Optimising vulnerability triage in DAST with deep learning

Millar, S., Podgurskii, D., Kuykendall, D., Martinez-del-Rincon, J., & Miller, P. (2022). Optimising vulnerability triage in DAST with deep learning. In *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security* (pp. 137-147). (AISec: Artificial Intelligence and Security Proceedings). Association for Computing Machinery. https://doi.org/10.1145/3560830.3563724

Published in:
Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
Copyright 2022 Association for Computing Machinery.
This work is made available online in accordance with the publisher's policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Open Access
This research has been made openly available by Queen's academics and its Open Research team. We would love to hear how access to this research benefits you. – Share your feedback with us: http://go.qub.ac.uk/oa-feedback

Download date:18. Mar. 2025

Optimising Vulnerability Triage in DAST with Deep Learning

Anonymous
anon@anon.com
Anonymous

Anonymous
anon@anon.com
Anonymous

Anonymous
anon@anon.com
Anonymous

ABSTRACT

Managing false positives generated by vulnerability scanners is a fundamental industry-wide challenge in web application security. Accordingly, this paper presents a novel multi-view deep learning architecture to optimise Dynamic Application Security Testing (DAST) vulnerability triage. Deliberate, task-specific design decisions are taken to exploit the structure of traffic exchanges between our rules-based DAST scanner and a given web app. By leveraging multiple convolutional neural networks, natural language processing and word embeddings, our model learns separate yet complementary internal feature representations of these exchanges before fusing them together to make a prediction of a verified vulnerability or a false positive. Given the amount of time and cognitive effort required to constantly manually review high volumes of DAST results correctly, the addition of this deep learning capability to a rules-based scanner creates a hybrid system that enables expert analysts to rank scan results, deprioritise false positives and concentrate on likely real vulnerabilities. This improves productivity and reduces remediation time, resulting in stronger security postures. Evaluations are conducted on a real-world proprietary dataset containing 91,324 findings of 74 different vulnerability types curated from DAST scans on nineteen individual organisations. Results show our multi-view architecture significantly reduces both the false positive rate by 20% and the false negative rate by 40% on average across all organisations compared to the single-view approach.

CCS CONCEPTS

• **Security and privacy → Web application security**; • **Computing methodologies → Neural networks**; *Natural language processing*; *Supervised learning*.

KEYWORDS

web application security, convolutional neural networks, deep learning, DAST

ACM Reference Format:
Anonymous, Anonymous, and Anonymous. 2022. Optimising Vulnerability Triage in DAST with Deep Learning. In *Proceedings of ACM CCS (AISec '22).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
AISec '22, Nov 11, 2022, Los Angeles, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Cyber-crime has been predicted to cause yearly financial losses globally of US$6trillion [2], with the average organisational cost of an attack via a vulnerable web application recently calculated at US$2.2million [1]. Low adoption of more secure technologies adds to the web application security issue - based on 206 commercial pen-testing engagements in the period June 2019 to June 2020, 48% of enterprise web apps were written in ASP.NET, 22% in Java and 14% in PHP, languages that are less type-safe than the likes of Rust, Kotlin or TypeScript [37]. The ubiquitous nature of these technologies and the rise of the cloud mean improving web application security is crucial, with [44] reporting that web application attacks continue to be the most common type of compromise and are a preferred vector for attackers.

Web application Dynamic Application Security Testing (DAST) scanners help organisations detect vulnerabilities both early in the development lifecycle and in production. DAST tools scan web apps in real-time, and their improvements have led to more frequent vulnerability discovery. However, the industry is aware that scanning for vulnerabilities is prone to significant levels of false positives. Results often need manually inspected by a security analyst to determine which are real vulnerabilities, and which correspond to false positives. Known as triaging, this process is both time-consuming and costly, potentially causing alert fatigue, burnout and even mental health issues in overwhelmed front-line analysts [7, 39]. It is clear that research opportunities exist for new techniques to augment DAST tools and optimise vulnerability triage.

Artificial intelligence (AI) and machine learning (ML) have been proposed as a way to address this problem [39], for example to identify these false positives and filter them out of a triage workflow. Furthermore, ML models that automatically learn and encapsulate domain knowledge can reduce the labour-intensive maintenance of manually engineered, hard-coded banks of detection rules. These time savings are central to reducing exposure and improving security postures. Considering the average time to detect a web breach is 200 days [33], the sooner a vulnerability can be discovered, verified and remediated, the smaller the window of opportunity for an attacker to successfully carry out an exploit. Vulnerability triage tasks may seem a logical choice for automation, however [1] also showed that only 38% of their enterprise respondents deploy defensive AI/ML systems in an operational setting. A key aspect in this lack of uptake is whether system performance meets a certain standard, with the limiting factor commonly being an acceptable false positive rate [5]. Management of false positives is considered one of most important problems in cybersecurity, alongside detecting the security concerns themselves [5, 7, 32, 36, 41].

Additionally, there is a need for research-based web application security tools to maintain a level of abstraction and perform well across a variety of attacks without an obligatory requirement for original source code to be exposed. Whilst useful in advancing

the field, not all attack-specific research and methods that rely on source code analysis can transfer to a real-world environment in practice. Security-sensitive customers require protection from a wide range of web attacks and with each customer having tens or hundreds of applications each, it cannot be presumed 3rd party tools will be granted such privileged low-level access. Moreover, academia and the cybersecurity industry have operated in separate silos in the past [39], with academia developing state-of-the-art solutions on older data that may not be representative of what is seen in practice, and conversely industry has a vast amount of high-quality data but their approaches do not effectively harness its unique characteristics.

We address these shortcomings with a multi-view deep learning model using convolutional neural networks, natural language processing (NLP) and word embeddings to explicitly exploit the structure of client-server pen-test web traffic captured by our DAST scanner. Neither the model nor the scanner require access to source code, with this proposed hybrid system in the first instance finding potential vulnerabilities using the existing rules-based scan engine, followed by the deep learning model predicting whether these findings are real vulnerabilities or false positives. This enables augmentation of decisions made by expert analysts to optimise productivity and reduce remediation times by deprioritising false positives. After evaluation on a large proprietary dataset, results show our multi-view architecture produces best-in-task performance compared to the single-view approach. Our contributions are:

- A multi-view deep learning model to enhance a rules-based DAST scanner, creating a hybrid system for optimised, decision-augmented triage that improves analyst productivity and reduces remediation time.
- A novel task-specific architecture leveraging multiple convolutional neural networks with NLP to exploit the request and response structure of client-server web traffic.
- An evaluation on real-world proprietary data comprising 91,324 findings of 74 different vulnerability types from nineteen organisations, showing a significant 20% reduction in false positives and 40% reduction in false negatives on average across all customers compared to a single-view approach.

This paper is organised as follows: Section 2 discusses related work, Section 3 explains the methodology and Section 4 outlines the experimental setup. Detailed results are in Section 5 with Section 6 containing conclusions and future work.

2 RELATED WORK

2.1 Web Application Vulnerability Detection

Domain knowledge for web application security is considered expensive, and research has sometimes been conflated with areas such as detecting network intrusions, ex-filtration, attack observation or malware identification. For clarity, the focus of web application security is specifically the detection of exploitable vulnerabilities in a web app, like SQL injection or cross-site scripting (XSS) amongst many others. Previously published work regularly focuses on a single attack or a small group of attacks due to the difficulty in obtaining high quality and high volume datasets, with a common approach being development of one-off bespoke systems or undertaking white-box code analysis. For example, [9] explored the susceptibility of web apps to cookie-hijacking attacks via an automated black-box technique, detecting authentication and authorisation flaws stemming from incorrect handling or protection of cookies. [53] introduced differential traffic analysis to identify server-side vulnerable access control implementations in online services, and [52] tested network traffic for vulnerabilities in the Facebook SSO web app. [48] hand-engineered static string analysis to detect XSS vulnerabilities by checking HTML generated from layout engines for poor input validation, while [25] detected SQL injection vulnerabilities by hand-crafting malicious SQL queries based on their original form, then compared benign and attack SQL parse trees for differences. [12] proposed a static framework identifying SQL injection vulnerabilities at compile time using string analysis plus bytecode inspection via symbolic execution. [19] also focused on SQL injection using static string analysis of a code base, with a dynamic phase that checked run-time queries against that statically generated model. The drawback with white-box methods is the assumption that source code for the web apps under scrutiny is available for analysis, however in practice it may not be feasible for customers to provide this unfettered low-level access due to infosec policies or wider security concerns more generally.

Using ML to tackle the problem of detecting vulnerabilities in web applications has been under-researched. [42] detected SQL injection and XSS with random forests using an agent-like instance to extract web app execution features, with [40] proposing logistic regression and forests to detect SQL injection, XSS, remote code execution and file inclusion from hand-crafted attributes for input validation and sanitisation. Furthermore, deep learning approaches have only recently started to be considered. [34], an extension of [42], presented a stacked denoising autoencoder [45] that learnt from agent-generated application call graphs, and [38] adopted a character-level convolutional neural network for detecting code injection. This prior work made valuable theoretical contributions though was limited by the amount of real-world data available for evaluation, often focusing on only a handful of vulnerability types.

Lastly, methods for web application vulnerability detection are usually of a singular nature, where only one source of information from a set of rules or modules are considered. The absence of hybrid systems that specifically augment a rules-based DAST engine with ML to enhance vulnerability triage creates a research challenge that we focus on in this work.

2.2 Deep Learning for Cybersecurity

Progress has been made in applying deep learning to other selected cybersecurity use-cases, for example detecting malicious powershell [20], image steganalysis [51], biometric fingerprint matching [26], memory forensics [30], malware detection [11, 29], DDoS detection [8], presentation attack detection [13] and checking source code for defects [21]. [31] also presented a model to generate developer comments for source code, plus deep learning has been used for code vulnerability analysis [47], code summarisation [4] and code review [17]. The consistent trend is that the approach specifically fits the task and the dataset under investigation. This sits in contrast to wider research in deep learning where, broadly

speaking, methods of increasing complexity are tested across the same benchmark datasets and tasks from paper to paper. For our study we take advantage of a large proprietary dataset to train and evaluate with confidence, and furthermore our proposed architecture has been expressly designed to model and exploit the structure of that data.

3 METHODOLOGY

DAST tools actively investigate running internal and external web applications using automated penetration tests, known as scans, to detect vulnerabilities an attacker may try to exploit. Our multi-view deep learning model augments our existing DAST scanner to form a proposed hybrid system to optimise vulnerability triage, shown in Figure 1. Neither the model nor the scanner require any access to the application's original source code. For the avoidance of doubt, the deep learning model is not a replacement for the DAST scanner, and has no involvement in executing any attacks in a scan.

This hybrid approach can be considered to have two phases. Firstly, our rules-based DAST engine scans a web app or set of web apps and generates a list of potential vulnerabilities, also known as findings. Findings are persisted in a database as individual JSON files that contain both benign traffic and attack traffic exchanged between the scanner and the web app. Secondly, prior to the triage step, our proposed model uses this exchange traffic, represented as NLP word embeddings, to predict whether each finding is a verified vulnerability or a false positive. The predictions are also persisted in the database. The analyst then triages a set of findings, typically upwards of several thousand, via a User Interface (UI). Presented alongside the findings data in the UI is our deep learning model's prediction for each finding, either a real vulnerability, or a false positive. The findings list can then be ranked by the analyst, allowing predicted real vulnerabilities to be dealt with first while false positives are deprioritised, thus significantly optimising the triage process.

Previous deep learning research for web vulnerability detection has been limited due to the large amounts of data required to train complex architectures. In contrast, our model is evaluated on an extensive dataset and specifically takes advantage of DAST traffic. This carefully considered design process, explained in Section 3.2, is necessary as simply replicating a previous setup from one domain does not guarantee performance in another. Achieving high performance is logically more likely with architectures that exploit dataset structure, a view shared by [21], who noted prior approaches to finding bugs in code ignored program structure, with [3] surmising document classification models may not efficiently incorporate document structure in their architectures.

3.1 DAST Pre-processing and Input Feature Generation

In its original form, raw traffic generated by our DAST scanner is stored in JSON files as text strings, persisting the client-server request and response exchange. During the pen-test a web app is scanned for a multitude of vulnerabilities using a wide range of attack modules, and a potential vulnerability is considered a finding. There is one JSON file per finding containing the client-server traffic between the scanner and the web app under scrutiny, with JSON objects for the original request, original response, attack request and attack response. Original traffic exchanges are benign, with attack exchanges containing some type of payload or pen-test intent, and there is one vulnerability type per finding, such as SQL injection or XSS. This traffic is converted into input for our model using NLP techniques and pre-trained word embeddings.

Word embeddings are regularly used in pre-processing for task-specific deep learning, with the most common being word2vec [17, 18, 24, 27, 28, 43, 50], GloVe [18, 23, 27, 35, 49, 50] or FastText [3, 6]. As fixed length vectors of floating point numbers, word embeddings enable models to recognise how similar two words are, without losing the ability to encode each one distinctly. The words, also known as tokens, are viewed as points, or embeddings, in a multi-dimensional space. Words that appear in similar contexts are close to each other [14], and a pre-trained word embedding model always returns the same fixed length vector for a given token. With the term *word embedding* relating to one single token, we call the representation of a whole finding a *finding embedding*. Per Algorithm 1, each finding in the dataset is pre-processed to create a single-view finding embedding, E, and a corresponding set of four multi-view finding embeddings. Each element in the multi-view set $\{E_{or1}, E_{or2}, E_{at1}, E_{at2}\}$ represents the original request, original response, attack request and attack response respectively.

To compare the single-view and multi-view approaches, this input data must be structured appropriately. Figure 2 illustrates the difference between single-view and multi-view embeddings, with these matrices being input to the deep learning models. The single-view model is forced to learn across all original and attack traffic combined, whereas the multi-view model has the opportunity in training to adapt each part of its architecture to learn specific

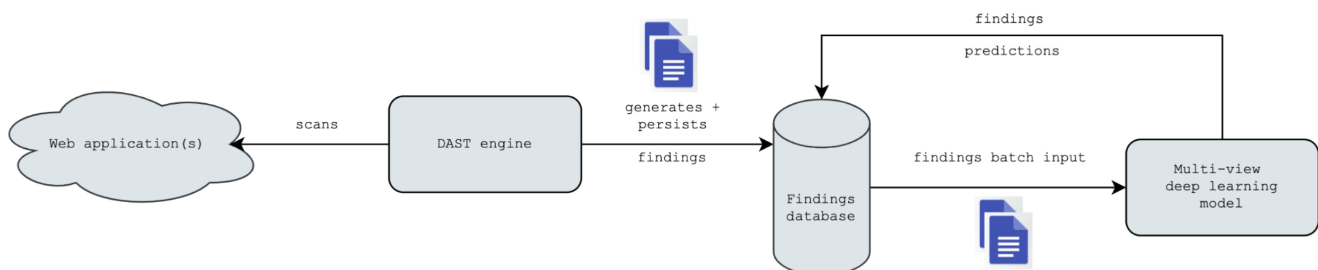
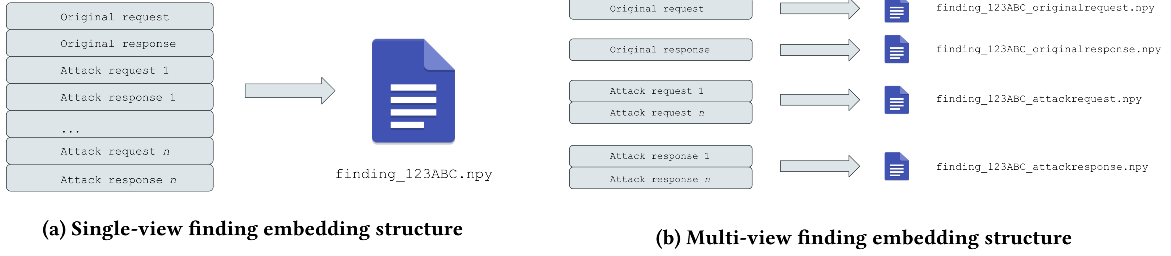


Figure 1: Hybrid DAST and deep learning system for web vulnerability triage



(a) Single-view finding embedding structure

(b) Multi-view finding embedding structure

Figure 2: Finding embedding structures

Algorithm 1 Generating a finding embedding E.

Require: Raw traffic to pre-process t, pre-trained embedding model *embeddingModel*, numpy library *np*.

1: **procedure** CREATE EMBEDDING(t)
2: $t \leftarrow lower(t)$ ▷ Convert traffic to lowercase.
3: $t \leftarrow whitespace(t)$ ▷ Remove dupe whitespaces.
4: $t \leftarrow specialchars(t)$ ▷ Replace non-alphabetical chars.
5: $t \leftarrow whitespace(t)$ ▷ Remove dupe whitespaces again.
6: $t \leftarrow tokenise(t)$ ▷ Split t into tokens.
7: $E \leftarrow embeddingModel(t)$ ▷ Make embedding matrix E.
8: $E.npy \leftarrow np.save(E)$ ▷ Save E to disk as numpy array.
9: **end procedure**

characteristics of original request, original response, attack request and attack response traffic.

The challenge of using pre-trained, general NLP word embeddings in domain-specific settings is that relevant domain-specific tokens may be lost since they are not part of an embedding model's original vocabulary. These tokens, such as dynamically generated strings or hashes, are considered out-of-vocab (OOV). Therefore each E, returned from a pre-trained word embedding model after passing a traffic token stream t, is a matrix composed of embeddings for only in-vocab tokens, stacked together to preserve the same order as the original sequence in t. In Section 5, experiments are presented using different embeddings and also mechanisms to include OOV tokens.

3.2 Multi-view Deep Learning Architecture

This paper presents a multi-view deep learning model that explicitly exploits the structure of client-server request and response traffic between our DAST scanner and a web app. We wish to study to what degree taking advantage of the exchange structure of DAST data affects performance, rather than combining all the traffic into one input source for a single-view model. Per Section 2.2, with previous work showing a task-specific approach is more suitable in a specialist domain such as cybersecurity, we carefully consider this structure of our data when designing the model. The resulting architecture enables more discriminative learning compared to vanilla single-view baselines that do not have any task-specific considerations. Given a finding comprises an original request with an original response, plus an attack request with an attack response, the core idea is a multi-view model with four inputs can learn separate yet complementary feature representations from each request and response before they are merged to take the final classification.

When the four internal representations are fused together for vulnerability prediction, the decision quality is higher as the features are more salient compared to a single-view model.

3.2.1 *Original and Attack Exchange CNNs.*

CNNs are a class of neural network that combines convolutional operations with the automatic learning of model parameters using deep learning for feature extraction and classification. They have applications in image and video analysis, natural language processing, speech recognition and financial time-series tasks. A detailed explanation of CNNs can be found in [15]. The multi-view architecture comprises four CNNs in parallel, one for each part of the original and attack exchanges. Each single-layer CNN learns discriminative areas of its respective input that, when combined, enable effective prediction of a verified vulnerability or a false positive. As explained in the previous subsection, the set of four input finding embeddings are $\{E_{or1}, E_{or2}, E_{at1}, E_{at2}\}$. Each E has size $l \times k$ where l is the number of in-vocab traffic tokens and k is the embedding dimension. Relationships are learnt across the sequence of traffic tokens in each E via a set of f filters in its corresponding CNN.

Per Figure 3, each filter has size $s \times k$ and slides over a given E with a stride of 1, with s being the length of the filter i.e. the number of consecutive tokens that are considered together when learning relevant patterns across the sequence. The CNN executes 1D convolutions for each filter, producing an activation map a of size $m = l - s + 1$ as follows:

$$a_f = ReLU(Conv(E)_{W_f b_f}) \qquad (1)$$

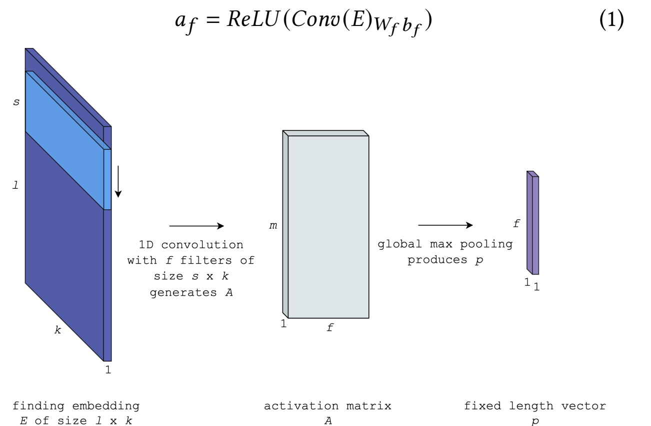


Figure 3: An individual CNN within the multi-view architecture

with W_f and b_f being the weight and bias parameters of the fth filter, learnt whilst training the model. It is important to note they are not shared between views, allowing each CNN to focus exclusively on only one part of an exchange. The rectified linear activation function is used where $ReLU(x) = max\{0, x\}$. Each activation map is stacked forming an activation matrix A of size $m \times f$. A can be written as:

$$A = [a_1|a_2|...|a_f] \tag{2}$$

We use global max pooling to select the maximum activation in each a, generating a fixed length vector p of length f:

$$p = [max(a_1)|max(a_2)|...|max(a_f)] \tag{3}$$

In essence, this pooling layer chooses the parts of each input E that activate each filter in the corresponding CNN the most, and thus can be considered more influential in a prediction. Since p is fixed length, pooling also ensures any E of arbitrary size can be handled. With this CNN architecture present in each of the four views, the four separate fixed length vectors p_{or1}, p_{or2}, p_{at1} and p_{at2} are therefore produced. To combine the four streams of features into one informative representation, these vectors are concatenated as p_{multi}, formally:

$$p_{multi} = [p_{or1}|p_{or2}|p_{at1}|p_{at2}] \tag{4}$$

Dropout occurs at a rate of d, after which p_{multi} is forwarded for classification.

3.2.2 Classifier. The classifier is a multi-layer perceptron comprised of an input layer matching the size of p_{multi} i.e. $f \times 4$, a hidden layer of size h neurons, and an output layer of $o = 2$ neurons since our task has two classes, with a finding predicted as either a verified vulnerability or a false positive. The output layer generates a vector z containing two elements with each being a score that a given finding belongs to the respective class. z is formally:

$$z = ReLU(W_h p_{multi} + b_h) \tag{5}$$

where W_h and b_h are the weight and bias parameters of the hidden layer learnt during the training phase. z then passes through a SoftMax layer to convert these scores to normalised probabilities of that finding belonging to each class. Formally:

$$p(y = i|z) = \frac{\exp(w_i^{\intercal} z + b_i)}{\sum_{i}^{I} \exp(w_i^{\intercal} z + b_i)} \tag{6}$$

where w_i and b_i are the weight and bias parameters of the SoftMax layer learnt during the training phase for each of the i classes, with $w_i^{\intercal} z$ being the inner product of w_i and z. The predicted label, either a verified vulnerability or a false positive, is y.

3.2.3 Loss Function. Lastly, the loss function C to be minimized for a batch of B training findings, $\{I_{(1)}, ..., I_{(B)}\}$, where each $I = \{E_{or1}, E_{or2}, E_{at1}, E_{at2}\}$, can be denoted as:

$$C = -\frac{1}{B} \sum_{j=1}^{B} \sum_{i=1}^{c} 1\{y'_{(j)} = i\} \log p(y_{(j)} = i|z_{(j)}) \tag{7}$$

where there are $c = 2$ classes in our task, $y'_{(j)}$ is the ground truth label for training finding $I_{(j)}$ and $z_{(j)}$ is the resulting output after the forward pass of $I_{(j)}$ through the model. During the training phase, the model is repeatedly presented with batches of training data in random order until its parameters converge, thus minimising C. Figure 4 shows the overall multi-view architecture described throughout this subsection.

4 EXPERIMENTAL SETUP

There are three stages to the experiments. First, extensive hyperparameter tuning takes place using a single-view CNN plus various single-view recurrent architectures, namely the recurrent neural network (RNN), the long short-term memory network (LSTM), and its bidirectional variant (BiLSTM). This allows selection of an architecture as the backbone for our multi-view approach. Recurrent architectures are a type of neural network that exhibit temporal dynamic behaviour, achieved by creating loops and connections between nodes, allowing information to be stored over time within

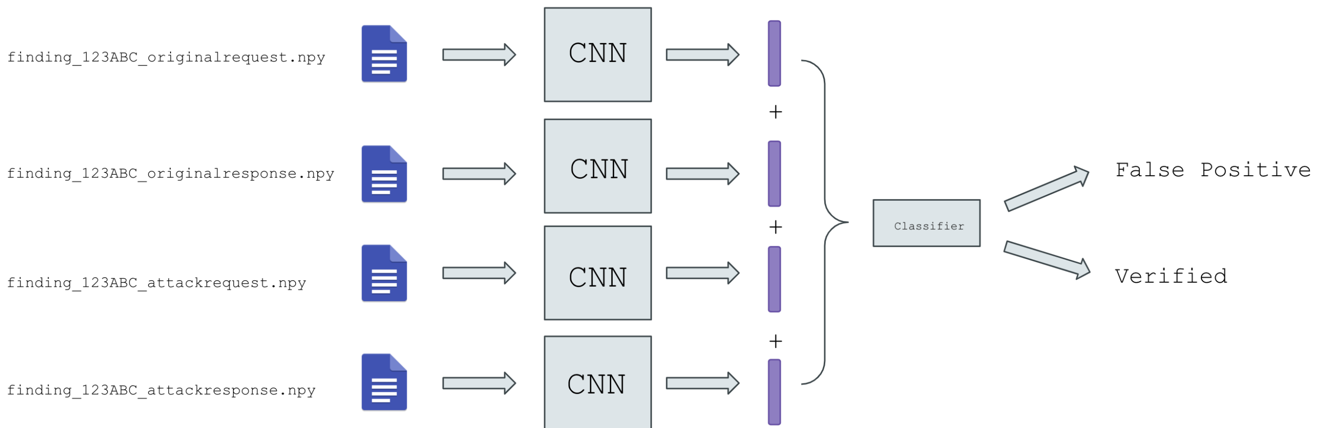


Figure 4: Multi-view deep learning architecture for web vulnerability detection

Table 1: Vulnerabilities dataset overview

Vulnerability type	# verified	# FP	Vulnerability type	# verified	# FP
Anonymous Access	12	13	Information Disclosure in response	90	136
Apache Struts Detection	0	1	Information Disclosure in scripts	7	146
ASP.NET Misconfiguration	267	33	Information Leakage in responses	2852	341
ASP.NET ViewState security	11	45	JavaScript Memory Leaks	8	29
Autocomplete attribute	396	183	LDAP Injection	1	12
Blind LDAP Injection	0	6	Local Storage Usage	4	0
Blind NoSQLi	2	9813	NoSQLi Injection	202	856
Blind SQL	68	1373	OS Commanding	2	2
Browser Cache directive (leaking sensitive information)	10769	733	Out of Band Cross-site scripting (XSS)	0	1
Browser Cache directive (web application performance)	87	144	Parameter Fuzzing	609	1421
Brute Force (Form Auth)	1	3	Persistent Cross-site scripting (XSS), (active)	6	7
Brute Force (HTTP Auth)	0	27	Predictable Resource Location	148	13641
Business logic abuse attacks	29	536	Privacy Disclosure	4	117
Clients Cross-Domain Policy Files	27	27	Privilege Escalation	68	105
Collecting Sensitive Personal Information	0	81	Profanity	0	1
Content Security Policy Header	99	986	Reflected Cross-site scripting (XSS)	5466	393
Cookie attributes	2463	167	Reflected Cross-site scripting (XSS), (simple)	524	13
Credentials over an insecure channel	28	79	Reflection	13	155
Credentials stored in clear text in a cookie.	28	0	Sensitive Data Exposure	1968	1844
Cross Origin Resources Sharing (CORS)	569	14	Sensitive data over an insecure channel	155	67
Cross-Site Request Forgery (CSRF)	358	2807	Server Configuration	6	1
Cross-site scripting (XSS), (DOM based reflected via AJAX request)	1	0	Server Side Request Forgery	0	7
Cross-site scripting (XSS), (DOM based)	0	7	Session Fixation	23	294
Cross-site tracing (XST)	2	0	Session Strength	7	31
Directory Indexing	161	0	Session Upgrade	42	107
Email Disclosure	0	2	SQL Information Leakage	165	118
File Inclusion	17	135	SQL Injection	42	18
Forced Browsing	197	571	SQL Injection Auth Bypass	0	3
Form Session Strength	0	1	SQL Parameter Check	1	0
FrontPage Checks	116	307	Subresource Integrity	781	10
HTTP Authentication over insecure channel	428	0	Unvalidated Redirect	18	8
HTTP Headers	759	328	Web Beacon	1	0
HTTP Response Splitting	6	0	X-Content-Type-Options	173	40
HTTP Strict Transport Security	147	81	X-Frame-Options	137	244
HTTP User-Agent Check	76	28	X-Powered-By	34	29
HTTPS Downgrade	800	6	X-XSS-Protection	61	31
HTTPS Everywhere	21599	440	XPath Injection	0	2

the network using internal state [16]. These models are chosen because they are regularly presented in previously published research on a variety of NLP tasks. Next, as word embeddings have been also been used in previous work to generate raw input representations, a detailed ablation experiment is conducted using word2vec [28], GloVe [35] and FastText [6] to select the best pre-trained embedding model, and whether to use word embeddings or character embeddings. After the hyperparameter and word embedding investigations, we progress to evaluating our multi-view architecture with this selected backbone, conducting a set of experiments using real-world customer data. A series of customer-specific models are trained, using only one customer's data in training and testing each model against the unseen data from that same customer. This is a realistic scenario whereby risk management infosec policies dictate customer vulnerability data cannot be aggregated nor transferred out of a geographical region for training.

4.1 Dataset

The proprietary dataset contains 91,324 findings, with each already triaged and labelled by expert analysts as either a verified vulnerability or a false positive. These findings were originally generated by our DAST tool during scanning activities for nineteen different customer organisations, which are not named for confidentiality reasons. Table I gives a breakdown of the 74 types of web vulnerabilities in the dataset, and Table II shows the number of verified and false positive findings per customer. The customers are from a wide variety of industries such as finance, education, healthcare and energy, who engage our managed services division to scan their web applications and triage the findings. A finding can be defined initially as an unreviewed potential vulnerability found in a web app after a given attack launched by the DAST scanner is finished, which then goes on to be triaged and labelled by an expert analyst as either verified or false positive. If an attack uncovers no potential vulnerability, no finding is generated. The data for each of the nineteen customer organisations is individually split into an 80% training split, a 10% validation split, and a 10% testing split to use in our experiments.

Table 2: Per customer dataset overview

Customer	# verified	# FP	Customer	# verified	# FP
1	844	723	11	1003	460
2	2728	878	12	683	316
3	3002	1723	13	419	1079
4	8005	2383	14	350	2671
5	5137	2540	15	1133	805
6	5754	4436	16	1066	1628
7	8446	4218	17	461	1275
8	3342	4771	18	1157	1809
9	5010	4972	19 (internal)	3898	974
10	469	756			

4.2 Experimental Details

Given the setup in Figure 4, the proposed model must effectively discriminate between real verified vulnerabilities and false positives so the latter can be deprioritised to optimise an expert analyst's triage and cost-effectiveness. For our deep learning model, the positive class is considered to be a verified vulnerability, and the negative class a false positive finding. In ML, a false positive occurs when a sample of the negative class is falsely predicted to be positive - for us, this is a false positive finding mistakenly classified as a real vulnerability. In practice however it is not realistic to eradicate false

positives completely. Rather, the aim should be to minimise the false positive rate whilst monitoring the miss rate, with a miss (or false negative) being a sample of the positive class being incorrectly predicted as negative - that is, a verified vulnerability mistakenly classified by the model as a false positive finding. Thus, we measure accuracy, precision, recall, F1 score, false positive rate and miss rate, formally:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

$$precision = \frac{TP}{TP + FP} \tag{9}$$

$$recall = \frac{TP}{TP + FN} \tag{10}$$

$$F1 = \frac{2 \times precision \times recall}{precision + recall} \tag{11}$$

$$false\ positive\ rate\ = \frac{FP}{FP + TN} \tag{12}$$

$$miss\ rate = \frac{FN}{FN + TP} \tag{13}$$

where TP, FP, TN, FN are the amount of true positives, false positives, true negatives and false negatives respectively, with the false positive rate and miss rate expressed as a percentage in our results.

For all experiments, the following training parameters are used: $B = 1$ to ensure maximum updates to the model parameters via stochastic gradient descent, with backpropagation using the gradient of C with respect to these parameters. The updates are weighted to handle any class imbalance, with the learning rate $\alpha = 0.001$ used in this parameter update process and dropout $d = 0.5$.

5 RESULTS

5.1 Hyperparameter Tuning Study

Deep learning architectures require hyperparameter optimisation, however the samples from a dataset used for the training and validation stages must not also be included in the test phase, otherwise performance can be artificially high. Therefore for tuning, a set of internal findings are used where we have scanned a selection of our own web applications, listed as customer 19 in Table II. This a sensible choice because it completely avoids the involvement of any real-world customer data that is intended for later evaluation, and reduces overfitting.

Taking the training and validation splits of these internal findings, a study is performed on convolutional and recurrent architectures to select hyperparameters that optimise performance on the validation split. Per Section 3.1 and Algorithm 1, the finding embeddings are generated using the standard pre-trained word2vec embedding model as it has been widely used previously. In the next subsection, word2vec, GloVe and FastText are compared in detail.

A comprehensive grid search is used, rather than individual hyperparameter ablations, as the grid search covers every possible permutation. When training a model, validation-based early stopping [46] saves the best version of that model to disk, based on the highest F1 score on the internal findings validation set. This helps avoid overfitting to the training data where it is learnt too well and validation performance peaks then starts to decline. The training and testing hardware used was a GeFORCE GTX 1080 Ti GPU, 64MB RAM and an Intel i7-8700 CPU.

For our single-view CNN approach, the grid search parameters are:

- number of filters, $f = 2 - 256$ in powers of 2
- filter length, $s = 2 - 256$ in powers of 2
- filter width is fixed at $k = 300$
- classifier hidden layer size, $h = 4 - 512$ in powers of 2
- number of epochs, $e = 10, 20, 30$

The single-view recurrent models tuned are a vanilla RNN, an LSTM, and a BiLSTM. Each has a single-layer and are trained for 50 epochs as recurrent architectures require more training compared to convolutional methods [16]. For each recurrent model, the grid search parameters are:

- hidden dimension, $d = 64 - 512$ in powers of 2
- input size is fixed at $k = 300$
- classifier hidden layer size, $h = 4 - 512$ in powers of 2
- number of epochs, $e = 50$

Results in Table III show the best tuning settings from these exhaustive grid searches for each of the architectures, including three different single-view CNNs with equivalent F1 scores and some variation in the FP and miss % rates. The speed of training and testing is also important, with t_{train} denoting the total time taken to train an architecture in seconds and t_{test} denoting the time taken to test a single finding. In a laboratory setting, training time can be absorbed offline, allowing a sole focus on the time taken to predict a given data point, but in practice this is not always the case. For example, with our task the training of any eventual solution must take place online in the cloud to preserve the security of sensitive vulnerability data, hence the training phase has a cost implication. It can be seen the single-view CNNs performs better across all metrics than the single-view recurrent architectures, including an F1 score of 0.96 compared to only 0.85 for the LSTM and BiLSTM. The recurrent architectures clearly take vastly longer to train and test also. Given these results, the single-view RNN, LSTM and BiLSTM are ruled out at this stage. The three single-view CNN settings have similar performance in the first instance, and so are carried forward into the ablation study that follows on word embeddings.

Table 3: Hyperparameter tuning

Architecture	f	s	d	k	h	# params	e	t_{train}	t_{test}	Acc.	Prec.	Recall	F1	FP %	Miss %
Single-view CNN	128	8	-	300	64	315714	10	33.05	0.0027	0.96	0.92	1.00	0.96	7.69	0.00
Single-view CNN	128	8	-	300	64	315714	30	99.23	0.0027	0.96	0.95	0.97	0.96	3.08	3.08
Single-view CNN	32	32	-	300	4	307374	30	133.05	0.0038	0.96	0.94	0.98	0.96	6.15	1.54
RNN	-	-	256	300	4	143886	50	3,493.67	0.0235	0.82	0.82	0.82	0.82	18.46	18.46
LSTM	-	-	128	300	32	224354	50	3,217.70	0.0163	0.83	0.92	0.78	0.85	7.69	26.15
BiLSTM	-	-	64	300	4	187918	50	6,632.43	0.0505	0.83	0.94	0.77	0.85	6.15	27.69

5.2 Word Embedding Ablation Study

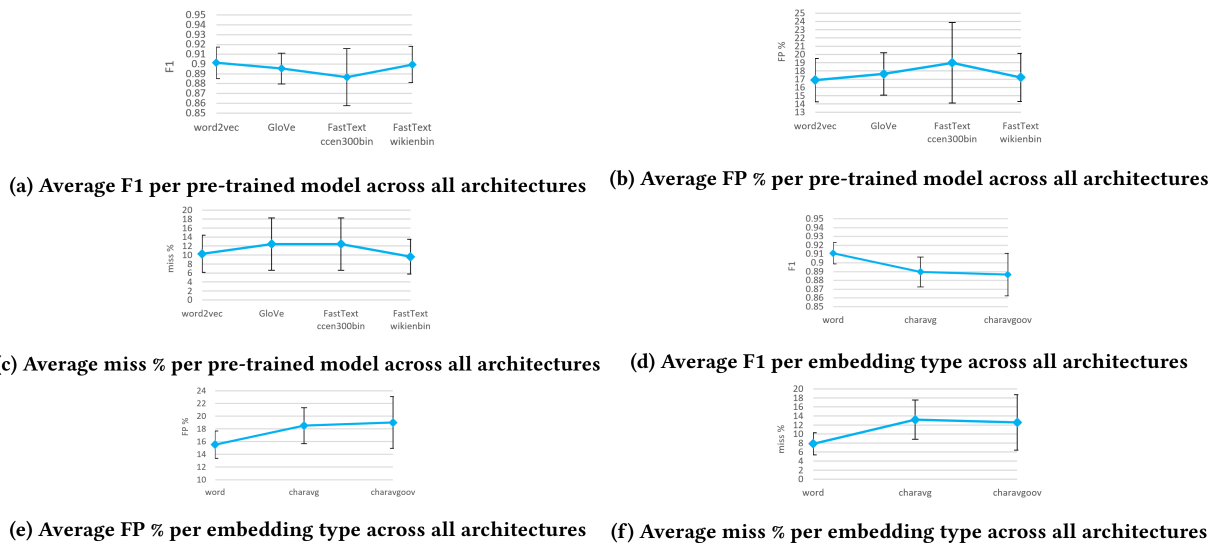
The three single-view CNN settings from Table III are used to decide which pre-trained embedding model to adopt and whether to select standard word embeddings or character embeddings. The OOV problem was discussed in Section 3.1, with tokens not in the original vocab of a pre-trained embedding model discarded. Leveraging character embeddings to create a word embedding for OOV tokens may help solve this issue. Using the internal findings, while keeping the splits exactly the same, finding embeddings are generated using word2vec [28], GloVe [35] and FastText [6], with each permutation in Table IV tested to see which performs best. Note the only difference between the FastText ccen300bin and wikienbin models is their original training corpus. For each of the three single-view CNN settings, and using each of the embeddings in Table IV, models are trained and validated with the same internal findings training split and validation split, and then tested with the internal test split. Per Figure 5, the test split metrics are averaged across all the pre-trained embedding models to enable selection of the best model, and also averaged across all the embedding types, to decide whether to use standard word or character embeddings.

Results in Figures 5a, 5b and 5c show the average F1, FP % rate and miss % rate per pre-trained embedding model across all architectures. As a brief reminder, a false positive occurs when the negative class is falsely predicted to be positive i.e. a false positive finding mistakenly classified as a real vulnerability, and conversely a miss (or false negative) occurs when the positive class is incorrectly predicted as negative i.e. a verified vulnerability mistakenly classified as a false positive finding. Given word2vec achieves a high F1 score, low FP % rate and low miss % compared to GloVe and both FastText models, it is selected as the pre-trained embedding model to use for real-world customer data evaluations. FastText wikienbin performs comparably to word2vec but is not selected as the extra expense in producing an embedding vector for every token does not result in a worthwhile performance gain, indicating a degree of redundancy. With word2vec, 24% of internal findings tokens are OOV and discarded. Using FastText wikienbin to process these missing OOV tokens increases computation time by 48% compared to word2vec and so can be avoided since the expense offers little improvement. Additionally, the embeddings must be stored in our cloud environment at the scale of potentially thousands of customers, thus reductions in run-time and disk space with no discernible drop in classifier performance are favourable.

Regarding the choice of word or character embeddings, results in Figures 5d, 5e and 5f show the standard word embedding technique is best with a higher F1 score, lower FP % rate and lower miss % rate. We assert using character embeddings to deal with OOV tokens

Table 4: Embedding permutations

Pre-trained model	Type	Description
word2vec	standard	Standard word2vec.
word2vec	charavg	For any word in-vocab, each word embedding is the average of the character embeddings for that word.
word2vec	charavgoov	For any word in-vocab or OOV, each word embedding is the average of the character embeddings for that word.
GloVe	standard	Standard GloVe.
GloVe	charavg	For any word in-vocab, each word embedding is the average of the character embeddings for that word.
GloVe	charavgoov	For any word in-vocab or OOV, each word embedding is the average of the character embeddings for that word.
FastText ccen300bin	standard	Standard FastText using the ccen300 corpus. For OOV words, FastText creates an embedding by using in-vocab subwords.
FastText ccen300bin	charavg	For any word in-vocab, each word embedding is the average of the character embeddings for that word.
FastText ccen300bin	charavgoov	For any word in-vocab or OOV, each word embedding is the average of the character embeddings for that word.
FastText wikienbin	standard	Standard FastText using the wikien corpus. For OOV words, FastText creates an embedding by using in-vocab subwords.
FastText wikienbin	charavg	For any word in-vocab, each word embedding is the average of the character embeddings for that word.
FastText wikienbin	charavgoov	For any word in-vocab or OOV, each word embedding is the average of the character embeddings for that word.



(a) Average F1 per pre-trained model across all architectures

(b) Average FP % per pre-trained model across all architectures

(c) Average miss % per pre-trained model across all architectures

(d) Average F1 per embedding type across all architectures

(e) Average FP % per embedding type across all architectures

(f) Average miss % per embedding type across all architectures

Figure 5: Average performance metrics for word embedding ablation study

provides no significant advantage, in fact they hamper performance, again implying redundancy as dealing with OOV tokens means including everything, for example all random payloads and dynamic strings. In turn the finding embeddings are noisier and hence model performance is reduced.

In summary, the pre-trained word2vec embedding model with standard word embeddings is selected and held constant for remaining experiments. From Table III the single-view CNN setting of $f = 128$, $s = 8$, $h = 64$ and $e = 10$ results in the best F1 score and the best balance between FP rate, 7.69%, and miss rate, 0%. Whilst these results are only for the internal dataset they are extremely encouraging. In practice, the deep learning model has managed to correctly discard approximately 92% of the false positives, whilst not missing any of the verified vulnerabilities. Thus with our aim to achieve a miss rate of 0% whilst significantly reducing the number of false positives, this architecture is also fixed. We now move to evaluations using real-world customer data to compare our single-view and multi-view approaches.

5.3 Real-world Customer Data Evaluations

In this section we use our architectures to generate customer-specific models. This is a realistic production scenario as corporate infosec policies may dictate vulnerability data must not be aggregated across customers, nor transferred from one geographical region to another. There is potential for personal data to be contained within vulnerability data, and such preventative safeguarding policies reduce exposure to data protection laws, for example in the United States, where legal frameworks vary between federal states and industries, and in Europe [10, 22]. These experiments directly compare our single-view CNN and multi-view CNN by training a series of models with a sole customer's data, and evaluating them against their own test split. Any performance gain via the multi-view architecture versus the single-view can then be studied.

5.3.1 Training and testing customer-specific models. Here a single-view CNN and a multi-view CNN are trained separately for each customer using their training and validation split, and tested with that same customer's test split. Validation-based early stopping is again used, per Section 5.1. This produces a set of 18 single-view models, and another set of 18 multi-view models, one per customer.

With the original findings used to generate the individual training, validation and test splits frozen and held constant in each split, any change in performance can be attributed to the differences between the single-view and multi-view architecture.

Results in Table V show that on average across all customer-specific models, the multi-view architecture outperforms single-view. In practice, 96% of false positives are successfully removed using multi-view, whilst only missing 3% of the verified vulnerabilities. Multi-view also produces better FP % rates and miss % rates in the case of nearly every customer. In absolute terms, the multi-view average FP rate of 4% is 1% less than single-view, and the multi-view average miss rate of 3% is 2% less than single-view. Considering the volume of findings in production, small changes in the FP rate and miss rate have a big impact, thus effectively our multi-view architecture offers a worthwhile reduction of 20% in false positives and 40% in misses compared to single-view. Multi-view also has a better average F1 score of 0.96 in comparison to the single-view average of 0.95. Regarding computation time, with the previous single-view training and testing times of 99.23s and 0.0027s from Table 3 for comparison, the training and testing times for the multi-view model using the same data are 181.58s and 0.0056s. These multi-view times remain minimal and inexpensive, even with the increase versus single-view, which can be expected given the higher architectural complexity.

Overall, we conclude the demonstrated significant reduction of false positives is due to our multi-view architecture specifically exploiting the structure of client-server DAST data, learning from each of the views corresponding to the original request, original response, attack request and attack response traffic. This multi-view architecture preserves the underlying structure of the raw data from the original client-server exchanges, compared to the single-view architecture that disregards this structure. Note no extra data is required for the multi-view model, per Figure 2, since it uses the same data fed to the single-view model, with the internal architecture the only difference. Multi-view is further justified by virtue of enabling both better separation and learning of features specific to original and attack request/response traffic which are subsequently fused together internally for improved performance.

Table 5: Customer-specific single-view CNN vs. multi-view CNN architecture comparison

Customer	Architecture	Acc.	Prec	Recall	F1	FP %	Miss %	Customer	Architecture	Acc.	Prec	Recall	F1	FP %	Miss %
1	Single-view	0.98	0.98	0.98	0.98	3	2	11	Single-view	0.91	0.94	0.93	0.94	13	7
1	Multi-view	0.98	1.00	0.97	0.98	0	3	11	Multi-view	0.98	0.97	1.00	0.99	7	0
2	Single-view	0.97	0.96	1.00	0.98	12	0	12	Single-view	0.95	0.94	1.00	0.97	15	0
2	Multi-view	0.97	0.97	0.99	0.98	9	1	12	Multi-view	0.97	0.96	1.00	0.98	9	0
3	Single-view	0.98	0.98	0.98	0.98	3	2	13	Single-view	0.97	0.97	0.91	0.94	1	9
3	Multi-view	0.98	0.97	0.99	0.98	5	1	13	Multi-view	0.98	1.00	0.93	0.96	0	7
4	Single-view	0.99	0.99	0.99	0.99	3	1	14	Single-view	0.98	0.94	0.91	0.93	1	9
4	Multi-view	0.99	0.99	1.00	0.99	3	0	14	Multi-view	0.99	0.92	1.00	0.96	1	0
5	Single-view	0.98	0.98	0.99	0.98	4	1	15	Single-view	0.96	0.96	0.97	0.97	6	3
5	Multi-view	0.98	0.98	0.99	0.99	4	1	15	Multi-view	0.98	0.97	0.99	0.98	4	1
6	Single-view	0.96	0.98	0.94	0.96	2	6	16	Single-view	0.89	0.96	0.74	0.84	2	26
6	Multi-view	0.96	0.98	0.95	0.97	2	5	16	Multi-view	0.87	0.81	0.89	0.85	14	11
7	Single-view	0.98	0.97	0.99	0.98	6	1	17	Single-view	0.98	0.96	0.98	0.97	2	2
7	Multi-view	0.99	1.00	1.00	1.00	1	0	17	Multi-view	0.98	0.96	0.96	0.96	2	4
8	Single-view	0.99	0.99	1.00	0.99	1	0	18	Single-view	0.82	0.77	0.79	0.78	15	21
8	Multi-view	0.99	0.98	1.00	0.99	1	0	18	Multi-view	0.82	0.76	0.84	0.79	17	16
9	Single-view	0.98	0.98	0.98	0.98	2	2	Average	Single-view	0.96	0.96	0.95	0.95	5	5
9	Multi-view	1.00	0.99	1.00	1.00	1	0	**Average**	**Multi-view**	**0.97**	0.96	**0.97**	**0.96**	**4**	**3**
10	Single-view	0.98	1.00	0.94	0.97	0	6								
10	Multi-view	0.98	1.00	0.96	0.98	0	4								

6 CONCLUSIONS AND FUTURE WORK

This paper presents a novel hybrid model for web application vulnerability triage that augments a rules-based DAST engine with a multi-view deep learning architecture. This architecture explicitly exploits the structure of our proprietary DAST dataset through deliberate design decisions, demonstrating deep learning can optimise vulnerability triage. This occurs via the learning of separate representations of client-server original request, original response, attack request and attack response DAST traffic. Real-world evaluations show our multi-view approach improves performance over a single-view baseline, with the opening ablation experiments indicating that, for our task, convolutional architectures outperform recurrent ones, and word2vec embeddings are effective in pre-processing.

There are opportunities for future work related to data curation and model architecture. As domain adaptation is a known challenge in deep learning, if time can be invested to expand the dataset sufficiently, perhaps a strategy of architecting and training a larger range of attack-specific rather than customer-specific models could be of benefit, with each targeting a single vulnerability type regardless of the customer domain. This may enable optimised triage for both existing customers and brand new customers who have no historical data. The feasibility of this will depend on infosec policies surrounding aggregation and transfer of data between geographical regions.

Further, there is potential for web application technology to change, with new web servers, firewalls, protocols and of course vulnerabilities, however a given model need not be static - with sufficient training examples it can be updated to handle new tech stacks and emergent vulnerabilities. With false positives traditionally being a limiting factor for putting such models into production, this innovative hybrid system therefore is attractive for real-world deployment. We feel multi-view architectures could perhaps be explored further, with each view designed in a way that maximises the predictive power of the traffic content, and it might be that the full use of original request, original response, attack request and attack response traffic may not always be needed in some cases to identify a vulnerability. It is also recognised that the explainability of a deep learning model is useful, though this is non-trivial and best analysed fully in a separate paper.

Lastly, with the presented method being an application of supervised deep learning, there may be opportunities to extend the work to unsupervised techniques that could deal with known and unknown vulnerabilities by learning representations for benign, non-vulnerable web app traffic, thus in theory being able to identify potentially vulnerable responses as anomalous deviations from the norm.

REFERENCES

[1] 2019 Cost of Cybercrime Study. [Online: last accessed June 2022]. Ponemon Institute LLC Accenture. https://www.accenture.com/_acnmedia/PDF-96/Accenture-2019-Cost-of-Cybercrime-Study-Final.pdf.

[2] 2021 Report: Cyberwarfare in the C-Suite. [Online: last accessed June 2022]. Cybercrime Magazine Intrusion, Inc. https://1c7fab3im83f5gqiow2qqs2k-wpengine.netdna-ssl.com/wp-content/uploads/2021/01/Cyberwarfare-2021-Report.pdf.

[3] Jader Abreu, Luis Fred, David Macêdo, and Cleber Zanchettin. 2019. Hierarchical Attentional Hybrid Neural Networks for Document Classification. In *Artificial Neural Networks and Machine Learning – ICANN 2019: Workshop and Special Sessions*, Igor V. Tetko, Věra Kůrková, Pavel Karpov, and Fabian Theis (Eds.). Springer International Publishing, Cham, 396–402.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations.* https://openreview.net/forum?id=H1gKYo09tX

[5] Stefan Axelsson. 1999. The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection *(CCS '99)*. Association for Computing Machinery, New York, NY, USA, 1–7.

[6] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.

[7] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (July 2009), 58 pages.

[8] Roberto Doriguzzi Corin, S. Millar, Sandra Scott-Hayward, Jesus Martinez-del Rincon, and D. Siracusa. 2020. Lucid: A Practical, Lightweight Deep Learning Solution for DDoS Attack Detection. *IEEE Transactions on Network and Service Management* PP (02 2020), 1–1.

[9] Kostas Drakonakis, Sotiris Ioannidis, and Jason Polakis. 2020. The Cookie Hunter: Automated Black-Box Auditing for Web Authentication and Authorization Flaws. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1953–1970. https://doi.org/10.1145/3372297.3417869

[10] EUR-Lex. [Online: last accessed June 2022]. Consolidated text: Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1528874672298&uri=CELEX:02016R0679-20160504.

[11] Ruitao Feng, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, and Yang Liu. 2021. A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1563–1578.

[12] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. 2007. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. 87–96.

[13] Anjith George and Sébastien Marcel. 2021. Learning One Class Representations for Face Presentation Attack Detection Using Multi-Channel Convolutional Neural Networks. *IEEE Transactions on Information Forensics and Security* 16 (2021), 361–375.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning, Chapter 12 Applications, 12.4.2 Neural Language Models.* MIT Press. 451–452 pages. http://www.deeplearningbook.org.

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning, Chapter 9 Convolutional Neural Networks.* MIT Press. 321–363 pages. http://www.deeplearningbook.org.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning, Chapter 10 Sequence Modeling: Recurrent and Recursive Nets, 10.2 Recurrent Neural Networks.* MIT Press. 371–372 pages. http://www.deeplearningbook.org.

[17] A. Gupta. 2018. Intelligent code reviews using deep learning, KDD '18, Deep Learning Day.

[18] Zied Haj-Yahia, Adrien Sieg, and Léa A. Deleris. 2019. Towards Unsupervised Text Classification Leveraging Experts and Word Embeddings. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics.* Association for Computational Linguistics, Florence, Italy, 371–379. https://www.aclweb.org/anthology/P19-1036

[19] William G. J. Halfond and Alessandro Orso. 2005. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1–7. https://doi.org/10.1145/1082983.1083250

[20] Danny Hendler, Shay Kels, and Amir Rubin. 2018. Detecting Malicious PowerShell Commands Using Deep Neural Networks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (Incheon, Republic of Korea) *(ASIACCS '18)*. Association for Computing Machinery, New York, NY, USA, 187–197.

[21] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence* (New York, New York, USA) *(IJCAI'16)*. AAAI Press, 1606–1612.

[22] Information Commissioner's Office. [Online: last accessed June 2022]. International transfers after the UK exit from the EU Implementation Period. https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/international-transfers-after-uk-exit/.

[23] Jaeyoung Kim, Sion Jang, Eunjeong Park, and Sungchul Choi. 2020. Text classification using capsules. *Neurocomputing* 376 (2020), 214–221. https://www.sciencedirect.com/science/article/pii/S0925231219314092

[24] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Association for Computational Linguistics, Doha, Qatar, 1746–1751. https://www.aclweb.org/anthology/D14-1181

[25] Yuji Kosuga, Kenji Kono, Miyuki Hanaoka, Miho Hishiyama, and Yu Takahama. 2007. Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. 107–117. https://doi.org/10.1109/ACSAC.2007.20

[26] Chenhao Lin and Ajay Kumar. 2019. A CNN-Based Framework for Comparison of Contactless to Contact-Based Fingerprints. *IEEE Transactions on Information Forensics and Security* 14, 3 (2019), 662–676.

[27] Avinash Madasu and Vijjini Anvesh Rao. 2019. Sequential Learning of Convolutional Features for Effective Text Classification. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Association for Computational Linguistics, Hong Kong, China, 5658–5667. https://www.aclweb.org/anthology/D19-1567

[28] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[29] Stuart Millar, Niall McLaughlin, Jesus Martinez del Rincon, and Paul Miller. 2021. Multi-view deep learning for zero-day Android malware detection. *Journal of Information Security and Applications* 58 (2021), 102718. https://www.sciencedirect.com/science/article/pii/S2214212620308577

[30] Govind Mittal, Paweł Korus, and Nasir Memon. 2021. FiFTy: Large-Scale File Fragment Type Identification Using Convolutional Neural Networks. *IEEE Transactions on Information Forensics and Security* 16 (2021), 28–41.

[31] Jessica Moore, B. Gelman, and D. Slater. 2019. A Convolutional Neural Network for Language-Agnostic Source Code Summarization. In *ENASE*.

[32] Benjamin Morin, Ludovic Mé, Hervé Debar, and Mireille Ducassé. 2002. M2D2: A Formal Data Model for IDS Alert Correlation. In *Recent Advances in Intrusion Detection*, Andreas Wespi, Giovanni Vigna, and Luca Deri (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–137.

[33] OWASP Open Web Application Security Project. [Online: last accessed June 2022]. Top Ten Web Application Security Risks. https://owasp.org/www-project-top-ten/.

[34] Yao Pan, Fangzhou Sun, Zhongwei Teng, Jules White, Douglas C. Schmidt, Jacob Staples, and Lee Krause. 2019. Detecting web attacks with end-to-end deep learning. *J Internet Serv Appl* 10 (2019).

[35] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[36] Tadeusz Pietraszek. 2004. Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In *Recent Advances in Intrusion Detection*, Erland Jonsson, Alfonso Valdes, and Magnus Almgren (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 102–124.

[37] Rapid7. [Online: last accessed June 2022]. Under The Hoodie 2020. https://www.rapid7.com/research/report/under-the-hoodie-2020/.

[38] Wei Rong, Bowen Zhang, and Xixiang Lv. 2019. Malicious Web Request Detection Using Character-Level CNN. In *Machine Learning for Cyber Security*, Xiaofeng Chen, Xinyi Huang, and Jun Zhang (Eds.). Springer International Publishing, Cham, 6–16.

[39] Sagar Samtani, Murat Kantarcioglu, and Hsinchun Chen. 2020. Trailblazing the Artificial Intelligence for Cybersecurity Discipline: A Multi-Disciplinary Research Roadmap. *ACM Trans. Manage. Inf. Syst.* 11, 4, Article 17 (Dec. 2020), 19 pages.

[40] Lwin Khin Shar, Lionel C. Briand, and Hee Beng Kuan Tan. 2015. Web Application Vulnerability Prediction Using Hybrid Program Analysis and Machine Learning. *IEEE Transactions on Dependable and Secure Computing* 12, 6 (2015), 688–707.

[41] Robin Sommer and Vern Paxson. 2003. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (Washington D.C., USA) *(CCS '03)*. Association for Computing Machinery, New York, NY, USA, 262–271.

[42] F. Sun, P. Zhang, J. White, D. Schmidt, J. Staples, and L. Krause. 2017. A Feasibility Study of Autonomically Detecting In-Process Cyber-Attacks. In *2017 3rd IEEE International Conference on Cybernetics (CYBCONF)*. 1–8.

[43] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. 2019. Import2vec Learning Embeddings for Software Libraries. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 18–28. https://doi.org/10.1109/MSR.2019.00014

[44] Verizon. [Online: last accessed June 2022]. 2020 Data Breach Investigations Report: Official. https://enterprise.verizon.com/resources/reports/dbir/.

[45] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. 2010. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *J. Mach. Learn. Res.* 11 (Dec. 2010), 3371–3408.

[46] I. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning, Chapter 7 Regularization for Deep Learning, 7.8 Early Stopping.* MIT Press. 239–240 pages. http://www.deeplearningbook.org.

[47] Huanting Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958.

[48] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 171–180.

[49] W. Xu and Y. Tan. 2020. Semisupervised Text Classification by Variational Autoencoder. *IEEE Transactions on Neural Networks and Learning Systems* 31, 1 (2020), 295–308. https://doi.org/10.1109/TNNLS.2019.2900734

[50] Wenpeng Yin and Hinrich Schütze. 2015. Multichannel Variable-Size Convolution for Sentence Classification. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, Beijing, China, 204–214. https://www.aclweb.org/anthology/K15-1021

[51] Weike You, Hong Zhang, and Xianfeng Zhao. 2021. A Siamese CNN for Image Steganalysis. *IEEE Transactions on Information Forensics and Security* 16 (2021), 291–306.

[52] Yuchen Zhou and David Evans. 2014. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 495–510.

[53] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. AUTHSCOPE: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 799–813. https://doi.org/10.1145/3133956.3134089