping:

$$\textbf{ML}: \{L_i^L\}_{i\in\{1,\ldots,n\}} \to L \tag{33}$$

In addition, these models are implemented as follows:

- **MLP** is a multi-layer perceptron with three hidden layers and sigmoidal activation. Each layer contains n neurons, which, is identical to the number of components in the software system. This architecture does not cause feature shrinkage related to the local vulnerability states of components. Thus, it allows MLP to learn how to assess system-wide vulnerabilities based on local states. Considering the value range of the likelihood ratio, we use the Rectified Linear Unit activation function at each layer of MLP. Moreover, this model is implemented using Keras API in Python and trained via the Adam (Kingma & Ba, 2014) optimizer for 300 epochs to minimize mean squared error.
- **KNN** uses equally weighted 4 neighbours, which are approximately 25% of training samples in each fold of cross-validation and equal to the minimum number of components through all software versions. The KNN model is implemented using scikit-learn library in Python.
- **Lasso** is selected to represent the models with the ability of feature selection and used with the regularization coefficient of 0.1. It is also implemented using the scikit-learn library.

Note that the rest of the parameters of models which are mainly implementation-dependent are used as the default values provided by the libraries.

Fig. 7 displays the performance of ARNN against the MLP, KNN, and Lasso models with respect to median Accuracy, TPR, and TNR respectively from top to bottom. The results in Fig. 7 (top) mainly show that ARNN outperforms other models with 4% to 10% accuracy difference for the majority of the software versions. In more detail, as shown in Fig. 7 (top), ARNN achieves the best accuracy for 8 of 13 versions, while its performance is the same with either Lasso or KNN for the remaining versions v2-4, v10, and v13. That is, ARNN is able to reach the top performance in terms of accuracy for all software versions. The superior performance of ARNN is due to its specific architecture and learning algorithm. The architecture and learning algorithm of ARNN allows it to learn the propagation of vulnerabilities so as to successfully assess the system-wide security. The results in Fig. 7 (bottom left) also shows that ARNN assesses the vulnerabilities significantly better than all of the other models, while the TPR difference between ARNN and the second-best model is about 15% on average. We also see in Fig. 7 (bottom right) that ARNN is able to achieve very high TNR along with its successful assessment of vulnerabilities.

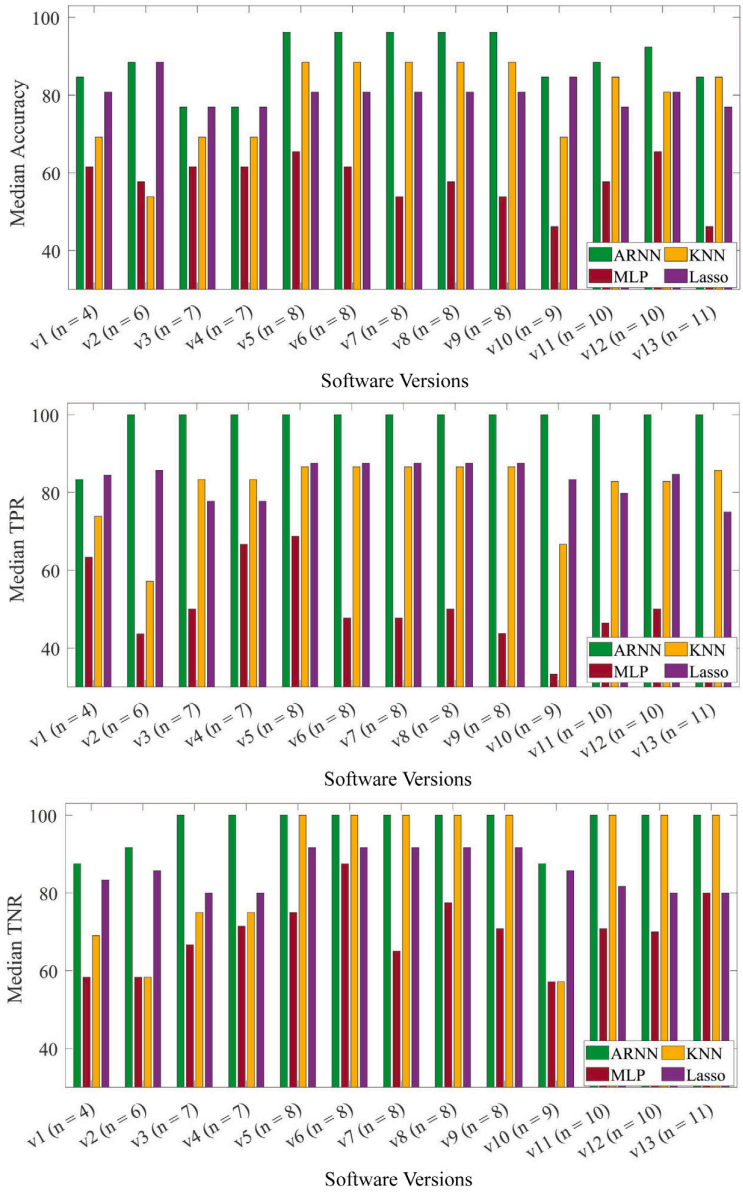

Fig. 7. Performance comparison of different ML models with respect to average Accuracy (top), TPR (bottom left), and TNR (bottom right) for all software versions.

In addition, among the ML models compared, MLP is shown to be the worst-performing model due to the low number of training samples. We see that the TNR of KNN significantly increases with n while that of Lasso and MLP remains almost the same.

In Fig. 8, we present the performance of the compared ML models for each software component in v13. The results in this figure show that ARNN achieves considerably low accuracy only for the "XmlLogger" component. On the other hand, considering all components, ARNN provides consistently high and acceptable accuracy. One should note that although ARNN is a neural network model with a significantly larger set of learnable parameters compared to KNN and Lasso, all ML models are trained only with 10 samples at each fold of the cross-validation.

6.4. Performance evaluation on extended dataset

Furthermore, we evaluate the performance of our SWVA framework on the extended dataset. To this end, for each software version in the original dataset, we first enlarged the data to have 6000 new versions of the software system half having vulnerabilities and half not. Subsequently, we trained the ARNN using 1500 vulnerable and 1500 invulnerable software systems, and we tested it using the remaining 3000 versions of the software system.

For each software version in the original dataset, in order to create 3000 new vulnerable software systems, we inserted a vulnerability code into a randomly selected line of a randomly selected software component. The vulnerability is also randomly chosen among the set of vulnerabilities: {*Heap Overflow*, *CWE-22-CWE-20*, *CWE-209*, *CWE-248*, *CWE-396*, *CWE-772*, *CWE-835*}. Each component to which a vulnerability was introduced was considered to have a local vulnerability level of 1.

In order to create invulnerable software systems, we first manually removed all vulnerabilities in the original system. Then, to create a new invulnerable software system, we removed a random number of bidirectional connections (up to half of all connections), which are randomly

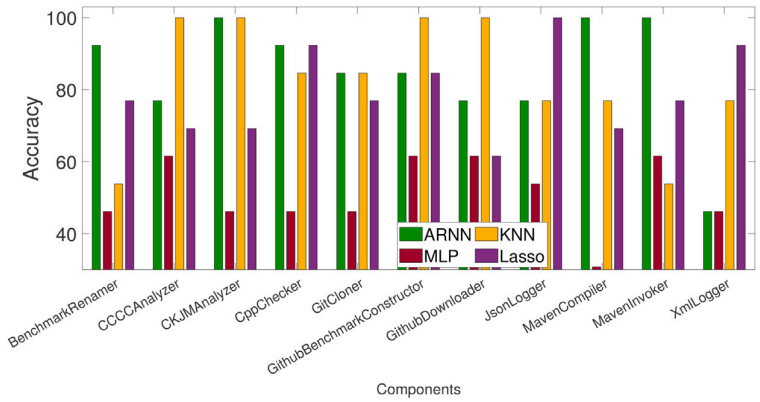

Fig. 8. Performance comparison of different ML models for each component in the latest version (v13) of the considered software with respect percentage accuracy.

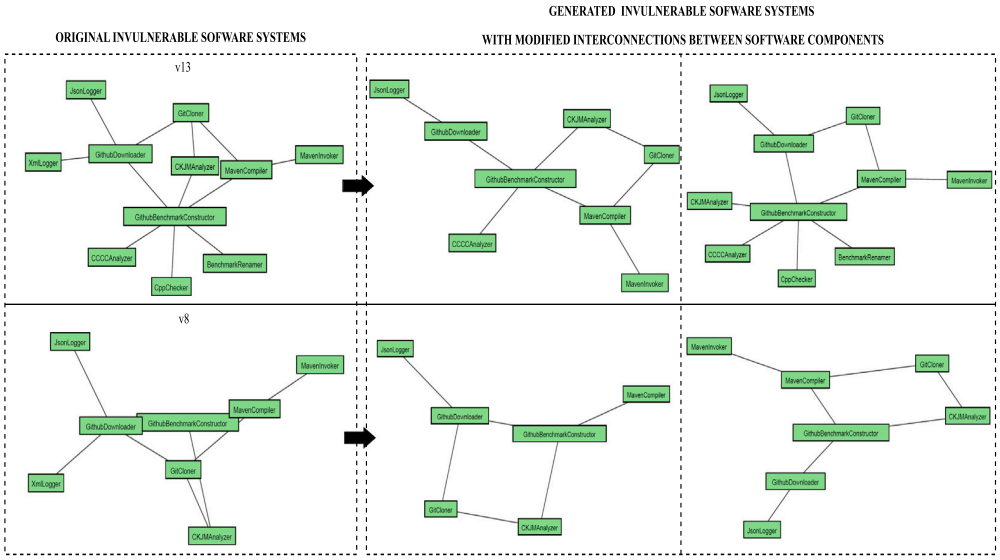

Fig. 9. Examples of generated invulnerable software systems.

selected. Note that the first two software versions are not considered due to the low number of components they contain. Examples of the resulting systems are displayed in Fig. 9.

The average performance of the SWVA framework for assessing the vulnerability of individual components in each software system is given in Fig. 10. The results in this figure show that the ARNN-based SWVA can classify vulnerable and invulnerable components with accuracy varying between 60% to 80%. On the other hand, SWVA classifies vulnerable components with higher accuracy than invulnerable components, indicating slightly high false positive decisions.

We also evaluate the performance of the SWVA for deciding whether the complete software system is vulnerable, where a system is classified as vulnerable if at least one of its components is predicted to be vulnerable by ARNN. That is, a software system is invulnerable only if it does not contain any vulnerabilities. The ground truth for vulnerable systems is calculated in the same way, using the ground truths for component vulnerabilities instead of ARNN predictions.

Fig. 11 displays the performance of the ARNN-based SWVA framework for classifying vulnerable and invulnerable software systems. The results show that the proposed framework can perfectly classify vulnerable software systems. On the other hand, it provides a large number of false decisions for more than half of the software systems. One may say that false positive decisions are less costly than false negative decisions for vulnerability assessment problems.

6.5. Evaluation of the computation time

We finally analyse the computation time of the ARNN model. To this end, Fig. 12 displays the average training time (top) and execution time per sample (bottom) with standard deviation bars for increasing number of software components observed through the system versions. One may say that the computation time is not an important performance indicator for vulnerability assessment as both the training and execution of ARNN for SWVA shall be performed offline. On the other hand, computation time metrics can be indicative of the scalability of ARNN-based SWVA.

In Fig. 12 (top), we see that the training time of ARNN is highly acceptable about 200 seconds for the largest version of the system with $n = 11$. In Fig. 12 (bottom), we see that the mean execution time of ARNN is between 6 to 10 ms, and it increases slightly with n. Note that we implemented ARNN using TensorFlow on Python and measurements are taken using the CPU of a PC with 32 GB of RAM and AMD Ryzen

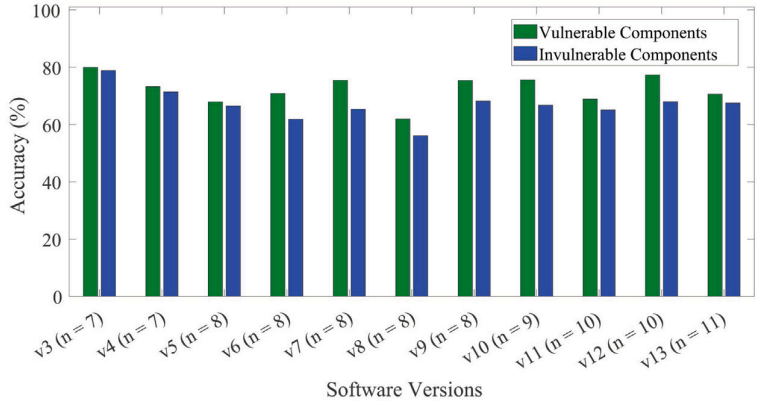

Fig. 10. Accuracy of the ARNN-based SWVA for classifying vulnerable and invulnerable components, corresponding to performance with respect to TPR and TNR.

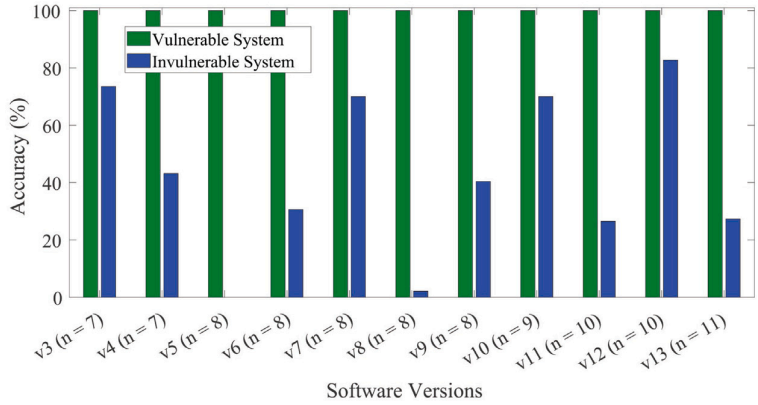

Fig. 11. Accuracy of the ARNN-based SWVA for classifying vulnerable and invulnerable software systems.

7 3.70 GHz processor. In addition, recall that the training of ARNN is performed for 100 epochs with approximately 10 samples at each fold of cross-validation.

6.6. Another example of the use of SWVA

The SWVA method is currently being utilized in practice for providing system-level vulnerability assessment of actual software, and for facilitating the identification and elimination of actual vulnerabilities, in a real-world software application that is operating on a Smart Home environment. The software application called WeatherAPI(i) gathers information from sensors installed outside Smart Home, (ii) to provide weather forecasts based on ML/DL models, and (iii) enable actuators that control the interior temperature of the house based on the forecasts via thermostats and on/off switching of the air-conditioning. WeatherAPI is written in Java by developers who previously used traditional local vulnerability prediction mechanisms for identifying potentially vulnerable components, to prioritize their software testing and improvement. The simple local vulnerability predictors have now been replaced by the SWVA method, which the WeatherAPI developers are actively using as part of their software development pipeline.

In Figure 13, the output of SWVA for a specific commit of the WeatherAPI software application, is illustrated in the form of a graph, whose nodes correspond to the software components of the WeatherAPI application, whereas the vertices denote the interconnections between its components, i.e., the procedure calls. The colour of each node reflects how likely it is for its corresponding class to be vulnerable, as computed by the SWVA mechanism through (32). The darker the shade, the higher the Vulnerability Likelihood of the corresponding component, allowing the developers prioritize their testing and fortification efforts by focusing on high-risk, potentially vulnerable, components. For instance, we see that the WeatherBitDescription component has a much higher likelihood to be vulnerable, and therefore requires further investigation by the development team.

7. Conclusions

In an interconnected system of hardware and software components, when individual components have some vulnerabilities, it is difficult to determine how these local vulnerabilities may propagate across the system and potentially affect other components.

Thus, in this paper, we have developed the System-Wide Vulnerability Assessment (SWVA) framework based on the ARNN model which infers whether the vulnerability of some interconnected components may be affected by the security vulnerabilities of other components.

To this end, the SWVA framework first extracts the interconnections between software components, and the local vulnerabilities are predicted using existing techniques. Then, based on this information, the ARNN – via its problem-specific gradient-based learning algorithm – learns the effect of any vulnerable component on the security of other components connected to it. In this manner, the vulnerability of the given software system as a whole is evaluated, and the effect of local vulnerabilities on the system security.

We have evaluated the performance of the SWVA framework for 13 different versions of the GitHubCrawler, a real-world software system, and compared the results of SWVA with several well-known ML models.

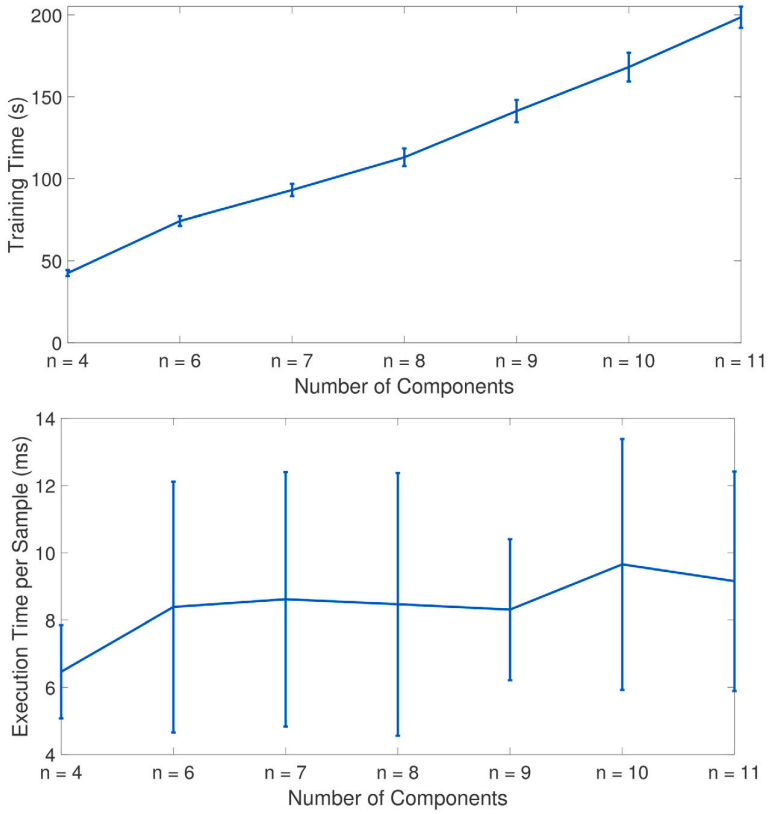

Fig. 12. Training (top) and execution (bottom) times of ARNN for increasing number of components n.

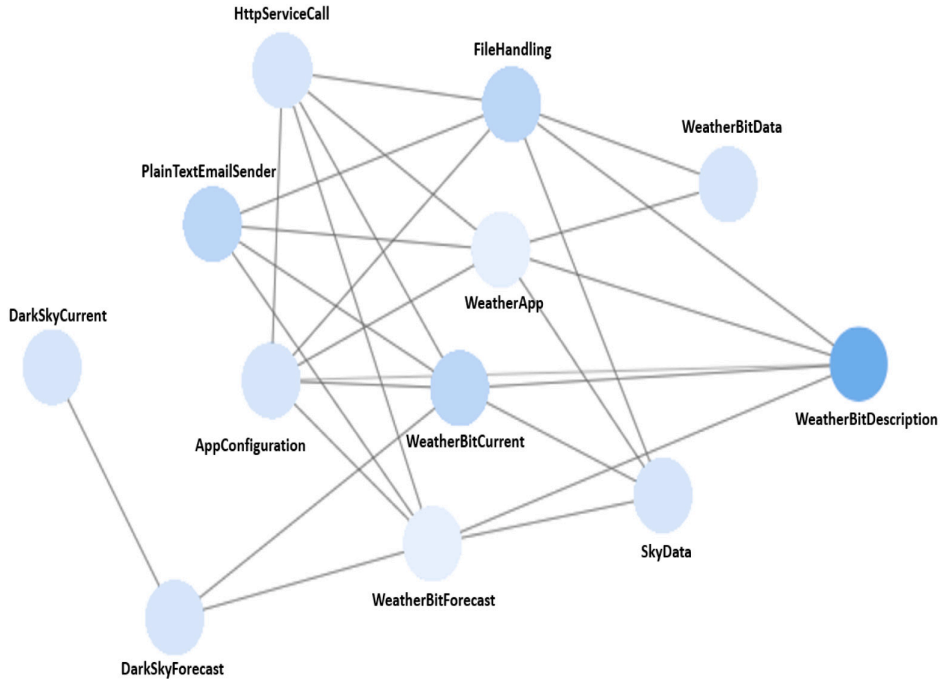

Fig. 13. Graphical output of the SWVA method for the WeatherAPI Java multi-component application that gathers Smart Home sensor data to provide weather forecasts, and control the temperature and air-conditioning actuators of the house. The graph shows the software components and their procedure calls which affect the system vulnerability. A darker colour on a component (nodes), indicates that it is more vulnerable, and therefore that it may require a more careful risk analysis, and possible reprogramming, or that the procedure calls with which it is linked to other components may need to be re-organized. However, we also see that the most "connected" modules are not necessarily those that are the most vulnerable.

Specifically, the evaluation has compared the predictions the ARNN-based SWVA method, applied to the vulnerability prediction of the real-world GitHubCrawler software system, against predictions offered by three well-known ML methods: Multilayer Perceptron (MLP), K-Nearest Neighbours (KNN), and Least Absolute Shrinkage and Selection Operator (Lasso). We have used data from 13 different lexicographic techniques that detect the vulnerability of a software component, for each of the 13 versions of the software, yielding a total of 169 samples. We performed 5-fold cross-validation that trains a given model with 80% of the dataset and tests it with the remaining 20%, iteratively changing the training and test data, to obtain generalizable results using the relatively small dataset. The ARNN was trained for 100 epochs. We also used the best value of the decision threshold for each of the ML models that were compared, and vulnerability prediction was evaluated with respect to percentage Accuracy, True Negative Rate (TNR), and True Positive Rate (TPR).

Our results show that the ARNN-based SWVA successfully assesses the system-wide vulnerability of a multi-component interconnected software system, outperforming the other ML models by achieving over 85% median accuracy. In addition, the ARNN has been shown to have a reasonable computation time — with about 200 s of training time and 9 ms of execution time for an 11-component system. This is relatively low, and it augurs well for the scalability of this novel ARNN based SWVA framework.

In future work, we will extend the SWVA framework to interconnected systems whose number and connections change dynamically over time, so that the structure of the ARNN, namely the neurons and connection weights, will evolve and learn for different successive versions of a software system.

CRediT authorship contribution statement

Erol Gelenbe: Conceptualisation, Investigation, Methodology, Formal Analysis, Algorithm Design, Writing – original draft, Review & writing successive revisions, Project administration, Funding acquisition. **Mert Nakıp:** Investigation, Writing – original draft, Algorithm Implementation, Data Preparation, Performance Measurements. **Miltiadis Siavvas:** Investigation, Writing – original draft, Software Development, Data Extraction, Local Vulnerability Assessent.

Data availability

Data will be made available on request.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cie.2024.110453.

References

Adeel, A., Larijani, H., & Ahmadinia, A. (2017). Random neural network based cognitive engines for adaptive modulation and coding in LTE downlink systems. *Computers & Electrical Engineering*, *57*, 336–350. http://dx.doi.org/10.1016/j.compeleceng.2016.11.005.

Ahmad, J., Larijani, H., Emmanuel, R., Mannion, M., Javed, A., & Phillipson, M. (2017). Energy demand prediction through novel random neural network predictor for large non-domestic buildings. In *2017 annual IEEE international systems conference, SysCon 2017, Montreal, QC, Canada, April 24-27, 2017* (pp. 1–6). IEEE, http://dx.doi.org/10.1109/SYSCON.2017.7934803.

Ahmad, J., Tahir, A., Larijani, H., Ahmed, F., Shah, S. A., Hall, A. J., et al. (2020). Energy demand forecasting of buildings using random neural networks. *Journal of Intelligent & Fuzzy Systems*, *38*(4), 4753–4765. http://dx.doi.org/10.3233/JIFS-191458.

Aiello, G., Gaglio, S., Lo-Re, G., Storniolo, P., & Urso, A. (2005). The random neural network model for the on-line multicast problem. In B. Apolloni, M. Marinaro, & T. R. (Eds.), *Biological and artificial intelligence environments* (pp. 157–164). Springer, Dordrecht, http://dx.doi.org/10.1007/1-4020-3432-6_19.

Basterrech, S., Mohamed, S., Rubino, G., & Soliman, M. A. (2011). Levenberg-Marquardt training algorithms for random neural networks. *Computer Journal*, *54*(1), 125–135. http://dx.doi.org/10.1093/comjnl/bxp101.

Catal, C., Akbulut, A., Ekenoglu, E., & Alemdaroglu, M. (2017). Development of a software vulnerability prediction web service based on artificial neural networks. In *Pacific-Asia conference on knowledge discovery and data mining* (pp. 59–67). Springer.

cert (2020). Computer emergency response team coordination center. [online] Available: https://www.kb.cert.org/vuls/ (Accessed: 05 August 2020).

Chowdhury, I., & Zulkernine, M. (2011). Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, *57*(3), 294–313.

cisco (2019). Cisco 2019 Annual Report. [online] Available: https://www.cisco.com/c/dam/en_us/about/annual-report/cisco-annual-report-2019.pdf. (Accessed: 05 August 2020).

ciscopriv (2019). Cisco Cybersecurity Series 2019. Consumer Privacy Survey. [online] Available: https://www.cisco.com/c/dam/en_us/about/annual-report/cisco-annual-report-2019.pdf. (Accessed: 05 August 2020).

Dam, H. K., Tran, T., Pham, T., Ng, S. W., Grundy, J., & Ghose, A. (2017). Automatic feature learning for vulnerability prediction. arXiv preprint arXiv:1708.02368.

Dam, H. K., Tran, T., Pham, T., Ng, S. W., Grundy, J., & Ghose, A. (2018). Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, *47*(1), 67–85.

DepTool (2024). Dependability Toolbox. URL https://gitlab.seis.iti.gr/sdk4ed-wiki/wiki-home/wikis/dependability-toolbox.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

Ding, Y., Duan, R., Li, L., Cheng, Y., Zhang, Y., Chen, T., et al. (2017). Poster: Rust SGX SDK: Towards memory safety in intel SGX enclave. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security* (pp. 2491–2493).

EU HORIZON Project SDK4ED. (2020). URL https://sdk4ed.eu/.

Evmorfos, S., Vlachodimitropoulos, G., Bakalos, N., & Gelenbe, E. (2020). Neural network architectures for the detection of SYN flood attacks in IoT systems. In *Proceedings of the 13th ACM international conference on pErvasive technologies related to assistive environments* (no. 69), (pp. 1–4). ACM; DOI: 10.1145/3389189.3398000.

Filus, K., Boryszko, P., Domańska, J., Siavvas, M., & Gelenbe, E. (2021). Efficient feature selection for static analysis vulnerability prediction. *Sensors*, *21*(4), 1133.

Filus, K., Siavvas, M., Domańska, J., & Gelenbe, E. (2020). The random neural network as a bonding model for software vulnerability prediction. In *Symposium on modelling, analysis, and simulation of computer and telecommunication systems* (pp. 102–116). Springer.

Fu, M., & Tantithamthavorn, C. (2022). LineVul: A transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th international conference on mining software repositories* (pp. 608–620). http://dx.doi.org/10.1145/3524842.3528452.

Gelenbe, E. (1989). Random neural networks with negative and positive signals and product form solution. *Neural Computation*, *1*(4), 502–510.

Gelenbe, E. (1993). Learning in the recurrent random neural network. *Neural Computation*, *5*, 154–164.

Gelenbe, E., & Nakip, M. (2023). IoT network cybersecurity assessment with the associated random neural network. *IEEE Access*, *11*, 85501–85512. http://dx.doi.org/10.1109/ACCESS.2023.3297977.

Gelenbe, E., & Yin, Y. (2017). Deep learning with dense random neural networks. In *International conference on man–machine interactions* (pp. 3–18). Springer, Cham.

Ghalut, T., & Larijani, H. (2014). Non-intrusive method for video quality prediction over LTE using random neural networks (RNN). In *9th international symposium on communication systems, networks & digital signal processing* (pp. 519–524). IEEE, http://dx.doi.org/10.1109/CSNDSP.2014.6923884.

Ghalut, T., & Larijani, H. (2018). Content-aware and QOE optimization of video stream scheduling over LTE networks using genetic algorithms and random neural networks. *Journal of Ubiquitous Systems and Pervasive Networks*, *9*(2), 21–33. http://dx.doi.org/10.5383/JUSPN.09.02.003.

Hanif, H., & Maffeis, S. (2022). Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 international joint conference on neural networks* (pp. 1–8). IEEE.

Hovsepyan, A., Scandariato, R., Joosen, W., & Walden, J. (2012). Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on security measurements and metrics*.

Hussain, K. F., & Moussa, G. S. (2016). On road vehicle classification based on random neural network and bag of visual words. *Probability in the Engineering and Informational Sciences*, *30*, 403–412. http://dx.doi.org/10.1017/S0269964816000073.

Jackson, K. A., & Bennett, B. T. (2018). Locating SQL injection vulnerabilities in Java byte code using natural language techniques. In *SoutheastCon 2018* (pp. 1–5).

Javed, A., Larijani, H., Ahmadinia, A., Emmanuel, R., Mannion, M., & Gibson, D. (2017). Design and implementation of a cloud enabled random neural network-based decentralized smart controller with intelligent sensor nodes for HVAC. *IEEE Internet of Things Journal*, *4*(2), 393–403. http://dx.doi.org/10.1109/JIOT.2016.2627403.

Javed, A., Larijani, H., Ahmadinia, A., & Gibson, D. (2017). Smart random neural network controller for HVAC using cloud computing technology. *IEEE Transactions on Industrial Informatics*, *13*(1), 351–360. http://dx.doi.org/10.1109/TII.2016.2597746.

Kalouptsoglou, I., Siavvas, M., Kehagias, D., Chatzigeorgiou, A., & Ampatzoglou, A. (2022). Examining the capacity of text mining and software metrics in vulnerability prediction. *Entropy*, *24*(5), http://dx.doi.org/10.3390/e24050651, URL https://www.mdpi.com/1099-4300/24/5/651.

Kehagias, D., Jankovic, M., Siavvas, M., & Gelenbe, E. (2021). Investigating the interaction between energy consumption, quality of service, reliability, security, and maintainability of computer systems and networks. *SN Computer Science*, *2*(1), 1–6.

Kim, S., Choi, J., Ahmed, M. E., Nepal, S., & Kim, H. (2022). VulDeBERT: A vulnerability detection system using BERT. In *2022 IEEE international symposium on software reliability engineering workshops* (pp. 69–74). http://dx.doi.org/10.1109/ISSREW55968.2022.00042.

Kingma, D. P., & Ba, J. (2014). ADAM a method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Latif, S., Huma, Z., Jamal, S. S., Ahmed, F., Ahmad, J., Zahid, A., et al. (2022). Intrusion detection framework for the Internet of Things using a dense random neural network. *IEEE Transactions on Industrial Informatics*, *18*(9), 6435–6444. http://dx.doi.org/10.1109/TII.2021.3130248.

Latif, S., Idrees, Z., Zou, Z., & Ahmad, J. (2020). DRaNN: A deep random neural network model for intrusion detection in industrial IoT. In *2020 international conference on UK-China emerging technologies* (pp. 1–4). http://dx.doi.org/10.1109/UCET51115.2020.9205361.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., et al. (2018). Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681.

Martínez, M., Moròn, A., Robledo, F., Rodríguez-Bocca, P., Cancela, H., & Rubino, G. (2008). A GRASP algorithm using RNN for solving dynamics in a P2P live video streaming network. In *2008 eighth international conference on hybrid intelligent systems, Barcelona* (pp. 447–452). http://dx.doi.org/10.1109/HIS.2008.23.

Maven (2024). MAVEN. URL https://maven.apache.org/.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Moshtari, S., & Sami, A. (2016). Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction. In *Proceedings of the 31st annual ACM symposium on applied computing* (pp. 1415–1421).

Nafi, K. W., Roy, B., Roy, C. K., & Schneider, K. A. (2020). A universal cross language software similarity detector for open source software categorization. *Journal of Systems and Software*, *162*, Article 110491.

Neuhaus, S., Zimmermann, T., Holler, C., & Zeller, A. (2007). Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on computer and communications security* (pp. 529–540).

owasp (2020). Open Web Application Security Project (OWASP). [online] Available: https://owasp.org/ (Accessed: 05 August 2020).

owaspguide (2020). OWASP secure coding practices quick reference guide. [online] Available: https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v1.pdf. (Accessed: 05 August 2020).

Pang, Y., Xue, X., & Wang, H. (2017). Predicting vulnerable software components through deep neural network. In *Proceedings of the 2017 international conference on deep learning technologies* (pp. 6–10).

Radhakrishnan, K., & Larijani, H. (2011). Evaluating perceived voice quality on packet networks using different random neural network architectures. *Performance Evaluation*, *68*(4), 347–360. http://dx.doi.org/10.1016/j.peva.2011.01.001.

Rubino, G., Tirilly, P., & Varela, M. (2006). Evaluating users' satisfaction in packet networks using random neural networks. In S. D. Kollias, A. Stafylopatis, W. Duch, & E. Oja (Eds.), *Lecture notes in computer science*: *vol. 4131*, *Artificial neural networks - ICANN 2006, 16th international conference, Athens, Greece, September 10-14, 2006. Proceedings, part I* (pp. 303–312). Springer, http://dx.doi.org/10.1007/11840817_32.

Salka, C. (2005). Programming languages and systems security. *IEEE Security & Privacy*, *3*(3), 80–83.

sans (2020). Information security training - SANS cyber security certifications & research. [online] Available: https://www.sans.org/ (Accessed: 05 August 2020).

Shin, Y., Meneely, A., Williams, L., & Osborne, J. A. (2010). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, *37*(6), 772–787.

Shin, Y., & Williams, L. (2008a). 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* (pp. 315–317).

Shin, Y., & Williams, L. (2008b). Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on quality of protection* (pp. 47–50).

Siavvas, M., Gelenbe, E., Kehagias, D., & Tzovaras, D. (2018). Static analysis-based approaches for secure software development. In *International ISCIS security workshop* (pp. 142–157).

Siavvas, M., Kehagias, D., Tzovaras, D., & Gelenbe, E. (2021). A hierarchical model for quantifying software security based on static analysis alerts and software metrics. *Software Quality Journal*, *29*(2), 431–507.

Siavvas, M., et al. (2024). SDK4ED: A platform for building energy efficient, dependable, and maintainable embedded software. *Auomated Software Engineering*, *31*(53), 52–123. http://dx.doi.org/10.1007/s10515-024-00450-z.

sonarqube (2024). SonarQube. [online] Available: https://www.sonarqube.org/. (Accessed: 03 August 2020).

Timotheou, S. (2009). A novel weight initialization method for the random neural network. *Neurocomputing*, *73*(1), 160–168.

Timotheou, S. (2010). The random neural network: A survey. *The Computer Journal*, *53*(3), 251–267.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., et al. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998–6008).

Veracode (2018). *State of software security vol. 9*: *Tech. rep.*.

verastat (2020). Veracode. [online] Available: https://www.veracode.com/. (Accessed: 05 August 2020).

VIEWER. (2024). URL https://marketplace.eclipse.org/content/java-dependency-viewer.

Walden, J., Stuckman, J., & Scandariato, R. (2014). Predicting vulnerable components: Software metrics vs text mining. In *2014 IEEE 25th international symposium on software reliability engineering* (pp. 23–33).

Yin, Y. (2018). Deep learning with the random neural network and its applications. arXiv, arXiv:1810.08653.

Zagane, M., Abdi, M. K., & Alenezi, M. (2020). Deep learning for software vulnerabilities detection using code metrics. *IEEE Access*, *8*.

Zhang, M., de Carnavalet, X. d., Wang, L., & Ragab, A. (2019). Large-scale empirical study of important features indicative of discovered vulnerabilities to assess application security. *IEEE Transactions on Information Forensics and Security*, *14*(9), 2315–2330.

Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. arXiv preprint arXiv:1909.03496.