Open Science in Software Engineering: A Study on Deep Learning-Based Vulnerability Detection

Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai

Abstract—Open science is a practice that makes scientific research publicly accessible to anyone, hence is highly beneficial. Given the benefits, the software engineering (SE) community has been diligently advocating open science policies during peer reviews and publication processes. However, to this date, there has been few studies that look into the status and issues of open science in SE from a systematic perspective.

In this paper, we set out to start filling this gap. Given the great breadth of SE in general, we constrained our scope to a particular topic area in SE as an example case. Recently, an increasing number of deep learning (DL) approaches have been explored in SE, including *DL-based software vulnerability detection*, a popular, fast-growing topic that addresses an important problem in software security. We exhaustively searched the literature in this area and identified 55 relevant works that propose a DL-based vulnerability detection approach. This was then followed by comprehensively investigating the four integral aspects of open science: *availability*, *executability*, *reproducibility*, and *replicability*.

Among other findings, our study revealed that only a small percentage (25.5%) of the studied approaches provided publicly *available* tools. Some of these available tools did not provide sufficient documentation and complete implementation, making them not *executable* or not *reproducible*. The uses of balanced or artificially generated datasets caused significantly overrated performance of the respective techniques, making most of them not *replicable*. Based on our empirical results, we made actionable suggestions on improving the state of open science in each of the four aspects. We note that our results and recommendations on most of these aspects (*availability*, *executability*, *reproducibility*) are not tied to the nature of the chosen topic (DL-based vulnerability detection) hence are likely applicable to other SE topic areas. We also believe our results and recommendations on *replicability* to be applicable to other DL-based topics in SE as they are not tied to (the particular application of DL in) detecting software vulnerabilities.

Index Terms—Open science, availability, executability, reproducibility, replicability, case study, vulnerability detection, deep learning

1 Introduction

Open science advocates that scientific research be accessible to all [1]. It aims to allow knowledge to be shared with other researchers or amateurs smoothly [1]. Open science has been frequently mentioned by the media, books, and government white papers in recent years [2]. It is vital for advancement of science in almost any field. Specifically, (1) it allows us to validate the credibility of research findings so that we know where/how far we have got; (2) it enables us to rigorously evaluate and compare the existing studies with quantitative measurement, so that we can understand the strengths and limitations of current works in order to determine the next steps; (3) by fostering reusability, it accelerates the progress of science by allowing researchers to build on previous research achievements, instead of reinventing wheels thereby slowing down the overall process of scientific discovery.

Despite these well-known benefits of open science and that it has been widely embraced in many different scientific disciplines, it has not yet become a real norm in software engineering (SE) [3]. In recent years, major SE venues started explicitly advocating/enforcing open science policies during paper submission and review processes. Yet it remains largely unclear what the current status of open science in SE is and, more importantly, what make successes and what cause failures in upholding the open science principles in our community. Answers to these questions would not only help us understand the possible gaps, they would also inform practical strategies to improve the open-science landscape in the SE community and beyond.

Motivated by the pressing need for these answers, we aim to investigate the state and issues of open science in SE from a systematic perspective. However, SE has a large breadth in general (e.g., testing, maintenance, evolution). Thus, it is difficult to investigate open science in different SE domains comprehensively using a unified methodology (e.g., the metrics of reproducibility and replicability in different SE domains may be different). Also, many deeper insights are tied with more specific topics. We aim to understand the underlying reasons why certain open-science aspects look like as they do now, not only just what the status is, because understanding those reasons is essential for providing actionable suggestions that help improve the open science outlook. Therefore, we would like to start with a particular topic area in SE as an example case to realistically carry out the investigation.

Recent years have witnessed increasing use of deep learning (DL) in traditional software engineering (SE) problems and tasks [4]. In particular, security is a growing quality concern for modern software, and accordingly vul-

- *Yu Nong, Rainy Sharma, and Haipeng Cai are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA.*
- *Abdelwahab Hamou-Lhadj is with the Department of Electrical and Computer Engineering, Concordia University, Montreal, Quebec, Canada.*
- *Xiapu Luo is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong SAR, China.*
- *Corresponding author: Haipeng Cai; Email: haipeng.cai@wsu.edu*

Manuscript received October 8, 2021; revised Month Day, 2022.

nerability detection has become a heavily studied topic in the general SE domain. According to our survey, various DL-based software vulnerability detection techniques have been being introduced in the last five years (2016-2020) (see Figure 2). It is essential to understand the status of, and potential issues with, open science in this area so as to advance it by taking advantage of the general merits of open science. Thus, we chose *DL-based vulnerability detection* as the example area to initiate our effort of understanding hence helping improve open science in SE.

Prior studies concerning open science in SE exist, but they were mostly focused on hence limited to relevant definitions [3], [5], [6], [7], [8], [9], benefits [10], [11], [12], [13], challenges [14], [15], and guidelines [16], [17]. Others addressed open science in specific areas of SE, yet solely based on literature reviews [18], [19], [20]. A few studies investigated open science by checking whether available source code could be obtained, but they did not reproduce/replicate the original experiment results using the code when available [21], [22], [23], although some also investigated executability in addition to availability [24]. There have also been studies investigating reproducibility and replicability of DL-based techniques. Yet they considered only a single model [25], examined a narrow facet of open science (e.g., documentation sufficiency) [26] without actual reproduction/replication experiments, or did not provide in-depth analysis of failures/gaps in open science [27].

In this paper, we exhaustively searched the current literature on DL-based software vulnerability detection techniques published and indexed by Google Scholar by October 2020. Based on the resulting 55 relevant papers, we conducted in-depth statistical and case studies to answer four research questions (RQs) corresponding to the four aspects of open science:

1) **RQ1 (availability)**: how often does a paper make the tool for the proposed technique publicly available and how does it provide the relevant information?
2) **RQ2 (executability)**: are the available tools executable, and what is it that makes them executable or not?
3) **RQ3 (reproducibility)**: are the executable tools reproducible, and what are the success patterns (if reproducible) or failure causes (if not reproducible)?
4) **RQ4 (replicability)**: are the reproducible tools replicable, and what are the success patterns or failure causes?

Guided by the these questions, our study revealed the following major findings, among many others:

1) Only a small portion (25.5%) of the 55 papers provided publicly *available* tools. For the success cases, the ways of obtaining their tool links varied, limiting wide access by the general public.
2) Among the available tools, 54.5% did not provide sufficient documentation. Moreover, 27.3% of them had incomplete and/or non-functional implementation, making them not *executable*.
3) Most (87.5%) of the *executable* tools were *reproducible* and most (71.4%) of the reproducible ones had datasets, implementation, and configurations consistent with those described in original papers. One of them was perfectly reproduced using the pre-trained DL model it provided.
4) All the *executable* tools that provided sufficient documentation were *strongly reproducible* (i.e., having $\leq$1% deviation of F1 accuracy from the originally reported).
5) Only 14.3% of the *reproducible* tools were *replicable* when we applied their pre-trained models to a different real-world testing dataset (i.e., *partial replicable*). Only 28.6% of them were *replicable* when we re-trained and tested their models both against a different real-world dataset (*full replicable*).
6) 57.1% of the *reproducible* tools could not process our program samples, or could only process a part of the samples. This significantly limited their *replicability* and practical use for real-world vulnerability detection.
7) The datasets originally used by 57.1% of the *reproducible* tools (not the same set as the 57.1% above) were highly balanced and/or entirely/mostly artificially generated, leading to significantly overrated performance in their original evaluation and making them not *replicable*.
8) The highest F1 score in our replication experiments was only 52%, indicating that the existing DL-based software vulnerability detection techniques are likely insufficient for practical adoption.
9) By comparing our study results with those of other open science related studies in SE, we noticed that, while not all, many of the open science issues we found were prevalent in other SE areas as well, suggesting that our findings and insights are likely applicable beyond the studied area (of DL-based vulnerability detection).

Based on these findings, we further distilled key lessons learned and provided extensive actionable recommendations for the community and researchers to improve open science in the area of DL-based software vulnerability detection, as well as in other SE areas and beyond. To better support open science, we have released all of the code and datasets used in our study on Figshare:

https://figshare.com/s/e048fa191503393275a1

2 Background and Motivation

In this section, we first briefly describe the status quo of open science in SE. Then, we discuss the open science dilemma in the chosen example cases of DL-based software vulnerability detection techniques, to justify the urgent need for a systematic, dedicated study on open science like ours.

2.1 Open Science in SE

Open science suggests that researchers make their publications, raw data, or physical samples produced in their research activities accessible to all levels of people in the society, so that knowledge can be shared without obstacles [2]. Open science has many benefits. It has been proved that open science helps researchers succeed in terms of publication citations, media attention, potential collaborators, job opportunities and funding opportunities [28].

In SE, open science is equally if not more important. It requires the researchers to (1) make the source code and data produced in their research activities *available*, (2) ensure that their source code is *executable* and other people can *reproduce* the experiment results by *directly* re-running their source code using the default/original experimental setup, and (3) ensure that the experiment results can still be *replicated* when

changing the experimental setup (e.g., changing the testing data) [3]. However, while open science has been generally accepted in many scientific disciplines, in SE, open science has not been comprehensively investigated, and thus has not yet become a norm in the community [3].

2.2 DL-Based Software Vulnerability Detection

Software security is an important topic in SE [29], and software vulnerability detection is one of the most essential tasks in software security [30]. Recently, deep learning (DL) has been increasingly exploited to detect software vulnerabilities automatically by using the patterns learned from existing vulnerability data. Figure 1 shows the workflow of a typical DL-based software vulnerability detection technique. First, the training and testing program samples are preprocessed according to the format preferred by the chosen DL model. Second, the DL model learns the vulnerability patterns from the training samples. Finally, the trained DL model is used to predict whether a given program sample is vulnerable or not. We can use the testing samples to assess the performance of the *model*.

DL-based software vulnerability detection is a booming area. According to our literature review, 55 papers that each introduced a new detection technique have been published in the recent five years (2016-2020). Many of these papers have reported impressive detection accuracy (up to 90% F1 score) of the proposed approach [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41]. Figure 2 shows that the majority (38) of these papers were published in just the last two years (2019 and 2020), indicating the dramatic growth of the popularity of this topic.

2.3 Motivating Examples

The lack of open science in SE significantly impedes scientific advancement of the field, which has gained growing attention by our community in the past few years. Now that the promotion (sometimes enforcement) of open science has been in place for some time already, it remains unclear what the current status is. In our own experiences with SE research concerning various topics, we have encountered and witnessed multiple instances of the failure to uphold the open science practice being critical obstacles for research.

As more and more papers on DL-based vulnerability detection techniques are being published, we intended to conduct a comprehensive empirical study to assess and compare the existing techniques in this area. However, we had considerable difficulties when we started collecting and setting up the tools that implement those techniques. A common cause was that many of the tools were either not available or not executable, and in even more cases we could not reproduce the original results, making us doubt about our setup. Without the ability to even reproduce, it would make no sense to proceed with replication, which was required for us to do a fair comparison based on the same datasets. We thus had to suspend our original study. This initially motivated us to look into the open science problem with a dedicated effort.

In fact, other researchers have encountered similar difficulties. For example, in [35], the authors intended to use a technique introduced in another paper as a baseline for

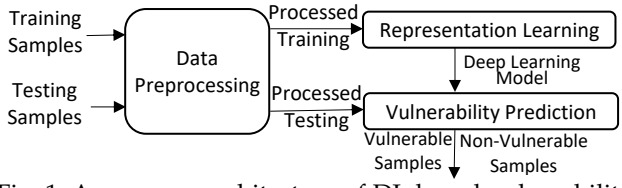

Fig. 1: A common architecture of DL-based vulnerability detectors.

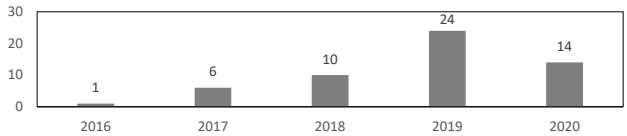

Fig. 2: Research trend on DL-based vulnerability detection techniques in terms of # publications (y axis) over the past five years (x axis).

their study. However, the implementation of that technique was not publicly available. As a result, these authors chose to re-implement the baseline technique. Despite their best effort, they were not able to reproduce the experiment results using their own implementation. Not only did the re-implementation take the researchers' time that could have otherwise been spared, the eventual outcome of the extra effort also left some questions unanswered (e.g., it remained unsure if the re-implementation was correct). Apparently, these were barriers for that research study [35].

In both examples, the shortage of open science for previous research impeded or even disabled current/future research that relies on those prior works. In this context, it is essential to gain a deeper understanding and comprehensive view of the contemporary open-science practice. Moreover, it is helpful to reveal the patterns of successes and investigate the symptoms and possible causes of failures, with the current practice, so as to understand how the SE community may improve the state of open science.

> While drawing growing attention, the status and gaps of open science in SE have not been well understood. In DL-based vulnerability detection as one example topic, failures to uphold open-science principles already appeared as critical obstacles for ongoing and future research. The SE community thus has an urgent quest to understand the current practice of open science and ways to improve it.

3 Study Design

In this section, we elaborate the design of our study. We start with an overview of the study process. Then, we present the design details for each research question (RQ), including the motivation, experimental method, and the dimensions in characterizing the studied tools for answering the RQ.

3.1 Process Overview

Figure 3 depicts the overall design of our study. We began with an extensive literature review on Google Scholar with a general search term "`deep learning software vulnerability`" against anywhere in any paper. We adopted this highly-conservative approach in order to scoop

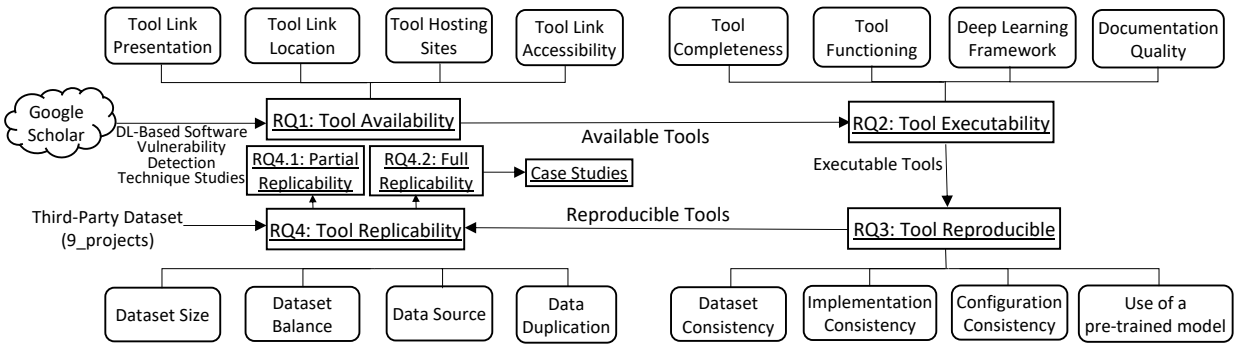

Fig. 3: An overview of our study design.

relevant papers as thoroughly as possible[1], at the cost of additional manual inspection effort in the following steps. Next, for each paper returned from the search, we looked into its citing and cited papers to collect more relevant ones. Then, we carefully examined each of the collected papers and selected the ones that introduced a new DL-based vulnerability detection technique. The long (over 500-man-hour) collection process ended with 55 relevant research papers, as summarized in Figure 2 and listed in Table 2.

These papers and the proposed techniques/tools formed the basis of our study, guided by four high-level RQs: *availability*, *executability*, *reproducibility*, and *replicability*[2], which correspond to four aspects of open science. Each RQ was examined using multiple dimensions (e.g., how the link to tool is presented) so that we could uncover success/failure patterns and provide actionable insights. Only the tools that passed a current RQ's success criterion went to the next RQ for further investigation, given the dependencies among the criteria (e.g., a reproducible tool must be executable).

3.2 RQ1. Availability

Motivation. Publicly *available* tools (i.e., source code and datasets) are essential for open science in SE. They are the bases for other researchers to access the raw data and technique implementation. Thus, in this RQ, we investigated *how often/commonly the studied DL-based vulnerability detection works provided publicly available tools*.

Method. For each paper we surveyed, we checked whether an accessible link for its introduced tool was provided in the main text or footnotes of the paper. If there was no such a link provided in the paper, we searched on the Internet to check if the authors had uploaded the tool to a hosting site (e.g., GitHub or GitLab). We also sent up to three emails to the authors to ask for the tool if we could not find it by ourselves. We considered a tool *available* if we successfully obtained the source code and dataset.

Dimensions. We examined the following four dimensions in characterizing the *availability* of each tool:

- **Tool Link Presentation:** If the tool link was available in the paper, how it was presented (e.g., as raw text, as a reference, or as a hyperlink).
- **Tool Link Location:** Where we found the tool link (e.g., main text or footnote in the paper, searching on the Internet, authors' emails).
- **Tool Hosting Site:** Which hosting site the available tool was uploaded to for public access (e.g., GitHub or GitLab).
- **Tool Link Accessibility:** Whether the tool link we obtained was valid for downloading the tool.

3.3 RQ2. Executability

Motivation. Executable tools are also important for open science in SE. A broken tool cannot generate valid experiment results, and thus has little value for other researchers. Thus, in this RQ, we investigated *whether the available tools worked properly to execute the whole experiments in their studies*.

Method. To better focus on the dominant class of DL-based vulnerability detection techniques, in RQ2, RQ3, and RQ4, we only investigated tools targeting C/C++, source code or binary. The rationale behind this was that (1) C/C++ is the most vulnerable language [43], and (2) many critical software systems are written by C/C++. As documentation is the key reference that informs about how to execute the tools, for each available tool, we first evaluated whether the documentation of the tool was *sufficient*. However, there is no commonly accepted criterion for the sufficiency of documentation for open science. For our study, we deem a documentation sufficient when we were able to figure out how to set up and use the tool following the documentation based on our knowledge about DL and vulnerability detection. If the documentation provided *sufficient* information, we followed its instructions to set up the DL framework, install the tool, and execute the experiments. If, due to insufficient documentation, we were not able to execute the experiments, we contacted the authors or manually checked the source code to figure out how to run the experiments. If we successfully executed the whole experiment of a tool and obtained valid results after doing these, we considered the tool *executable*.

Dimensions. We examined the following four dimensions in analyzing the success/failure patterns with *executability*:

- **Tool Completeness:** Whether the tool provided all the necessary components (e.g., raw data, data preprocessing code, DL model to be trained, etc.) that are needed for executing the entire original experiment.
- **Tool Functioning:** Whether all the components of the tool were functional such that they enable us to produce valid experiment results.
- **Deep Learning Framework:** The DL framework (e.g., Tensorflow, PyTorch) on which the tool was based.

1. We concluded our search on 10/31/2020. Thus, any papers that were not indexed by Google Scholar by that date were dismissed.
2. We use the replicability/reproducibility definitions by ACM [42].

- **Documentation Quality:** Whether the documentation for the tool was *sufficient* so that we can execute the entire experiment without external resources.

3.4 RQ3. Reproducibility

Motivation. Ideally, the experiment results in a paper can be reproduced by re-running the tool using the same experiment setup. If the experiment results cannot be reproduced, the reliability and even credibility of the tool is questionable. Thus, in this RQ, we investigated *whether the executable tools can reproduce the experiment results in the original papers.*

Method. Based on the definition by ACM [42], the reproducibility experiments should be done with the same measurement procedure, measuring system, and operating conditions. Thus, we should use the same source code, input data, dependency libraries, operating systems, and hardware as the one used by the authors to do the experiments. However, realistically it is very difficult to use exactly the same hardware and operating system versions. We could not buy all the different hardware and install all the different operating system versions to match the original. Virtual machines may seem to be a solution, but they often did not work either because hardware cannot always be virtualized (esp. GPUs, which are a major part of the computing environment for DL-based tools). Therefore, we followed the methodology in [24]. We used the slightly weaker definition of *reproducibility* and ensured the use of the same source code or executable, input data, and dependency libraries.

For each tool, we executed the experiment using the default configuration. Then, the tool generated experiment results and reported the accuracy of predicting vulnerabilities on the testing set. For some tools, the prediction accuracy was presented in terms of different metrics, such as precision, recall and F1 score. In our study, we only considered F1 score, because (1) it accounts for both precision and recall, and is a common measure for machine learning [44], and (2) it allowed us to compare the *reproducibility* and *replicability* of different tools consistently. To measure the *reproducibility* of a tool, we computed the *deviation* between the F1 score in our experiment and the one reported in the original paper, as follows:

$$Deviation = \frac{F1\ in\ our\ experiment - F1\ in\ original\ paper}{F1\ in\ original\ paper}$$

If the absolute value of the *deviation* was $\leq 1\%$, we considered the tool *reproducible*. If the value was $> 1\%$ but $\leq 5\%$, we considered the tool *weakly reproducible*. If the value was $> 5\%$, we considered the tool *not reproducible*. These thresholds were chosen based on the intervals used in statistical testing [20] (e.g., 1% and 5% probabilities used in statistical significance analysis [45]).

Because of the randomness in training DL models, the F1 scores reported may be expectedly different across multiple runs of model training on the same dataset using the same configuration [46]. To mitigate relevant biases, when evaluating *reproducibility* and *replicability*, we repeated the experiment against each tool five times and reported the F1 score with the smallest absolute value of *deviation*.

Dimensions. We examined the success/failure patterns with *reproducibility* in the following four dimensions:

- **Dataset Consistency:** Whether the dataset used in our experiment was consistent with the one described in the original paper.
- **Implementation Consistency:** Whether the tool implementation of the proposed technique was consistent with the one described in the original paper.
- **Configuration Consistency:** Whether the configuration of the tool (e.g., parameter setting of the DL model) was consistent with the one described in the original paper.
- **Use of a Pre-Trained Model:** Whether a pre-trained DL model was provided so that we could skip the training phase and evaluate the model on the testing set directly.

3.5 RQ4. Replicability

To evaluate the replicability comprehensively, we considered two types of replicability: *partial replicability* and *full replicability*, as defined below. We separately discuss them in RQ4.1 and RQ4.2 respectively.

3.5.1 RQ4.1 Partial Replicability

Definition. For a DL-based tool, it is reasonable to use the tool as trained on the dataset as used in the original paper. If we can replicate the originally reported results using such a pre-trained model against a different (testing) dataset, we consider it *partial replicable*.

Motivation. According to the ACM definition [42], the replicability experiments should be performed on different measuring systems. Since we measured the DL-based vulnerability detectors with benchmark datasets, we used a *third-party* (i.e., different from the ones used in the original papers) dataset for the replicability experiments. Such a dataset does not have to have a similar distribution to the one in the original papers and can be any kinds of vulnerability datasets. In fact, with respect to its goal, replication is expected to validate research work in real-world use scenarios, where the distribution of users' dataset is unlikely to be guaranteed to be the same as that of the original one.

Ideally, the *trained* DL models should be ready for real-world application tasks; that is, the models we trained in our reproducibility experiments in RQ3 should have similar performance when we test them on a *third-party* vulnerability dataset. If the performance results cannot be replicated, the respective tool has a *partial replicability* issue, which means the (pre-trained) model may not be ready for practical use. This issue is a major barrier in open science, as discussed in [3]. Thus, in this RQ, we investigated *whether the models we trained in our reproducibility experiments can replicate the originally reported performance results when we test them on a third-party, real-world dataset.*

Method. We used the `9_projects` dataset introduced in [47] for our replicability experiments. This dataset contained a large number of program samples from 9 open-source projects, thus it is an appropriate benchmark for evaluating DL-based vulnerability detection techniques. Such a dataset also enabled us to consistently compare the replicability across different tools. Table 1 shows the numbers of vulnerable and non-vulnerable samples in this dataset [3].

3. The numbers are different from the ones in the original paper because the authors have updated the dataset after paper publication.

The dataset came with both C/C++ `source code` and compiled C/C++ `binary code` versions, allowing us to evaluate tools that target different code formats.

We first checked the documentation, source code, or contacted the authors to ensure that a tool was compatible with third-party program samples. Those tools that were not able to process third-party program samples were considered *not reproducible* immediately. Then, for the remaining ones, we did the *partial replicability* experiments on the `9_projects` dataset. For the tools working with C/C++ source code, we used the `source code` version of the dataset. For those working with compiled C/C++ binary code, we used the `binary code` version.

For each tool, we first split the `9_projects` dataset into a training set and a testing set. Then, we tested the model which was trained in our reproducibility experiments on the split testing set. We used the same testing sets for both RQ4.1 and RQ4.2 to enable fair comparisons and more insights.

We then obtained the F1 score of each tool, and computed the *deviation*, in the same fashion as we gauge *reproducibility*, to measure *replicability*. Similarly, we considered a tool *replicable* if the absolute value of the deviation was $\leq 1\%$, *weakly replicable* if the value was $> 1\%$ but $\leq 5\%$, and *not replicable* if the value was $> 5\%$. As justified earlier, we repeated each experiment for each tool five times and reported the F1 score with the smallest *deviation*.

Dimensions. For each tool, we analyzed the partial replicability successes/failures from the dataset perspective, by comparing the one we used with the one used in the original paper in the following four dimensions:

- **Dataset Size:** The number of samples in the dataset.
- **Dataset Balance:** The proportion of positive (vulnerable) samples in the dataset.
- **Data Source:** Whether the samples in the dataset were artificially generated or from real-world projects.
- **Duplication Rate:** The proportion of samples that each has $> 95\%$ similarity to at least one other sample in the dataset. We used a dedicated tool, `difflib` [4], to compute the similarity between two samples (programs).

3.5.2 RQ4.2 Full Replicability

Definition. For differentiating from *partial replicability*, we refer to the standard definition of replicability [42] for *full replicability*. That is, a tool is *full replicable* if we can replicate its originally reported performance results against training and testing datasets that are both different from the originally used ones.

Motivation. In RQ4.1, we investigate *partial replicability*. Being *partial replicable* or not indicates whether a pre-trained model is ready for use in an off-the-shelf manner. However, such results (for RQ4.1) would only reflect the performance of the pre-trained models, not accounting for the capabilities of the holisit DL-based vulnerability detection technique. In practice, users may desire to retrain the DL model on new datasets from their particular use scenarios. For a comprehensive open-science assessment, in RQ4.2 we investigated *full replicability* for each tool: *whether the technique can replicate the originally reported performance results when we retrain and re-test the model both on third-party, real-world datasets.*

4. https://docs.python.org/3/library/difflib.html

TABLE 1: The number of vulnerable and non-vulnerable samples in the `9_projects` dataset.

Dataset	# of vulnerable samples	# of non-vulnerable samples	Total
Source Code	1471	60132	61603
Binary Code	210	20646	20856

Method. As for RQ4.1, we used the `9_projects` dataset for our *full replicability* experiments.

For each tool, we used the split training set to re-train the DL model. Then, we tested the re-trained model on the same split testing set as for RQ4.1. We then obtained the F1 score, computed the deviation, and determined whether the tool was *replicable*, *weakly replicable*, or *not replicable* similarly to what we do for RQ4.1.

Dimensions. For each tool, we analyzed the full replicability successes/failures, also from the dataset perspective, by comparing the one we used with the one used in the original paper in the same four dimensions as for RQ4.1.

3.5.3 Case Studies

To further understand the impact of dataset on replicability, we conducted additional case studies, focusing on two of the four dataset dimensions, *dataset size* and *dataset balance*, which we observed had substantial impacts.

Impact of dataset size. For each tool, we reduced the dataset (`9_projects`) size (number of samples) to 70%, 40%, and 10% of the whole, separately, while not changing the dataset balance (as detailed later). Then, we split the reduced datasets into training sets and testing sets, and performed *full reproducibility* experiments on these reduced datasets to investigate how the dataset size impacted the tool performance.

Impact of dataset balance. For each tool, we kept the dataset size at 10% so that we could experiment with as many different data balance ratios (proportion of positive samples) as possible without changing the total dataset size. We found that the original dataset balance ratios of `9_projects` when applied to different tools were all very small (<6%, see Figure 9). Thus, we only considered increasing (but not decreasing) the dataset balance b, to $4b$, $7b$, and $10b$. We then again split the datasets into training and testing sets, and did *full reproducibility* experiments on these increased-balance datasets to investigate how the dataset balance impacted the tool performance.

4 Results

In this section, we present and discuss the results of our study by answering the four RQs. For each RQ, we first show the overall status/statistics. Second, we discuss the success cases and point out their possible limitations. Third, we analyze the failure cases and discuss their reasons. Finally, we give a summary for that RQ.

4.1 RQ1. Availability

Overall status/statistics. Table 2 shows the tool availability results of the 55 investigated papers/studies. The color of each row indicates whether a tool for that study was publicly available (green for *available* and red for *unavailable*). For each study, results in the four dimensions we investigated are shown by different abbreviations, and the full names for

The preparation of the dataset followed two steps. The first step consists of parsing the original dataset to separate each CS then calculate its code metrics. The second step consists of eliminating any redundant data. The final version of the dataset [32] which is used in experiments contains 95351 instances. To work with the Java API of Weka, the actual version of the dataset is in the ARFF (Attribute-Relation File Format) format.

Fig. 4: One example of tool link presentation: the link is presented as a reference, which is from the study ZMVP [34].

4) We also open source all our code and data we used in this study for broader dissemination. Our code and replication data are available in https://git.io/Jf6IA.

Fig. 5: Another example of tool link presentation: the link is presented in the main body of the paper [35] as a raw text.

these abbreviations are shown below the table. For example, we found the publicly available tool for TRL [31] (row 1, which is filled by the green color). The tool link was found in the paper as a raw text (RT), in the footnote of the experiments section (FE). The tool was uploaded to GitHub for sharing (GH), and the tool link was accessible so that we could download the tool.

After trying our best to look for the available tools of these 55 papers, we only found 14 actually having the tool available. That is, only a low ratio (25.5%) of the studies provided publicly available source code and datasets for the techniques they introduced. This indicates a very poor observance of the open science practice at least in the area of DL-based software vulnerability detection, potentially hindering the advancement of this area. That is, to assess or improve an existing DL-based software vulnerability detection technique, it is likely that other researchers or interested users have to re-implement it. This would waste much time/resources; moreover, the custom implementation may not be consistent with the one used in the original study.

Success cases/possible limitations. We then inspected the 14 papers/studies which provided publicly available tools. We noticed that 11 of these presented the tool links in their papers, while the other 3 (SaBabi [38], VulDeePecker [39], and BiVuld [41]) did not—their tool links could be only obtained by searching on the Internet or contacting the authors via email. In the 11 studies, all the links were presented as raw texts in the papers except one (ZMVP [34]), which was presented as a paper reference.

We found that the presentations of the tool links affected the difficulties for the readers to find them. Figures 4 and 5 show the examples of presenting tool links in different ways. In Figure 5, presenting the tool link as a raw text in the paper is conspicuous, thus readers can find the link easily by searching relevant keywords (e.g., http). In contrast, in Figure 4, presenting the tool link as a reference is not as effective, because readers may only find the tool link by checking the context of each reference. The latter practice compromises availability hence hinders open science.

Although most of the papers presented their tool links in raw texts, which were conspicuous, we noticed that the tool link locations were more varied. In the 11 studies which provided the tool links in the papers, the links were in different sections (Introduction, Methodology, or Experiments). Besides, we noticed that some papers presented the tool links in the footnotes rather than the main texts, as Figure 6 shows. Furthermore, we also found one more available tool (SaBabi [38]) by searching on the Internet (we confirmed

sorted data at Github[1]. In summary, the contributions of this paper are three-fold:

- We propose a deep learning framework utilizing heterogeneous vulnerability-relevant data sources based on two independent deep representation learning networks capable of extracting the useful features for software vulnerable code detection.

1. https://github.com/DanielLin1986/RepresentationsLearningFromMulti_domain

Fig. 6: An example of tool link presented in a footnote of the paper [32].

TABLE 2: Results on tool availability (along with respective venue ranks)

Study	Tool Link Presentation	Tool Link Location	Tool Hosting Site	Tool Link Accessibility	Venue Ranking
TRL [31]	RT	FE	GH	✓	Unranked
RLMD [32]	RT	FI	GH	✓	A
LAVDNN [33]	RT	MM	GH	✓	Unranked
ZMVP [34]	RF	MM	GH	✓	Unranked
ReVeal [35]	RT	MI	GH	✓	A*
DCKM [36]	RT	ME	GH	✓	Unranked
MDSeqVAE [37]	RT	FE	GH	✓	A*
SaBabi [38]	NA	SI	GH	✓	Preprint
VulDeePecker [39]	NA	AM	GH	✓	A*
SySeVR [40]	RT	MI	GH	✓	A
BiVuld [41]	NA	AM	GH	✓	Unranked
TAP [48]	RT	MI	GH	✓	Unranked
VulHunter [49]	RT	MI	GH	✓	B
Project Achilles [50]	RT	MM	GL	✓	A*
VulSeeker-Pro [51]	RT	FE	GH	✗	A*
Russell [52]	NA	NA	NA	✗	C
Devign [53]	NA	NA	NA	✗	A*
VulDeeLocator [54]	NA	NA	NA	✗	A
μVulDeePecker [55]	NA	NA	NA	✗	A
Choi [56]	NA	NA	NA	✗	A*
DCDDA [57]	NA	NA	NA	✗	A
DeepBalance [58]	NA	NA	NA	✗	A*
RNNBinary [59]	NA	NA	NA	✗	Unranked
BVDetector [60]	NA	NA	NA	✗	A
MinIntermediate [61]	NA	NA	NA	✗	Unranked
Harer [62]	NA	NA	NA	✗	A*
CDVuld [63]	NA	NA	NA	✗	A
StaticCFVul [64]	NA	NA	NA	✗	B
VulSniper [65]	NA	NA	NA	✗	A*
TFI-DNN [66]	NA	NA	NA	✗	Unranked
Caesar [67]	NA	NA	NA	✗	B
CPGVA [68]	NA	NA	NA	✗	Unranked
AI4VA [69]	NA	NA	NA	✗	Preprint
VulFrame [70]	NA	NA	NA	✗	Unranked
Khanh [71]	NA	NA	NA	✗	A*
DPAM [72]	NA	NA	NA	✗	B
Fang [73]	NA	NA	NA	✗	Unranked
SCDAN [74]	NA	NA	NA	✗	B
MDSAE-NR [75]	NA	NA	NA	✗	Unranked
DeepVL [76]	NA	NA	NA	✗	Unranked
MIPSVul [77]	NA	NA	NA	✗	Unranked
Vulcan [78]	NA	NA	NA	✗	Unranked
Tanwar [79]	NA	NA	NA	✗	Preprint
Pang [80]	NA	NA	NA	✗	Unranked
Jabeen [81]	NA	NA	NA	✗	Preprint
Catal [82]	NA	NA	NA	✗	Preprint
Junae [83]	NA	NA	NA	✗	C
WebVul [84]	NA	NA	NA	✗	A
ALF [85]	NA	NA	NA	✗	A*
DLFV [86]	NA	NA	NA	✗	B
BinaryHan [87]	NA	NA	NA	✗	Unranked
AVDHRAM [88]	NA	NA	NA	✗	C
Explainable [89]	NA	NA	NA	✗	A*
FTCLNet [90]	NA	NA	NA	✗	B
Filus [91]	NA	NA	NA	✗	Unranked

Tool Availability:
Green: Available Red: Unavailable

Tool Link Presentation:
NA: None RT: Raw Text RF: Reference

Tool Link Location:
NA: None MI: Main Text of Introduction MM: Main Text of Methodology
ME: Main Text of Experiment FI: Footnote of Introduction FE: Footnote of Experiments
SI: Searching on the Internet AM: Authors' Email

Tool Hosting Site:
NA: None GH: Github GL: GitLab

that the tool was developed by the authors), and obtained two more available tool links by emailing the authors (VulDeePecker [39] and BiVuld [41]). Note that the authors of VulDeePecker [39] provided the link to SySeVR [40] and told that SySeVR was an extension of VulDeePecker (i.e., the experiments of VulDeePecker could be done by using parts of the SySeVR source code and datasets). Relevant

issues for VulDeePecker and SySeVR will be discussed in RQ2. The varied tool link locations indicate that readers have to review different paper sections or use different ways to find the tool links in different studies. This costs time and limits the access of all levels of people in the society. This variation leads to unnecessary time costs and limits potentially broader tool access.

On the other hand, to allow for convenient public access, many researchers upload their works to public hosting sites. In our study, we noticed that the authors who made their tools publicly available did the same thing. We found that all the available tools were shared on GitHub [5], except for one (Project Achilles [50]), which was shared on GitLab [6]. A possible reason is that GitHub is the most popular hosting site for open source code. These authors may use it as a social media and collaboration platform to promote their research in the community [92].

Failure cases and causes. Meanwhile, we noticed that most of the papers/studies that did not provide publicly available tools provided nothing about their source code. VulSeeker-Pro [51] provided a Github tool link in the paper, but the link expired and thus it was not accessible. After getting contacted by us, a few authors told us the reasons for not sharing the tools: (1) the developed techniques were used to apply for patents, (2) the developed techniques were modified for further development, and (3) the source code and datasets were lost due to some incidents.

> *Only 25.5% of the considered studies provided publicly available tools for their proposed techniques, indicating poor open-science practice at least in this studied area.* Even for the available ones, the tool link locations varied largely, limiting broad and easy access by the general public. Most of these available tools were hosted on GitHub likely for its popularity. The reasons that the majority of the studied tools were not publicly available included artifact loss and reservation for other future purposes.

4.2 RQ2. Executability

As we mentioned earlier, for RQ2 through RQ4, we only considered the tools targeting C/C++ source code or compiled C/C++ binary code. Thus, we excluded TAP [48], VulHunter [49], and Project Achilles [50] because their targeting languages are PHP or Java. We focused on the remaining 11 tools to avoid further complicating our study with potential effects of programming languages on open science practice.

Overall status/statistics. Table 3 shows the executability results of the 11 tools. Similar to RQ1, the color of each row indicates whether the tool was executable, and the four dimensions of the tool are indicated by different abbreviations.

After trying our best, we still could not execute three of the 11 tools. That is, 27.3% of the available tools were not executable. This indicates that executability issues are an extant threat to open science, at least in the area of DL-based vulnerability detection.

Success cases/possible limitations. We inspected the eight executable tools based on the four dimensions we discussed

TABLE 3: Results on tool executability

Study	Tool Completeness	Tool Functioning	Deep Learning Framework	Documentation Quality
TRL [31]	✓	✓	TF	IS
RLMD [32]	✓	✓	TF	IS
LAVDNN [33]	✓	✓	CN	SF
ZMVP [34]	✓	✓	WK	SF
ReVeal [35]	✓	✓	PT	SF
DCKM [36]	✓	✓	TF	SF
MDSeqVAE [37]	✓	✓	TF	IS
SaBabi [38]	✓	✓	TF	SF
VulDeePecker [39]	✓	✗	TF	NA
SySeVR [40]	✓	✗	TF	IS
BiVuld [41]	✗	✗	TF	NA

Tool Executability:
Green: Executable Red: Not Executable

Deep Learning Framework:
TF: Tensorflow PT: PyTorch CN: CNTK WK: Weka

Documentation Quality:
NA: None IS: Insufficient SF: Sufficient

before. We noticed that all the eight tools were complete—they provided all the necessary tool components for us to execute the entire experiments step by step. Besides, the components provided in the tools were all functional. They were executed without crashing so that we obtained valid experiment results smoothly.

Documentation is an important part of a tool. Sufficient documentation helps the users install, configure, and execute a tool smoothly and correctly. We found that among the eight executable tools, five provided sufficient documentation, which included instructions that allowed us to smoothly set up and run the experiments. As an example, Figure 7 shows the documentation of DCKM [36] which was sufficient for us to execute the tool. The documentation described the statistics and format of its dataset and discussed its model implementation. It also included guidelines for users to prepare the execution environment, configure the model parameters, and run model training and testing.

However, the other three tools did not provide sufficient documentation. TRL [31] and RLMD [32] did not provide the detailed steps for executing the experiments. MDSeqVAE [37] did not specify which DL framework and other dependencies should be installed before executing the tool.

The insufficient documentation implies that other researchers have to spend extra time on inspecting the source code to understand how the tool works, so that they can execute the whole experiment correctly. It is also possible that an amateur cannot execute the experiment correctly because of his/her lack of technical knowledge. Thus, insufficient documentation limits the research accessibility to the public, and becomes a threat to open science in the area of DL-based software vulnerability detection.

Meanwhile, we found that all the tools were developed on top of an existing DL framework. Yet the DL frameworks used by the eight executable tools varied. Five tools used Tensorflow[7], while the other three tools used CNTK[8], Weka[9], and PyTorch[10], respectively. The variety of underlying DL frameworks make the run-time environments more difficult to set up than otherwise. Other researchers have to configure different run-time environments for different tools, potentially incurring extra time and storage costs.

5. https://github.com/
6. https://gitlab.com
7. https://www.tensorflow.org/
8. https://github.com/microsoft/CNTK
9. https://www.cs.waikato.ac.nz/ml/weka/
10. https://pytorch.org/

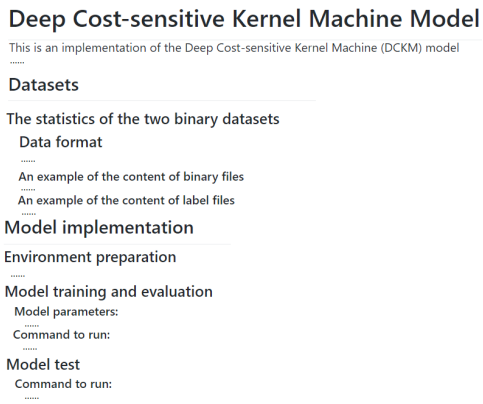

Fig. 7: An example of executability success: the tool (DCKM [36]) provided sufficient documentation including instructions easy for users to follow. Only the headings are shown to save space.

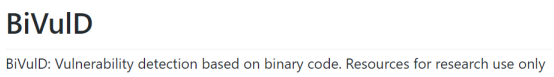

Fig. 8: An example of executability failure: the tool (BiVuld [41]) did not provide meaningful documentation in its repository.

Failure cases and causes. Looking at the three tools that were not executable, we noticed that VulDeePecker [39] and SySeVR [40] were not functional. As we mentioned for RQ1, SySeVR was an extension of VulDeePecker and the experiments of VulDeePecker could be done by using the source code and datasets of SySeVR as informed by the authors. Thus, we discuss them together as one tool.

Based on the papers and the documentation, VulDeePecker/SySeVR has three steps for its experiments. It first generates program slices from the given source code. Then, it preprocesses the program slices to get their vector representations. Finally, the vector representations are used to train and test the DL models.

We noticed that, while providing the code for generating the program slices from the source code, VulDeePecker/SySeVR provided several files in the repository which seemed to be the generated program slices (the documentation did not tell the purpose of these files). We first tried to use these files to directly get the vector representations, but VulDeePecker/SySeVR did not produce any valid outputs as such. Then, we generated the program slices by ourselves. Although the newly generated program slices were different from the ones in the repository, VulDeePecker/SySeVR still did not output any valid vector representations. This suggested possible inconsistency between different components and datasets of the tool, making the tool not functional.

We also noticed that the tool BiVuld [41] was not complete. Based on the paper, BiVuld first generates vector representations from the binary code files, which were compiled C/C++ object files. Then, it uses the vector representations to train and test the DL models. However, the repository of BiVuld only provided the compiled C/C++ object files and the code for training/testing the DL models. We did not find the generated vector representations or the code for generating vector representations. Thus, the tool BiVuld had missing components (i.e., it was incomplete).

In addition, none of the tools that were not executable provided sufficient documentation. The documentation of SySeVR only briefly discussed the functionality of each code file in the repository. The authors of VulDeePecker told us to use parts of SySeVR source code and datasets for the experiments of VulDeePecker, but we did not find any documentation related to VulDeePecker in the SySeVR documentation. BiVuld did not even provide meaningful documentation, as Figure 8 shows.

Considering all the 11 available tools we investigated, 54.5% did not provide sufficient documentation. This indicates very poor overall documentation quality in the studied area. A possible reason was that these authors might not have a strong intent to support others in using their tools. It is likely that they just uploaded the tools to hosting sites as backups of their development.

> *27.3% of the available tools were not executable, compromising open science in the area of DL-based software vulnerability detection.* All the executable tools were complete and functional, but the insufficient documentation and variations in the underlying DL frameworks that have to be set up with considerable effort limited the research accessibility to all. The tools that were not executable were either incomplete or having components that were nonfunctional, and their documentation was insufficient or missing.

4.3 RQ3. Reproducibility

Overall status/statistics. Figure 9 shows the reproducibility statistics of the eight executable tools. For each tool, we compare the F1 score reported in the original paper (the blue bar) with the one we obtained in our reproducibility experiment (the orange bar). The deviation values are shown above the bars, and its color indicates whether the tool was *reproducible* (blue for *strongly reproducible*, orange for *weakly reproducible*, red for *not reproducible*). The results on the four reproducibility characterization dimensions are shown by different shapes (square, diamond, circle, and triangle) under the chart, and their values are indicated by different colors (gray for *no* and green for *yes*).

We noticed that six of these tools were *strongly reproducible*—the absolute values of their deviations were all less than 1%. RLMD [32] was *weakly reproducible* with a deviation -4.31%. Only for TRL [31] we failed largely in reproducing the original experiment results: our F1 score was 26.71% lower than the one reported in the original paper. As a result, 87.5% of the executable tools were considered reproducible (strongly or weakly). This indicates that for the studied tools the gap between executability and reproducibility was overall quite small.

Success cases/possible limitations. We inspected the seven reproducible tools along the four dimensions we discussed before. We first checked whether the datasets used by the tools were consistent with the ones described in the papers. We found this inconsistency in RLMD [32]. The authors had updated the dataset by adding new program samples (the number of program samples in the dataset described in the paper was 33,822, while that in the dataset provided in the repository was 39,942). This was a plausible reason

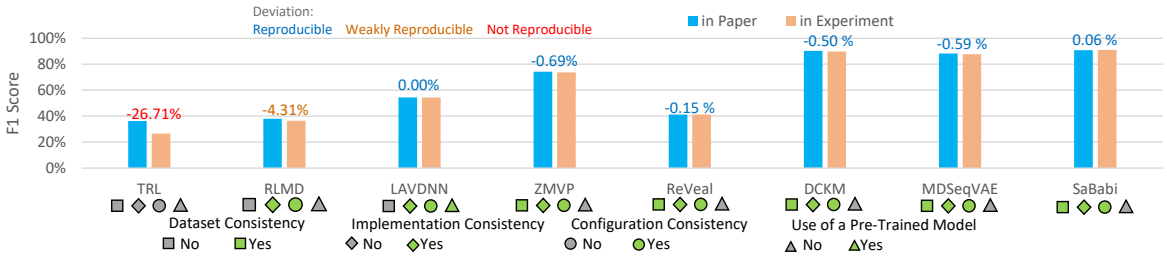

Fig. 9: Results on tool reproducibility.

Generate two datasets, a training set and test set. The `SA_SEED` variable is optional, but used here for reproducibility.

```
$ SA_SEED=0 ./sa_e2e.sh working/sa-train-1000 1000
$ SA_SEED=1 ./sa_e2e.sh working/sa-test-100 100
```

Fig. 10: An example of dataset consistency: the tool (SaBabi [38]) uses *fixed random seeds* for consistent formation of training/testing sets.

Fig. 11: Another example of dataset consistency: the tool (ReVeal [35]) uses *pre-split training/validation/testing sets* to ensure the consistency.

that RLMD was only *weakly reproducible*. LAVDNN [33] did not provide its complete training set. Its success in reproducibility was based on its pre-trained model, which will be discussed later.

For the remaining five reproducible tools, the dataset used was strictly consistent with the description in paper. These tools not only ensured the consistency for the entire dataset, they also ensured the consistency for both training and testing sets. This was either achieved by using *fixed* random seeds (numbers or vectors which make a random process behave the same in each run) to split the dataset (exemplified in Figures 10) or by simply using *pre-split* training/testing sets (exemplified in Figure 11). That is, the samples we used for training and testing were completely the same as the ones used in the authors' experiments.

We also checked the implementation and configuration of each tool. By checking the source code against the descriptions in the paper, we did not find any inconsistency in either regard for any of the seven reproducible tools. Put together, 71.4% of the reproducible tools had datasets, implementation, and configurations all being consistent with the ones described in the original papers. These consistencies contributed to the reproducibility successes.

Combining with the findings for RQ2 revealed that all the executable tools that provided *sufficient* documentation (LAVDNN [33], ZMVP [34], ReVeal [35], DCKM [36], and SaBabi [38]) were *strongly reproducible*. In contrast to the tools providing insufficient or entirely missing documentation, the authors of these tools may have paid particular attention to reproducibility, purposely supporting open science.

For LAVDNN [33], the F1 score obtained in our experiment was exactly the same as the one reported in the paper. The reason was that it provided a pre-trained DL model, which allowed us to run the tool on the testing samples directly. Because of the randomness in DL model initialization and optimization, it is common that the models trained on the same dataset are different, and thus they report different accuracy [46]. In this case, a pre-trained model is a simple way to eliminate the randomness hence facilitate successful reproduction. It also saves time and computational resources that would have been spent to re-train the DL model.

However, we noticed that LAVDNN did not provide a complete training set. Because of this, we were not able to re-train the model to evaluate the training process. The incompleteness also restricted our replicability evaluation in RQ4. Therefore, it is recommended that authors provide the training set even if a pre-trained model is provided.

Failure cases and causes. We found that TRL [31] was not reproducible, with a -26.71% deviation. TRL (Transferable Representation Learning) is a technique that aims to solve the problem that high-quality vulnerability detection training data is in shortage in a single real-world project [31]. It did so by learning rich features from different similar real-world projects. Based on the paper and the documentation, TRL has several steps for its reproduction experiments. At the beginning, the vulnerable/non-vulnerable function samples of 6 real-world projects are given. In the 6 projects, 5 are chosen as historical data for pre-training a DL model, and the remaining one is chosen as the testing project. Then, the pre-trained DL model is used to extract the rich features for the samples in the testing project. Finally, using the rich features, only 25% of the samples in the testing project are used for training another machine learning model (based on the hypothesis that training data is in shortage in a real-world project). The new machine learning model can predict the vulnerabilities in the testing project well although only 25% of the samples are used for training.

We followed the several steps above to reproduce the experiment. According to the experiment configuration in the paper, we chose FFmpeg as the testing project and the other 5 projects were used for pre-training the DL model. However, we did not succeed at reproducing the results after five trials and our best F1 score was still 26.71% lower than one reported in the paper. We then contacted the authors for help. The authors told us that it was normal for getting the model under-performed because the code in the Github repository was not optimized. They recommended us to add implementation of dropout layers and regularizers in the DL model to be pre-trained. They also recommended us to modify the configuration of the DL model, such as tuning the learning rate of the optimizer and reducing the batch size. This indicates that the implementation and the configuration of the DL model were not consistent with the ones originally used by the authors. Furthermore, despite

following their recommendations, the results did not improve or even became worse. A possible reason was that we did not know necessary details for correctly modifying the implementation and configuration.

We also noticed that the dataset provided in the repository was not the same as the one used in the original paper. This might have further enlarged the reproducibility deviation in TRL.

In sum, our analysis of this failure case indicates that, while providing the source code, the tools on the public repositories may not be fully/well developed or fine-tuned, making the tools *not reproducible*.

> *Most (87.5%) of the executable tools were reproducible, strongly supporting open science in the area of DL-based software vulnerability detection.* Some authors provided consistent datasets, implementation, configurations, and sufficient documentation, which greatly facilitate reproduction. Training a DL model has randomness in it and costs time, for which offering a pre-trained model helped achieve perfect reproduction. The tool which was *not reproducible* provided inconsistent dataset, implementation, and/or configuration in its repository, leading to a large deviation in our *reproducibility experiments*.

4.4 RQ4. Replicability

As justified earlier (Section 3.5), we separately examine *partial replicability* and *full replicability*, for which we present the results in RQ4.1 and RQ4.2, respectively.

4.4.1 RQ4.1 Partial Replicability

For the *partial replicability* experiments, we first used each reproducible tool to preprocess the `9_projects` dataset and split it into training and testing sets. For the tools detecting vulnerabilities in C/C++ source code, i.e., RLMD [32], LAVDNN [33], ZMVP [34], ReVeal [35], and SaBabi [38], we used the `source code` version (61,603 samples) of the `9_projects` dataset. For the tools detecting vulnerabilities in C/C++ binaries, i.e., DCKM [36] and MDSeqVAE [37], we used the `binary code` version (20,856 samples). For each tool, we tested the model trained in our reproducibility experiments to gauge its accuracy.

Overall status/statistics. Figure 12 shows the results of the seven replicable tools in our partial and full replicability experiments. Like for RQ3 (Section 4.3), we compare the F1 scores in our experiments (orange bars) with the ones reported in the papers (blue bars), and show the deviations with different colors. For each tool, the deviation for *partial replicability* and *full replicability* is shown to the left and right of the slash, respectively. We also compare the datasets used in our experiments (green bars or texts) with the ones described in the papers (yellow bars or texts) in the four dimensions defined earlier. Note that when we processed and split the `9_projects` dataset, some of the tools only successfully processed a part of the samples (e.g, ZMVP only processed 7,277 samples, although the whole dataset had 61,603 samples, as shown in the parenthesis in the figure). The green bars/texts shown on the figure are for the actual datasets used in our replicability experiments. LAVDNN did not provide its whole dataset, thus the duplication rate for it was missing.

We noticed that the overall *partial replicability* of these reproducible tools was poor. Only RLMD was *weakly reproducible*, with a -4.96% deviation. The deviation of ZMVP was large (-40.90%), although its F1 score itself appeared to be the highest among these tools in our experiments. ReVeal and DCKM had much worse F1 scores, with -78.62% and -79.87% deviations, respectively. LAVDNN and MDSeqVAE performed extremely bad in our experiments, with only 5.70% and 3.56% F1, respectively. SaBabi was not able to process any third-party program samples, thus it was not replicable at all.

Overall, only 14.3% of the *reproducible* tools were *replicable* in our *partial replicability* experiments. The results indicate that many pre-trained DL models from the DL-based software vulnerability detection tools worked poorly or did not work at all when applying to a third-party real-world dataset. This significantly limited these techniques from being applied to real-world software vulnerability detection, threatening open science in this area.

Success cases/possible limitations. We inspected RLMD, the only tool that was *(weakly) replicable* in our *partial replicability experiments*, and found that it was able to process all the 61,603 program samples in the `9_projects` dataset. Both this dataset and the one used in the paper (over 35,000 samples) are large and imbalanced (with less than 3% positive/vulnerable samples), and they both fully consist of programs from real-world projects. Their duplication rates were also both low ($< 10\%$). These similarities were probably the key for the success of this tool in replicability.

Failure cases and causes. We then inspected the six tools that were not *partially replicable*. We observed a few patterns of the failure and discovered several underlying causes.

First, we noticed that the original dataset used by several of the tools was small in size and the samples were not diverse. The dataset of MDSeqVAE only had 14,304 samples. ReVeal only used 18,169 samples from two open-source projects to construct its dataset [35]. LAVDNN built its dataset with only 18,925 samples and 2,400 of them were augmented to avoid overfitting [33]. With datasets of low diversity and such relatively small sizes (for deep learning), the models were not very well trained, which significantly limited their capabilities to generalize to other datasets.

Second, some of the tools originally only used mostly or entirely artificially generated program samples to train their models. In particular, all the samples used by MDSeqVAE and SaBabi were artificially generated, while ZMVP used a mixed dataset where more than 85% of the samples were artificial. In contrast, the samples we used for testing the models are all from real-world software projects. We observed that these real-world program samples are much more complex than, hence unlikely well represented by, the artificially generated ones. Thus, the DL models, mostly only seeing patterns in artificial samples, were not able to sufficiently learn realistic vulnerability patterns [35] hence unable to work well with real-world programs.

Figure 13 shows a comparison of a real-world vulnerability sample and an artificially generated one. In the artificial sample, the vulnerability is conspicuous at line 6, where the

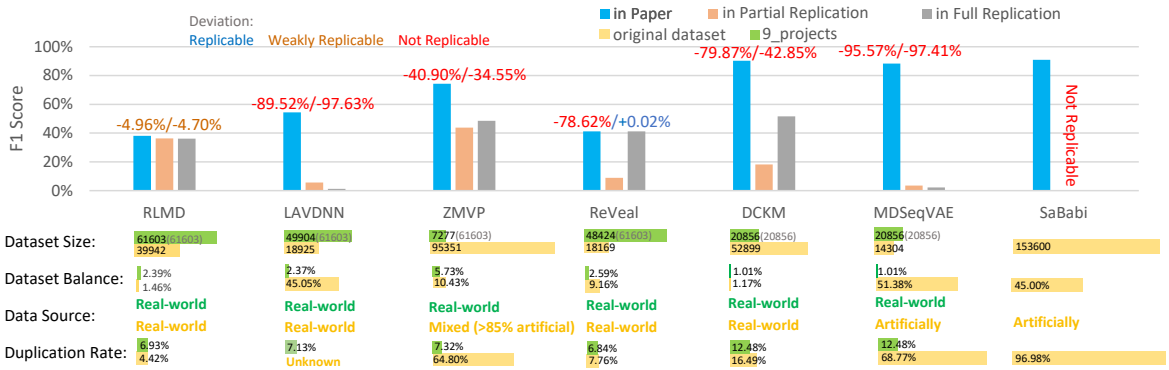

Fig. 12: Results on tool replicability.

buffer write (to buf+17) blatantly reaches beyond the buffer boundary (buf+9). However, in the real-world sample, the vulnerability is not even close as easy to detect—There are many function calls (lines 9 and 14) and other complex code contexts (lines 3-11) involved in the vulnerable operation (at line 14). While this example just illustrates one case of how much more complex real-world programs are than artificial ones, such cases were common between our dataset (`9_projects`) and (most or all of) the samples used by MDSeqVAE, SaBabi, and ZMVP. Apparently, it is quite challenging for models trained on these simple programs to detect vulnerabilities in complex real-world programs.

We also noticed that SaBabi was not able to process third-party samples, because it requires special information that was provided in its own original dataset but not commonly available in other datasets (including `9_projects`). Figure 14 shows a program sample originally used by SaBabi. The comment (the text after "///") in each line provided such information that SaBabi relies on in order to train and test its model. SaBabi did not provide any tool to generate such information for a given third-party sample either. As a result, it was *not replicable* at all in our experiments.

Yet another cause, which we found contributed to the partial replicability failure with MDSeqVAE and DCKM, concerns the data the model took as immediate inputs. Both of these two tools target C/C++ binaries, unlike the other tools detecting vulnerabilities in source code. It is well known that extracting semantic information (even very basic ones like control flow [93]) from binary code is much harder than from source code. Yet a DL model needs to learn such information in order to make correct predictions of a given program as vulnerable or not. As a result, it was even more difficult for the DL models in MDSeqVAE and DCKM, trained on one set of binaries, to generalize its prediction capabilities to a quite different set of binaries.

> *Only 14.1% of the reproducible tools were replicable in our partial replicability experiments, indicating a large gap in this regard of open science in the studied area of DL-based vulnerability detection.* Main causes underlying the gaps included limited sizes and diversity of datasets used in original evaluation, challenging model input format that is intrinsically hard to learn from, largely/entirely using artificial (rather than real-world) samples, and requirement for special information in both training and testing samples.

Artificially Generated Vulnerability Sample

```
1 int main(int argc, char *argv[])
2 {
3     char buf[10];
4     // Write to a postion that is outside the buffer
5     // Causing a buffer overflow vulnerability (CWE119)
6     buf[17] = 'A';
7     return 0;
8 }
```

Real-World Vulnerability Sample

```
1 int new_msg_register_event(u_int32_t seqnum, struct lsa_filter_type *filter)
2 {
3     u_char buf[OSPF_API_MAX_MSG_SIZE];
4     struct msg_register_event *emsg;
5     int len;
6     emsg = (struct msg_register_event *)buf;
7     len = sizeof(struct msg_register_event)
8           + filter->num_areas * sizeof(struct in_addr);
9     emsg->filter.typemask = htons(filter->typemask);
10    emsg->filter.origin = filter->origin;
11    emsg->filter.num_areas = filter->num_areas;
12    // "len" may be greater than the buffer size of emsg,
13    // Causing a buffer overflow vulnerability (CWE119)
14    return msg_new(MSG_REGISTER_EVENT, emsg, seqnum, len);
15 }
```

Fig. 13: An example of the greater complexity of real-world programs we used than the ones originally used by some of the studied tools.

```
1  #include <stdlib.h>              // Tag.OTHER
2  int main()                       // Tag.OTHER
3  {                                // Tag.OTHER
4      int entity_6;                // Tag.BODY
5      int entity_2;                // Tag.BODY
6      char entity_5[48];           // Tag.BODY
7      int entity_0;                // Tag.BODY
8      entity_0 = rand();           // Tag.BODY
9      entity_2 = 32;               // Tag.BODY
10     entity_6 = 72;               // Tag.BODY
11     if (entity_0 < entity_2){    // Tag.BODY
12     } else {                     // Tag.BODY
13     entity_0 = 10;               // Tag.BODY
14     }                            // Tag.BODY
15     while(entity_6 < entity_0){  // Tag.BODY
16     entity_6++;                  // Tag.BODY
17     }                            // Tag.BODY
18     entity_5[entity_6] = 'w';    // Tag.BUFWRITE_COND_UNSAFE
19     return 0;                    // Tag.BODY
20 }                                // Tag.OTHER
```

Fig. 14: An example showing the special, additional information required by a tool (SaBabi [38]) for its model training and testing.

4.4.2 RQ4.2 Full Replicability

As mentioned earlier, in the *full replicability* experiments, for each tool, we re-trained the model using the split training set of `9_projects` and tested the re-trained model on the split testing set.

Overall status/statistics. We show in Figure 12 the full replication results of the seven reproducible tools. We compare the F1 scores in our full replicability experiments (gray bars) with the ones reported in the papers (blue bars).

We noticed that the overall *full replicability* of these

tools was poor. Only ReVeal was *replicable*, with a +0.02% deviation, and RLMD was *weakly replicable*, with a -4.70% deviation. The deviations seen by ZMVP and DCKM were large (-34.55% and -42.85%, respectively), yet they still achieved fine F1 scores in absolute terms (48.57% and 51.61%, respectively). LAVDNN and MDSeqVAE, however, performed extremely poor in our experiments, with a very low F1 score of 1.29% and 2.29%, respectively.

Overall, only 28.6% of the *reproducible* tools were *replicable* in our *full replicability* experiments. While slightly (just one more tool achieving success) better than those in our *partial replicability* experiments, this number still indicates an undesirable open-science situation. That is, these DL-based software vulnerability detectors can still not perform well even when their models are completely retrained (i.e., when the testing data is more likely to be closer to the training set, compared to using pre-trained models). The implication of these results is that the *generalizability* of these tools is not promising overall.

Success cases/possible limitations. We inspected RLMD and ReVeal, which succeeded in our *full replicability experiments*. RLMD achieved similar performance compared to the one in our *partial replicability experiments*. The reasons were the same as the ones we discussed in Section 4.4.1 (i.e., similarities between our dataset and the originally used one). Regarding ReVeal, while it was *not partially reproducible*, because of its use of small and non-diverse dataset for training the model, its unbalanced, realistic dataset still represented the real-world vulnerability detection scenario, which made its original performance evaluation fair.

Importantly, in our partial replication, the testing samples were from a different dataset compared to the training set, while in the full replication both the training and testing sets were from the same dataset (`9_projects`). The fact that ReVeal succeeded in the full replication while failing in the partial replication implies that its model was well generalized to the new dataset (after getting retrained on this dataset), and this new dataset was quite different from the original training set used in the paper.

Thus, while facilitating reproduction, using a pre-trained model offered by a tool could make partial replication more challenging than full replication as the former additionally requires the replication testing set to be similar to the dataset used for the pre-training—otherwise, the testing samples would be out of distribution for the (pre-trained) model.

Failure cases and causes. We examined the five tools that were *not replicable* in our *full replicability experiments*. We identified several causes/patterns of the replication failures.

First, some of the replicable tools could not process third-party samples, or could only process a part of third-party samples. For example, SaBabi only worked on its own dataset as we discussed in Section 4.4.1, thus no full replicability could be evaluated. ZMVP crashed when it processed some program samples in the `9_projects` dataset, thus only a small proportion (7,277) of the (61,603) samples were actually used for training and testing in our replicability experiment. LAVDNN and ReVeal did not process all the samples either. While the inability to process all the samples might not impact the tool's core technical performance, it was still a threat to open science, because the samples that can not be processed would not be usable in the reproduction/replication experiments, immediately limiting the scale of such experiments.

Second, some of the tools originally used balanced datasets to evaluate their performance. However, real-world datasets for software vulnerability detection are often extremely imbalanced, making the DL model biased if the tool does not purposely handle the imbalance issue [35]. We noticed that the tools using balanced datasets (those having data balance around 50%) had much larger performance deviations. Specifically, LAVDNN and MDSeqVAE, which used highly balanced datasets to evaluate performance, had extremely large deviations in our *full replicability* experiments (-97.63% and -97.41%, respectively). This indicates that the authors of LAVDNN and MDSeqVAE might have not considered data imbalance issues in the real world carefully when developing and evaluating their tools. In consequence, their tools were not able to deal with imbalanced datasets, making them *not replicable* against such datasets (like the one we used).

Third, as we discussed in Section 4.4.1, some of these five tools only used artificially generated program samples to evaluate the performance. Not only were such artificial samples too simple to train a model sufficiently for real-world tasks, they also led to inflation in the performance measurement. We noticed that the artificially generated datasets introduced many duplicate samples: the datasets of ZMVP, MDSeqVAE, and SaBabi had significant portions (64.80%, 68.77%, and 96.98%, respectively) of duplicate samples. Figure 15 delineates the impact of duplication on our replicability results. It shows the fitting curve, as well as the coefficient of determination R^2 which indicates how close the data fits to the curve. While the correlation was not strongly linear ($R^2 < 0.8$) because of the impact of other factors (e.g., dataset size, data balance), overall the tools using datasets with higher duplication rates tended to have larger performance deviations. The uses of artificially generated samples, as well as the data duplication issues, made the DL model learn irrelevant features. Due to these issues, the overall original performance of the tools [94] was inflated, contributing to the *full replicability* failures.

> *Only 28.6% of the reproducible tools were replicable in our full replicability experiments, indicating poor overall status of open science in this aspect—despite the slightly better situation than for partial replicability.* For the two replicable tools, the success was attributed to better model generalizability and/or the similarities of the replication dataset to the one used in original evaluation. For the other tools, the failures can be explained by issues with data preprocessing and/or the use of original evaluation dataset that was highly duplicated or highly balanced.

4.4.3 Case Studies

Our results on the full replicability of the studied tools suggested the impact of various dataset dimensions. To further understand these impacts, we conducted two additional case studies to quantify them, focusing on two major dimensions: dataset size and balance, respectively.

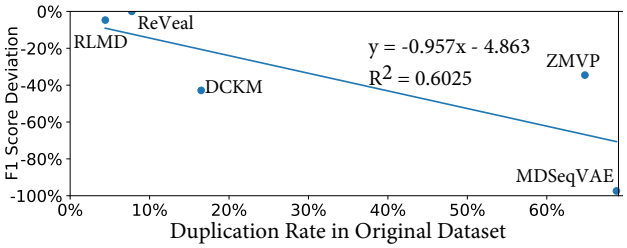

Fig. 15: Impact of dataset duplication on F1 in full replication.

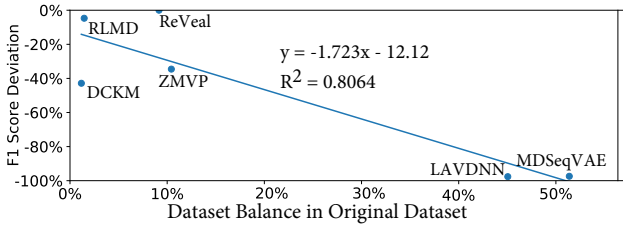

Fig. 16: Impact of dataset balance on F1 in full replication.

Case study 1: impact of dataset size. We first investigated how the dataset size impacts replicability. We reduced the dataset sizes and used the reduced datasets to repeat our *full replicability* experiments as discussed in Section 3.5.2. Table 4 lists the F1 scores of each tool under different size reduction ratios. We noticed that many of the tools' performance decreased when we reduced the entire dataset. For instance, when the size was reduced to 10% of the original, the F1 score of dropped from 36.29% to 28.40% for RLMD, from 1.29% dropped to 0% for LAVDNN, and from 51.61% to 4.65% for DCKM. The main reason for these performance drops was that these tools suffered from greater overfitting when their models were trained on fewer samples, as found earlier in [95]. In consequence, the DL models tended to learn irrelevant features to fit the smaller training set, causing the models to perform worse when applied to the testing samples. Note that MDSeqVAE and LAVDNN barely worked in our replication experiments, and we found that their predictions were essentially random, as seen from their F1 score triviality/variations.

In comparison, ReVeal and ZMVP were much more stable. Their performance stayed almost unchanged with various dataset reductions. We found that this stability was mainly due to the more effective (semantic) program representations learned by these tools. Specifically, ReVeal constructs a code property graph (CPG) for each program sample, which contains both syntactic (from abstract syntax tree) and semantic (from control/data flow) information [35]. It took the CPGs as inputs to a graph-based DL model (gated graph neural network) for vulnerability detection.

A similar merit was found in ZMVP as well. It extracts the code slices that were related to vulnerabilities from each program sample, and computed 19 code metrics to quantitatively characterize the vulnerability-relevant code [34]. These code metrics were then fed into to a multilayer perceptron (MLP) model for vulnerability detection.

These semantic representations learned by the DL models captured program behaviors that were actually responsible for vulnerabilities, making these two tools much less sensitive to the number of training samples used. In contrast, the other four tools simply treated a program as a natural-language text (i.e., a sequence of tokens), which did not capture semantic information of the code. As a result, the representation learned by these tools did not model vulnerability-relevant code behaviors. That is, the models mostly ended up learning irrelevant features.

TABLE 4: F1 scores of each tool when changing the dataset size

Dataset Size	100%	70%	40%	10%
RLMD	36.29%	36.76%	38.71%	28.40%
LAVDNN	1.29%	1.28%	1.26%	0.00%
ZMVP	48.57%	48.39%	48.26%	47.26%
ReVeal	41.26%	41.07%	40.96%	40.00%
DCKM	51.61%	42.11%	33.33%	4.65%
MDSeqVAE	2.29%	4.17%	7.69%	3.03%

TABLE 5: F1 scores of each tool when changing the dataset balance

Dataset Balance	$1 \times b$	$4 \times b$	$7 \times b$	$10 \times b$
RLMD	28.40%	31.78%	45.79%	55.25%
LAVDNN	0.00%	7.11%	8.70%	7.43%
ZMVP	47.26%	48.25%	48.55%	48.40%
ReVeal	40.00%	41.07%	40.96%	40.00%
DCKM	4.65%	25.00%	51.85%	42.86%
MDSeqVAE	3.03%	12.28%	17.14%	28.88%

> *Overall, dataset size had significant impacts on the performance of the tools.* Using large enough datasets was crucial to training DL models adequately and avoiding overfitting, hence to better replicability. On the other hand, learning semantic data (program) representations largely mitigated model sensitivity to dataset size, also making the tools more replicable.

Case study 2: impact of dataset balance. In a similar way of exploring the relationship between dataset duplication rate and tool performance (F1) as shown in Figure 15, we also attempted to discover the correlation between dataset balance ratio and tool performance (also in terms of F1) through curve fitting. As plotted in Figure 16, based on the performance results of the six tools in our *full replicability* experiments (y axis) and their original balance ratios (x), we observed a reasonably linear ($R^2 > 0.8$) relationship between the two variables. Overall, the tools using balanced (a balance ratio of around 50%) datasets had much larger performance deviations, as we discussed in Section 4.4.2.

To further examine the impacts of dataset balance, we did extended experiments by changing the dataset balance of the dataset while keeping the total dataset size unchanged, as we discussed in Section 3.5.2. Table 5 shows the F1 score of each tool with each different balance ratio we considered. We noticed that the performance of LAVDNN and MDSeqVAE, which originally used balanced datasets to evaluate their performance, was significantly impacted by the dataset balance variations. After making the dataset more balanced, the F1 score of LAVDNN improved from 0.00% (i.e., not working at all) to 8.70% and the F1 score of MDSeqVAE improved from 3.03% (i.e., barely working) to 28.88%. These F1 scores were even much higher than the results in our *full replicability* experiments (where we used the entire usable datasets but with the original balance ratios). These noticeable performance variations can be explained by the fact that LAVDNN and MDSeqVAE did not adopt

TABLE 6: Association between venue ranking and open science

Open Science	Venue Rank	Support	Confidence	Lift
Availability	A*	7.27%	36.36%	1.43
	A	3.64%	25.00%	0.98
	B	1.82%	17.00%	0.65
	C	N.A	N.A	N.A
	Unranked	10.91%	30.00%	1.18
	Preprint	1.82%	12.50%	0.49
Executability	A*	18.18%	66.67%	0.92
	A	9.09%	50.00%	0.69
	Unranked	36.36%	80.00%	1.10
	Preprint	9.09%	100.00%	1.38
Reproducibility	A*	25.00%	100.00%	1.14
	A	12.50%	100.00%	1.14
	Unranked	37.50%	75.00%	0.86
	Preprint	12.50%	100.00%	1.14
Replicability	A*	14.29%	50.00%	1.75
	A	14.29%	100.00%	3.5
	Unranked	N.A	N.A	N.A
	Preprint	N.A	N.A	N.A

any mechanisms to deal with dataset imbalance in their designs [33], [37], making them sensitive to dataset balance.

We observed that ZMVP and ReVeal, which originally used imbalanced datasets to evaluate performance, were stable against dataset balance changes. This was because they purposely handled imbalanced datasets—ZMVP implemented an undersampling mechanism called SpreadSubsample unsupervised filter [34] while ReVeal adopted an oversampling mechanism called SMOTE [35]. These dedicated data handling techniques led to greater model stability hence better tool replicability, given that the balance of a replication dataset is very likely to diff from that of the dataset used in the original tool evaluation.

While RLMD and DCKM used imbalanced datasets to evaluate performance as well, they were still sensitive to dataset balance. Specifically, the F1 score of RLMD improved from 28.40% to 55.25% and that of DCKM jumped from 4.65% to 51.85%. We noticed that both RLMD and DCKM employed cost-sensitive learning to deal with imbalanced datasets. Thus, incorporating a dedicated handling mechanism, while helpful, did not constitute a guarantee for model stability against dataset balance variations. Using datasets whose balance represents real-world scenarios in tool evaluation is still necessary for successful replication by others using real-world datasets.

> *Overall, dataset balance had significant impacts on the performance of the tools—most of them were sensitive to balance variations.* Adopting dedicated imbalance handling strategies helped alleviate, but did not always eliminate, the sensitivity. In all, evaluating tools against datasets who balance ratios represent real-world scenarios could help discover gaps early on hence achieve better replicability later.

5 Discussion

We first present and discuss additional observations beyond those for answering the four RQs. Then, we discuss the implications of our major findings and make actionable suggestions accordingly based on those findings, to support open science not only in the studied area of DL-based software vulnerability detection but beyond.

5.1 Additional Observations

Association with publication venues. Intuitively, the venue (regarding its visibility, reputation, etc.) in which a paper is published may affect the open-science status of the paper. For instance, it is plausible that a paper published at a more visible/reputable (e.g., in terms of ranking) venue may look better in various open-science dimensions. Thus, we classified the 55 papers based on the ranking of their published venues. We used the *CORE Rankings* [11] as the reference, which classifies a journal or conference into one of four main classes (ranks), A*, A, B, C, in the order of decreasing visibility/reputation/prestige. For venues that are not listed in this ranking, we marked them as *Unranked*. For those papers only posted on preprint websites (e.g., arXiv) and not yet accepted by any peer-reviewed venues, we marked them as *Preprint*. Table 2 (last column) lists these ranks for the studied papers. Since we only evaluated executability, reproducibility and replicability on the tools that passed the previous dimensions, we only computed the association on the studies evaluated in the respective dimensions (e.g., we only computed association between executability and venue ranking on the 11 tools listed in Table 3). Table 6 shows the frequent *if-then* associations between the venue ranking and (*success*) status in each of four open-science dimensions, using the Apriori algorithm [96]. The *support* indicates how frequently a pair of venue rank and the open science status appears among the relevant papers. The *confidence* indicates the conditional probability of the occurrence of the open science status given the paper's venue rank. The *lift* indicates the strength of association between the paper's venue rank and the open science status: $lift<1$ indicates the paper venue rank and the open science status are mutually exclusive; $lift==1$ indicates no association; and $lift>1$ indicates that they are associated, with greater *lift* indicating stronger association. When there is not any paper at a rank (e.g., C) that has the success status for an open-science dimension (e.g., *Availability*), there are no applicable (*N.A.*) association results for that dimension.

Overall, our results show no consistent/significant association between open-science status and publication venue ranking. The top-venue (A*) papers tended to have better *availability*: the lift value is 1.43, greater than those with venues of any other ranks. However, the top-venue papers were found exclusive with *executability* as the lift is less than 1. For *reproducibility* and *replicability*, the association strengths with top-venue papers were not greater at all compared to the strengths with venues in other ranks. This indicates that the fact that the papers were published on top venues did not necessarily indicate the open science status of those papers, although the top venues usually enforce open science policies. In contrast, while the lower-ranked venues do not usually enforce open science policies, many papers in there still practiced open science. This indicates that the open science policies for the top venues may need to be improved. Specifically, current top SE venues like ICSE [12] and FSE [13] require the authors to upload their artifact packages when they submit the papers unless they

11. https://www.core.edu.au/
12. http://www.icse-conferences.org/
13. https://www.esec-fse.org/

can explain the reasons not to do so. This is a possible reason why the availability of the top-venue papers is better than others. However, while artifact sharing is required, there are no detailed requirements for the artifact package itself. We noticed that some authors only uploaded the data used without providing the source code to use it. Some authors did not provide complete source code. Other authors did not provide sufficient documentation for other researchers to use them. These are the possible reasons that the *executability*, *reproducibility*, and *replicability* of the top-venue papers are not better than others.

Effectiveness of existing techniques. In Section 4.4, by applying the existing DL-based vulnerability detection techniques to a real-world third-party dataset, we found that none of the techniques had reasonably high accuracy. DCKM and ZMVP achieved the best F1 scores (52% and 48%, respectively) in our replicability experiments, most likely not sufficient for practice use. This also implies that the actual overall effectiveness of current DL-based software vulnerability detection techniques might have been highly overrated. Although evaluating/comparing the performance of these techniques was not the aim of our study, our results did reveal that existing techniques still lack practicality.

Comparison with R&R of traditional vulnerability detectors. In [97], [98], we performed a comparative evaluation on five memory error vulnerability detectors that are based on code analysis, a major traditional approach to vulnerability detection. In that study, we used the `MemSafety` and `Systems_BusyBox_MemSafety` datasets from the SV-COMP 2019 competition [99] to evaluate the five detectors. One of them, `CBMC` [100], participated in the competition, making our evaluation of it a *reproduction*. In the competition, the overall F1 score of `CBMC` on these two datasets was 97%, but the number in our experiments was only 72%.

Two other evaluated detectors, `DrMemory` [101] and `Valgrind` [102], used different benchmarks (datasets) in their original performance evaluation. Thus, our evaluation of these two tools can be considered *replication*. Compared to the datasets we used, their benchmarks came with no ground truths; thus they were only evaluated in terms of the numbers of vulnerabilities found. As a result, we could not actually measure replicability since our metrics metrics (i.e., recall, precision, F1) were not comparable to theirs.

These previous results initially signified that the contemporary open science status may not be promising in the area of traditional vulnerability detection either.

Comparison with other R&R studies in Software Engineering. There exist other R&R studies showing the state of open science in SE. Robles [23] reviewed the papers in Mining Software Repositories (MSR) that contained experimental analysis. He noticed that out of the 154 papers published between 2004-2009, only two (1.30%) offered their raw data, processed data, as well as complete sets of tools/scripts. Rodríguez-Pérez et al. [19] reviewed 189 papers using the SZZ algorithm, only 24 (12.70%) provided reproduction packages and described the reproduction steps in detail. Daoudi [103] tried to reproduce and replicate five machine learning based Android malware detection tools. They spent much effort but only three tools could be reproduced and only one replicated successfully. Liu et al. [27] reviewed 94 DL-based software engineering papers and only 24 (25.53%) provided accessible links for their replication packages. These prior peer studies indicate that the issues of open science in software engineering are prevalent.

5.2 Open Science in DL-based Vulnerability Detection

Based on our empirical results and the additional observations, we offer insights and recommendations that our community could build upon to support open science in DL-based software vulnerability detection.

Taking care of data (pre)processing. In Section 4.4, we noticed that 57.1% of the reproducible tools had issues with (pre)processing third-party program samples, making it difficult for other researchers to comprehensively evaluate these tools (e.g., evaluation scale may have to be reduced since samples that can not be processed have to be dismissed). It also prevents these tools to be used against real-world software, hence limiting their usefulness in practice. Thus, we suggest that authors make their tools compatible to third-party program samples, so that the developed techniques can be better evaluated and more practically useful. Based on our observations, such better compatibilities can be achieved by either building data processing in the tool core or providing additional data-preprocessing utilities (e.g., scripts), *with respect to real-world usage scenarios*.

Building and using standard baselines/datasets. In Section 4.4, we noticed that only 14.3% and 28.6% of the reproducible tools were *replicable* in the *partial* and *full replicability* experiments, respectively. Put in the context of our holistic study, only 1/55 or 2/55 of the considered techniques were replicable, a serious warning sign for open science in our community. Based on our analyses, beyond those impeding the availability and executability bottom-line, the uses of balanced, artificially generated, or highly-duplicated datasets in the original tool evaluations were the main reasons for this dire outcome. In the 7 papers studied in RQ4, 28.58% (2) of them only used artificially generated datasets. We then went back to RQ1 and further checked the 55 studied papers, and found that 34.55% (19) of them only used artificially generated datasets. Using such datasets led to inflation in the reported performance, hence leading to big gaps with respect to real-world datasets and realistic/practical application settings.

Besides, we also noticed that the baselines/datasets used for evaluating the studied tools are usually different in different papers. The main reason was that there was a lack of standard baselines/datasets for evaluating vulnerability detection techniques. Many of the studies had to build new datasets to evaluate their techniques. For example, the authors of Devign [53] spent around 600 man-hours to collect real-world vulnerability samples to train and test their technique. The authors of ReVeal [35] did the similar thing to collect real-world vulnerability samples from two projects (i.e., Debian and Chromium). For traditional (i.e., code-analysis-based) vulnerability detectors, the tools were not even evaluated on the datasets with ground truths, as we discussed in Section 5.1. In contrast, many other DL or SE domains have standard baselines/datasets for evaluating techniques. For example, in computer vision,

ImageNet [104] is a large-scale image database which is widely used for evaluating image classification techniques. In software defect analysis, researchers usually use standard real-world baselines/datasets like Defect4J [105] and Bugs.Jar [106]. These standard baselines/datasets well represent real-world application scenarios. They are accompanied with ground truths, realistic, and diverse.

Thus, we strongly suggest that the software vulnerability analysis community (including the areas of DL-based and conventional code-based vulnerability detection) may build standard baselines/datasets that well represent real-world application scenarios and explicitly encourage or even require researchers to evaluate their techniques against standard baselines/datasets. Such baselines need to stay at the front line (representing the state of the art) and the datasets need to be large-scale, realistic, and diverse. Doing so would not only help assess the real performance of the techniques comprehensively and fairly; more importantly, it would reveal performance insufficiency early on and help researchers improve their technical designs accordingly, hence ensuring the practical applicability of the tools for end users while facilitating replication by other researchers.

Focusing more on tool practicality. Intuitively, a main purpose of developing DL-based vulnerability detectors is to help detect new vulnerabilities in real-world software systems hence reduce the losses they may cause. However, current tools seemed to be playing a "numbers arms race" in the community—They tend to primarily pursue higher numbers for effectiveness metrics like precision, recall, and F1 on the existing datasets, largely dismissing the criticality (e.g., security impact) of the vulnerabilities (e.g., treating all the vulnerabilities detected equally in terms of their severity). In this case, the practicability of the tools is not evaluated in most of the papers. Indeed, software developers may expect the new tools to detect more *critical* vulnerabilities (e.g., those are severer or harder to detect by other tools). Thus, we checked the 55 papers for whether they found new vulnerabilities and whether they discussed the severity of the vulnerabilities that the tools can detect. Unfortunately, only 7.27% (i.e., 4 papers [39], [40], [49], [54]) of them discussed the new real-world vulnerabilities found by the tools, and only 12.73% (i.e., 7 papers [36], [39], [40], [49], [53], [54], [64]) of them discussed the tool's capability to detect more critical vulnerabilities. For example, while not executable in our study, VulDeePecker was used by the authors to have detected 4 new vulnerabilities on 3 real-world projects, which were reported to the CVE database [14] [39]. Devign selected the latest 112 vulnerability samples in the CVE database and checked whether it had the potential to detect zero-day vulnerabilities, showing the tool's capability to detect more critical vulnerabilities [53]. The CVE database does have information regarding criticality/severity of each archived case. Also, usually following the assignment of CVEs, software engineering professionals (e.g., the developers of the vulnerable software project) offer feedback through responses/reactions to bug reports. Therefore, we suggest that future studies should assess tool practicability (e.g., discussing vulnerability criticality/severity, security consequence/impact) rather than only reporting/improving the numbers regarding common effectiveness metrics.

14. https://cve.mitre.org/data/downloads/index.html

Assessing technique stability. In Section 4.4.3, we noticed that the performance of some tools was significantly impacted by variations in dataset size and balance. Since the dataset used in a replication study is very likely to differ from the dataset used in authors' original evaluation, these and other dataset dimensions can be key players in replication failures. We thus suggest researchers conduct extensive evaluations to assess the stability of their techniques against variations in those dataset dimensions.

Our study results also offered further actionable strategies for improving the stability when found undesirable initially. We noticed that the performance of several tools was not significantly impacted (e.g., ZMVP and ReVeal) by variations in dataset size and balance because they explicitly accounted for such factors by adopting dedicated handling mechanisms (e.g, better program representation, oversampling, or undersampling). These mechanisms helped reduce overfitting hence improve the stability of the techniques. Therefore, we suggest that researchers explicitly consider stability when they design their techniques (e.g., by incorporating dedicated instability-mitigating mechanisms), and include performance evaluations against varying dataset characteristics to validate model stability.

Providing pre-trained models. In Section 4.3, we noticed that LAVDNN reported perfect reproduction results because it provided a pre-trained DL model. The randomness in training a DL model can become an obstacle for other researchers to exactly reproduce the original experiment results. Offering pre-trained models is a simple yet effective way to address the issue in reproduction of ML/DL-based research, while additionally saving computation resources (that would be incurred for re-training). Thus, we suggest authors of relevant research may consider including pre-trained models in their shared artifact package to ease reproduction by others.

5.3 Open Science in General Software Engineering

Since our study was focused on DL-based vulnerability detection techniques, we cannot claim the generalizability of our results to other areas in or beyond SE. The specific numbers for the four open-science dimensions we obtained may vary from tools to tools and from domains to domains in absolute terms. However, we would like to note that both the current status and success/failure effects/causes of open science with the studied works are not necessarily tied with the particular technical nature of this chosen area of study. In particular, we have not found any clear links between our availability/executability results and the fact that the studied tools are DL-based techniques of any kind. For instance, none of tools that were found executable or non-executable were so because they were DL-based or they aimed to detect vulnerabilities. For the results on reproducibility and replicability, our analyses of successes and failures were indeed linked to the DL-based nature of the studied tools, but the results were not clearly linked to the fact that they were vulnerability detectors. For example, none of the tools that were found not replicable were so because they dealt with the task of detecting vulnerabilities.

Thus, we believe our results on the four studied aspects of open science, and accordingly the insights and recommendations, are likely to apply to broader areas in SE. Next, we discuss such insights and make more general recommendations for SE broadly.

Making tools publicly available for open science. In Section 4.1, our investigations revealed that only one fourth of the DL-based software vulnerability detection studies (published up to late year 2020) provided publicly available tools. This critical lack in tool availability, the very first condition for reproduction and replication, rendered a concerning situation of open science in this area. Given the discussed benefits of open science, we suggest researchers in our community take better efforts to support open science by first making their research publicly accessible.

Our study also revealed that the ways of obtaining the tool links varied, limiting the access by the public. On the other hand, we observed the success pattern of offering easy-to-find pointers in papers to artifacts stored at persistent locations. Thus, we strongly advocate that our community should establish/improve open science policies to explicitly urge authors to present the artifact pointers (e.g., tool links) in a conspicuous and consistent place in their papers. In addition, in case that such pointers may expire soon, we further suggest authors may share their artifacts (e.g., source code and data) at persistent, archived places (e.g., Figshare [107]).

Offering complete/functional tool with sufficient documentation. In Section 4.2, we observed that 27.3% of the *available* tools were incomplete and/or not fully functional, making them *not executable*. Besides, the documentation of 54.5% of these *available* tools was insufficient or even entirely missing. While executable, some of the tools did not document required dependencies or experimentation steps, which largely impeded and slowed down our setup processes. In contrast, all of the tools that provided detailed and sufficient documentation were not only easy to set up and execute, but also strongly *reproducible* (Section 4.3). Thus, we strongly urge the SE research community to mandate that authors provide complete and functional tool packages, along with sufficient documentation on tool setup and experimentation instructions, to enable or facilitate artifact reproduction and replication.

Ensuring consistency in dataset and tool implementation. In Section 4.3, it was encouraging to see that almost all of the (executable) tools involved in our reproduction study were successfully reproduced. we noticed that these successes were mainly attributed to the authors providing datasets, tool implementation, and configuration information that are consistent between the artifact package and relevant descriptions in the original papers. And the only failing case was found the failure causes also in these same regards. Accordingly, We suggest that researchers supporting open science may make efforts to ensure these consistencies.

Developing metrics of reproducibility and replicability. In Sections 3.4 and 3.5, we considered F1 score as the metric for evaluating reproducibility and replicability (R&R) and we considered 1% and 5% deviations as the thresholds to differentiate successes versus failures. This method was based on the prevalent use of F1 and relevant conventions widely used in statistical analysis. However, to the best of our knowledge, there are no generally accepted metrics for quantifying R&R. Other researchers investigating R&R may use different metrics for such evaluations. For example, Liu et al. [27] used four different metrics to evaluate R&R of four DL-based software engineering (vulnerability-irrelevant) tools, while taking minimum, maximum, mean, and standard deviation to determine successes/failures. A standardized or at least widely accepted set of metrics for R&R would provide a necessary common reference, as well as a potential incentive, for researchers to evaluate R&R, hence potentially closing extant gaps in open science. We thus suggest out community take some efforts to develop such metrics and publicize them.

Developing community-agreed approaches for open science. In Section 5.1, we noticed that there was no significant association between the published venue rankings and open science. The top-venue papers did not seem to be in a better open-science status than the preprint or unranked-venue papers because the open science policies in these top venues are too coarse. Thus, besides the several recommendations discussed above, we call for a community-wide effort to establish an open-science standard for SE (like a similar effort focusing on how to conduct empirical studies [108]). The standard may put up recommendations like ours above as an agreement, and enforce it via major SE venue call-for-papers. Currently some conferences (e.g., ICSE and FSE) already enforce sharing of open data, unless there is a justified reason not to do so. But we should aim to do better than that. Requiring mere data sharing is not enough. Except for strong extenuating reasons related to the nature of the research, the open-science requirements ought to be more demanding. For example, the open science policies should ask authors to submit code and reproduction/replication documentation. Finally, the open-source standard should not only address what are required but also include recommended processes/procedures for achieving /validating the fulfillment of each open-science requirement. For instance, we may standardize the data sharing protocol/instrument such as the platform to use and the mechanisms to adopt (e.g., promoting the use of virtual machines/containers to facilitate faithful reproduction). For another example, researchers should conduct more evaluations using real-world/industrial datasets in realistic settings (e.g. independent testing with respect to real-world data imbalance situations) to assess the practicality of techniques/tools.

5.4 Threats to Validity and Limitations

Internal validity. The major threat to the internal validity of our study results lies in the possible errors that occurred when we manually reviewed the papers, set up the tools, and inspected the tool implementation. To mitigate the problem, we conducted multiple rounds of careful inspection of all the relevant data, code, and other information that we used to draw any conclusions presented in the paper.

Some of the criteria used in our study were subjective. For example, decisions on whether the documentation of a tool was sufficient were made in the paper according to our own relevant knowledge. It is possible that other researchers would determine that the documentation of a tool,

which was deemed insufficient in our study, is sufficient. Another example is that we were not able to check all the details of the tool implementation and configurations, thus the conclusions on the implementation and configuration consistencies drawn in RQ3 could be flawed. To mitigate the problem, we contacted the authors to verify our findings in these regards when necessary (e.g., we were uncertain).

For reproduction experiments, ideally we would like to always use exactly the same experimental setting and computing environments as those used for producing the original, published results. We did ensure that we followed the same experimental settings as originally used, including the code, data, and library dependencies. Yet practically we only have one set of hardware, thus we could not use exactly the same computing platform for every paper as that in the original experiment, as we explained in Section 3.4. As a result, our reproduction results and related conclusions are subject to potential biases induced by the differences in computing platforms (e.g., GPUs) used by us versus those by the original paper authors.

External validity. The primary threat to the external validity of our results concerns the choices of the investigated studies we used for evaluating the status of open science. To mitigate this problem, we have tried our best to collect as thoroughly as possible all the relevant papers. To the best of our knowledge, the 55 investigated papers/studies were those that introduced a DL-based software vulnerability detection technique published and indexed by Google Scholar prior to the end of October 2020.

Another threat to external validity lies in the representativeness of the dataset we chose for evaluating replicability and that we only considered a single dataset for this evaluation. To mitigate the threat, we chose a dataset containing a large number of real-world program samples collected from multiple, diverse open-source projects. We chose this dataset purposely when keeping in mind that our findings based on it could be better generalized to real-world vulnerability detection scenarios. Nevertheless, it is still possible that some of our findings and conclusions would shift if a different replication dataset were chosen.

Conclusion validity. We chose to focus on a particular topic area of SE (i.e., DL-based vulnerability detection) in order to enable in-depth investigations into open-science successes/failures hence actionable insights/recommendations for promoting good practices while improving against insufficiencies, as justified earlier. This choice, however, also limited our study scope, making our results and conclusions more applicable/relevant to this or similar (e.g., DL-based techniques for other SE tasks) areas than others. Thus, we cannot claim that our insights/recommendations are surely generalizable in SE broadly, especially those regarding reproducibility and replicability.

6 Related Work

Open science in SE has been previously examined. Several studies [3], [5], [6], [7], [8], [9] discussed the definitions and standards of open science, replicability, and reproducibility. In [10], [11], [12], [13], the authors demonstrated the benefits of open science in SE through several concrete examples. In [14], [15], possible challenges and barriers of promoting open science in SE were discussed. Other studies [16], [17] provided open science guidelines and recommendations for researchers in SE. However, these papers only discussed open science conceptually or using a few examples, unlike ours empirically and extensively investigating the state, success patterns, and failure causes of open science.

De Magalhães [18] studied how and how often the SE papers published in the last 17 years replicated other studies, but whether the experiment results were replicable was not discussed. Rodríguez-Pérez [19] checked whether the studies using the SZZ algorithm discussed replication packages in their papers, but no actual empirical experiment on replication was conducted. Mahmood [20] compared the software defect prediction results in the original reporting papers with the replication results obtained in other studies, but the replication experiments were conducted by different researchers. Overall, these previous studies investigated open science, focusing on replicability and reproducibility, but largely through literature reviews and qualitative discussions rather than empirical assessment like ours.

In [22], [23], the authors investigated whether they could obtain the source code of some studies, but they did not investigate whether the obtained source code could replicate the original experiment results. Kitchenham et al. [109] attempted to repeat the meta-analysis in other papers to assess their reproducibility and validity. In [25], the authors conducted experiments to evaluate the DL model's reproducibility, but only one specific model was considered. Piantadosi et al. [26] evaluated the reproducibility of DL studies by checking whether the studies documented sufficient information, but no actual reproducibility experiment on these studies was conducted. Daoudi et al. [103] tried to reproduce/replicate five machine learning based malware detection approaches and investigated the barriers for reproducing/replicating them. In [110], the authors replicated and extended the state-of-the-art multi-objective software effort estimation CoGEE with an independent implementation, which consolidated the validity of the original study. In [27], the replicability and reproducibility of several DL-based techniques studies were assessed. The assessment involved many topics yet without conducting in-depth analysis of why certain techniques were not replicable/reproducible as we did.

The study in [35] examined how well existing DL-based vulnerability detection techniques perform in a real-world vulnerability prediction scenario. They found the performance of the existing techniques dropped dramatically because of the challenges with the real-world data (e.g., data duplication, imbalanced training data, etc.). They focused on the failure reasons of four tools and discussed DL model related factors without quantifying the statistical relationships between the factors and model performance. In comparison, our study investigated the reproducibility of seven tools, filtered from a large set of (55) techniques. We also conducted in-depth studies to discover success/failure patterns/causes in various relevant characterization dimensions, as well as additional case studies to establish the impact of several dataset characteristics via statistical analysis.

Beyond SE, open science has been studied in other areas as well (e.g., replicability in computer graphics [21], repeatability in computer systems research [24]). They had some

similar findings to ours in that they also found code/tools were not always shared or the shared artifact did not correspond to the one used for producing the published results. Yet they focused primarily on availability, rather than conducting/diagnosing replication or replication in depth. For example, in [24], the authors investigated 601 papers published in ACM journals and conferences and checked the availability of source code and datasets used. They also executed the source code against the dataset and attempted to reproduce, yet they did not validate reproducibility as we did (i.e., comparing the reproduction experiment results with the published ones). Accordingly, their recommendations were largely different from ours.

Moreover, compared to all prior relevant studies, ours is more comprehensive in that it not only addressed one or two commonly-studied aspects of open science (replicability and reproducibility) but also systematically assessed availability and executability as well.

7 Conclusion

While open science is increasingly promoted in the SE community, it has not been systematically studied. In this paper, we intended to start filling this gap with such a systematic study targeting a specific topic area in SE.

We thoroughly investigated the open science status of DL-based software vulnerability detection techniques. We collected all the 55 relevant papers/studies published and indexed on Google Scholar up to October 2020. Based on this body of literature, we extensively examined four aspects of open science: *availability* (whether the papers provided publicly available tools), *executability* (whether the publicly available tools were executable), *reproducibility* (whether the executable tools can reproduce the originally reported performance results), and *replicability* (whether the originally reported performance results of reproducible tools can be replicated on a different dataset). For each aspect, in addition to reporting the current status, we carefully examined success patterns and failure causes in that aspect. Based on our findings and insights, we offered actionable suggestions to improve open science both in the studied area (of DL-based vulnerability detection) and beyond.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This research was sponsored by the Army Research Office (W911NF-21-1-0027) and Office of Naval Research (N000142212111).

References

[1] P. Masuzzo and L. Martens, "Do you speak open science? resources and tips to learn the language," PeerJ Preprints, Tech. Rep., 2017.

[2] P. Mirowski, "The future (s) of open science," *Social studies of science*, vol. 48, no. 2, pp. 171–203, 2018.

[3] D. Mendez, D. Graziotin, S. Wagner, and H. Seibold, "Open science in software engineering," in *Contemporary Empirical Methods in Software Engineering*. Springer, 2020, pp. 477–501.

[4] F. Ferreira, L. L. Silva, and M. T. Valente, "Software engineering meets deep learning: A literature review," in *SAC*, 2021, pp. 1542–1549.

[5] A. Brooks, J. Daly, J. Miller, M. Roper, and M. Wood, "Replication of experimental results in software engineering," *ISERN Technical Report 96-10, University of Strathclyde*, vol. 2, 1996.

[6] N. Juristo and O. S. Gómez, "Replication of software engineering experiments," in *Empirical software engineering and verification*. Springer, 2010, pp. 60–88.

[7] O. S. Gómez, N. Juristo, and S. Vegas, "Replications types in experimental disciplines," in *ESEM*, 2010, pp. 1–10.

[8] S. Krishnamurthi and J. Vitek, "The real software crisis: Repeatability as a core value," *CACM*, vol. 58, no. 3, pp. 34–36, 2015.

[9] N. Juristo and S. Vegas, "The role of non-exact replications in software engineering experiments," *EMSE*, vol. 16, no. 3, pp. 295–324, 2011.

[10] P. Louridas and G. Gousios, "A note on rigour and replicability," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 5, pp. 1–4, 2012.

[11] G. Robles and D. M. Germán, "Beyond replication: An example of the potential benefits of replicability in the mining of software repositories community," in *RESER*, 2010, pp. 1–4.

[12] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in empirical software engineering," *EMSE*, vol. 13, no. 2, pp. 211–218, 2008.

[13] D. C. Ince, L. Hatton, and J. Graham-Cumming, "The case for open computer programs," *Nature*, vol. 482, no. 7386, pp. 485–488, 2012.

[14] J. M. González-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," *EMSE*, vol. 17, no. 1, pp. 75–89, 2012.

[15] S. M. Easterbrook, "Open code for open science?" *Nature Geoscience*, vol. 7, no. 11, pp. 779–781, 2014.

[16] J. Fehr, J. Heiland, C. Himpe, and J. Saak, "Best practices for replicability, reproducibility and reusability of computer-based experiments exemplified by model reduction software," *arXiv preprint arXiv:1607.01191*, 2016.

[17] S. Vollmer, B. A. Mateen, G. Bohner, F. J. Király, R. Ghani, P. Jonsson, S. Cumbers, A. Jonas, K. S. McAllister, P. Myles *et al.*, "Machine learning and artificial intelligence research for patient benefit: 20 critical questions on transparency, replicability, ethics, and effectiveness," *BMJ*, vol. 368, 2020.

[18] C. V. de Magalhães, F. Q. da Silva, R. E. Santos, and M. Suassuna, "Investigations about replication of empirical studies in software engineering: A systematic mapping study," *IST*, vol. 64, pp. 76–101, 2015.

[19] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *IST*, vol. 99, pp. 164–176, 2018.

[20] Z. Mahmood, D. Bowes, T. Hall, P. C. Lane, and J. Petrić, "Reproducibility and replicability of software defect prediction studies," *IST*, vol. 99, pp. 148–163, 2018.

[21] N. Bonneel, D. Coeurjolly, J. Digne, and N. Mellado, "Code replicability in computer graphics," *TOG*, vol. 39, no. 4, pp. 93–1, 2020.

[22] A. Boll and T. Kehrer, "On the replicability of experimental tool evaluations in model-based development," in *ICSMM*, 2020, pp. 111–130.

[23] G. Robles, "Replicating MSR: A study of the potential replicability of papers published in the mining software repositories proceedings," in *MSR*, 2010, pp. 171–180.

[24] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *CACM*, vol. 59, no. 3, pp. 62–69, 2016.

[25] G. Piantadosi, S. Marrone, and C. Sansone, "On reproducibility of deep convolutional neural networks approaches," in *International Workshop on Reproducible Research in Pattern Recognition*, 2018, pp. 104–109.

[26] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence," in *AAAI*, vol. 32, no. 1, 2018, pp. 1644–1651.

[27] C. Liu, C. Gao, X. Xia, D. Lo, J. Grundy, and X. Yang, "On the replicability and reproducibility of deep learning in software engineering," *arXiv preprint arXiv:2006.14244*, 2020.

[28] E. C. McKiernan, P. E. Bourne, C. T. Brown, S. Buck, A. Kenall, J. Lin, D. McDougall, B. A. Nosek, K. Ram, C. K. Soderberg *et al.*, "Point of view: How open science helps researchers succeed," *elife*, vol. 5, p. e16800, 2016.

[29] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.

[30] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *International Conference on Multimedia Information Networking and Security*, 2012, pp. 152–156.

[31] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[32] G. Lin, J. Zhang, W. Luo, L. Pan, O. De Vel, P. Montague, and Y. Xiang, "Software vulnerability discovery via learning multi-domain knowledge bases," *TDSC*, vol. 18, no. 5, pp. 2469–2485, 2019.

[33] R. Li, C. Feng, X. Zhang, and C. Tang, "A lightweight assisted vulnerability discovery method using deep neural networks," *IEEE Access*, vol. 7, pp. 80 079–80 092, 2019.

[34] M. Zagane, M. K. Abdi, and M. Alenezi, "Deep learning for software vulnerabilities detection using code metrics," *IEEE Access*, vol. 8, pp. 74 562–74 570, 2020.

[35] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *TSE*, 2021, early Access.

[36] T. Nguyen, T. Le, K. Nguyen, O. de Vel, P. Montague, J. Grundy, and D. Phung, "Deep cost-sensitive kernel machine for binary software vulnerability detection," in *PAKDD*, 2020, pp. 164–177.

[37] T. Le, T. Nguyen, T. Le, D. Phung, P. Montague, O. De Vel, and L. Qu, "Maximal divergence sequential autoencoder for binary software vulnerability detection," in *ICLR*, 2018, pp. 1–15.

[38] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, "Towards security defect prediction with ai," *arXiv preprint arXiv:1808.09897*, 2018.

[39] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *NDSS*, pp. 1–15, 2018.

[40] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *TDSC*, vol. 19, no. 4, pp. 2244–2258, 2021.

[41] S. Liu, M. Dibaei, Y. Tai, C. Chen, J. Zhang, and Y. Xiang, "Cyber vulnerability intelligence for internet of things binary," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 3, pp. 2154–2163, 2019.

[42] Association for Computing Machinery, "Artifact review and badging - current: Terminology," https://shorturl.at/efior, 2021.

[43] A. Rasool, "Which is the most vulnerable programming language?" https://shorturl.at/fHJQ1, 2019.

[44] J. Brownlee, "Classification accuracy is not enough: More performance measures you can use," https://shorturl.at/diKR2, 2014.

[45] S. M. Ross, *Introduction to probability and statistics for engineers and scientists*. Academic Press, 2020.

[46] L. Li and A. Talwalkar, "Random search and reproducibility for neural architecture search," in *UAI*, 2020, pp. 367–377.

[47] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep learning-based vulnerable function detection: A benchmark," in *ICICS*, 2019, pp. 219–232.

[48] Y. Fang, S. Han, C. Huang, and R. Wu, "Tap: A static analysis model for php vulnerabilities based on token and deep learning technology," *PLOS One*, vol. 14, no. 11, p. e0225196, 2019.

[49] N. Guo, X. Li, H. Yin, and Y. Gao, "Vulhunter: An automated vulnerability detection system based on deep learning and bytecode," in *ICICS*, 2019, pp. 199–218.

[50] N. Saccente, J. Dehlinger, L. Deng, S. Chakraborty, and Y. Xiong, "Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network," in *ASE*, 2019, pp. 114–121.

[51] J. Gao, X. Yang, Y. Fu, Y. Jiang, H. Shi, and J. Sun, "Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation," in *FSE*, 2018, pp. 803–808.

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

[53] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *NIPS*, vol. 32, pp. 1–11, 2019.

[54] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *TDSC*, vol. 19, no. 4, pp. 2821–2837, 2021.

[55] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A deep learning-based system for multiclass vulnerability detection," *TDSC*, vol. 18, no. 5, pp. 2224–2236, 2019.

[56] M.-j. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-end prediction of buffer overruns from raw source code via neural memory networks," *arXiv preprint arXiv:1703.02458*, 2017.

[57] V. Nguyen, T. Le, O. de Vel, P. Montague, J. Grundy, and D. Phung, "Dual-component deep domain adaptation: A new approach for cross project software vulnerability detection," in *PAKDD*, 2020, pp. 699–711.

[58] S. Liu, G. Lin, Q.-L. Han, S. Wen, J. Zhang, and Y. Xiang, "Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection," *TFS*, vol. 28, no. 7, pp. 1329–1343, 2019.

[59] J. Zheng, J. Pang, X. Zhang, X. Zhou, M. Li, and J. Wang, "Recurrent neural network based binary code vulnerability detection," in *ACAI*, 2019, pp. 160–165.

[60] J. Tian, W. Xing, and Z. Li, "Bvdetector: A program slice-based binary code vulnerability intelligent detection system," *IST*, vol. 123, p. 106289, 2020.

[61] X. Li, L. Wang, Y. Xin, Y. Yang, and Y. Chen, "Automated vulnerability detection in source code using minimum intermediate representation learning," *Applied Sciences*, vol. 10, no. 5, p. 1692, 2020.

[62] J. Harer, O. Ozdemir, T. Lazovich, C. Reale, R. Russell, L. Kim *et al.*, "Learning to repair software vulnerabilities with generative adversarial networks," *NIPS*, vol. 31, pp. 7944–7954, 2018.

[63] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, and Y. Xiang, "CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation," *TDSC*, vol. 19, no. 1, pp. 438–451, 2020.

[64] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *ICECCS*, 2019, pp. 41–50.

[65] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities." in *IJCAI*, 2019, pp. 4665–4671.

[66] G. Huang, Y. Li, Q. Wang, J. Ren, Y. Cheng, and X. Zhao, "Automatic classification method for software vulnerability based on deep neural network," *IEEE Access*, vol. 7, pp. 28 291–28 298, 2019.

[67] C. J. Clemente, F. Jaafar, and Y. Malik, "Is predicting software security bugs using deep learning better than the traditional machine learning algorithms?" in *QRS*, 2018, pp. 95–102.

[68] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei, and H. Changyu, "CPGVA: code property graph based vulnerability analysis by deep learning," in *ICAIT*, 2018, pp. 184–188.

[69] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," *arXiv preprint arXiv:2006.08614*, 2020.

[70] Y. Hu, "A framework for using deep learning to detect software vulnerabilities," 2019, MS thesis. Harbin Institute of Technology.

[71] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *arXiv preprint arXiv:1708.02368*, 2017.

[72] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Deep semantic feature learning with embedded static metrics for software defect prediction," in *APSEC*, 2019, pp. 244–251.

[73] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *ICCC*, 2017, pp. 1298–1302.

[74] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, and D. Phung, "Deep domain adaptation for vulnerable code function identification," in *IJCNN*, 2019, pp. 1–8.

[75] M. A. Albahar, "A modified maximal divergence sequential autoencoder and time delay neural network models for vulnerable binary codes detection," *IEEE Access*, vol. 8, pp. 14 999–15 006, 2020.

[76] R. Li, C. Zhang, C. Feng, X. Zhang, and C. Tang, "Locating vulnerability in binaries using deep neural networks," *IEEE Access*, vol. 7, pp. 134 660–134 676, 2019.

[77] R. Demidov and A. Pechenkin, "Application of siamese neural networks for fast vulnerability detection in mips executable code," in *FTC*, 2019, pp. 454–466.

[78] S. Srikant, "Vulcan: classifying vulnerabilities in solidity smart contracts using dependency-based deep program representations," Ph.D. dissertation, MIT, 2020.

[79] A. Tanwar, K. Sundaresan, P. Ashwath, P. Ganesan, S. K. Chandrasekaran, and S. Ravi, "Predicting vulnerability in large codebases with deep code representation," *arXiv preprint arXiv:2004.12783*, 2020.

[80] Y. Pang, X. Xue, and H. Wang, "Predicting vulnerable software components through deep neural network," in *ICDLT*, 2017, pp. 6–10.

[81] G. Jabeen, L. Ping, J. Akram, and A. A. Shah, "An integrated software vulnerability discovery model based on artificial neural network." in *SEKE*, 2019, pp. 349–458.

[82] C. Catal, A. Akbulut, S. Karakatič, M. Pavlinek, and V. Podgorelec, "Can we predict software vulnerability with deep neural network?" in *International Multiconference: Information Society*, 2016, pp. 19–22.

[83] J. Kim, D. Hubczenko, and P. Montague, "Towards attention based vulnerability discovery using source code representation," in *ICANN*, 2019, pp. 731–746.

[84] C. Catal, A. Akbulut, E. Ekenoglu, and M. Alemdaroglu, "Development of a software vulnerability prediction web service based on artificial neural networks," in *PAKDD*, 2017, pp. 59–67.

[85] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for predicting vulnerable software components," *TSE*, vol. 47, no. 1, pp. 67–85, 2018.

[86] X. Ban, S. Liu, C. Chen, and C. Chua, "A performance evaluation of deep-learnt features for software vulnerability detection," *CCPE*, vol. 31, no. 19, p. e5103, 2019.

[87] W. Han, J. Pang, X. Zhou, and D. Zhu, "Binary software vulnerability detection method based on attention mechanism," in *ICMCCE*, 2020, pp. 1462–1466.

[88] W. An, L. Chen, J. Wang, G. Du, G. Shi, and D. Meng, "AVDHRAM: Automated vulnerability detection based on hierarchical representation and attention mechanism," in *ISPA/BDCloud/SocialCom/SustainCom*, 2020, pp. 337–344.

[89] Y. Mao, Y. Li, J. Sun, and Y. Chen, "Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks," in *Big Data*, 2020, pp. 4651–4656.

[90] D. Cao, J. Huang, X. Zhang, and X. Liu, "FTCLNet: Convolutional LSTM with Fourier transform for vulnerability detection," in *TrustCom*, 2020, pp. 539–546.

[91] K. Filus, M. Siavvas, J. Domańska, and E. Gelenbe, "The random neural network as a bonding model for software vulnerability prediction," in *MASCOTS*, 2020, pp. 102–116.

[92] A. Lima, L. Rossi, and M. Musolesi, "Coding together at scale: Github as a collaborative social network," in *AAAI*, vol. 8, no. 1, 2014, pp. 295–304.

[93] X. Meng and B. P. Miller, "Binary code is not easy," in *ISSTA*, 2016, pp. 24–35.

[94] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *SPLASH*, 2019, pp. 143–153.

[95] M. S. Santos, J. P. Soares, P. H. Abreu, H. Araujo, and J. Santos, "Cross-validation for imbalanced datasets: avoiding overoptimistic and overfitting approaches [research frontier]," *IEEE Computational Intelligence Magazine*, vol. 13, no. 4, pp. 59–76, 2018.

[96] R. Perego, S. Orlando, and P. Palmerini, "Enhancing the *Apriori* algorithm for frequent set counting," in *International Conference on Data Warehousing and Knowledge Discovery*, 2001, pp. 71–82.

[97] Y. Nong and H. Cai, "A preliminary study on open-source memory vulnerability detectors," in *SANER*, 2020, pp. 557–561.

[98] Y. Nong, H. Cai, P. Ye, L. Li, and F. Chen, "Evaluating and comparing memory error vulnerability detectors," *IST*, vol. 137, p. 106614, 2021.

[99] D. Beyer, "Automatic verification of C and Java programs: SV-COMP 2019," in *TACAS*, 2019, pp. 133–155.

[100] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2014, pp. 389–391.

[101] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *CGO*, 2011, pp. 213–223.

[102] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision." in *USENIX ATC*, 2005, pp. 17–30.

[103] N. Daoudi, K. Allix, T. F. Bissyandé, and J. Klein, "Lessons learnt on reproducibility in machine learning based android malware detection," *EMSE*, vol. 26, no. 4, pp. 1–53, 2021.

[104] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR*, 2009, pp. 248–255.

[105] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for Java programs," in *ISSTA*, 2014, pp. 437–440.

[106] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs. jar: a large-scale, diverse dataset of real-world Java bugs," in *MSR*, 2018, pp. 10–13.

[107] D. Science, "Figshare," https://figshare.com, 2022.

[108] P. Ralph, N. b. Ali, S. Baltes, D. Bianculli, J. Diaz, Y. Dittrich, N. Ernst, M. Felderer, R. Feldt, A. Filieri *et al.*, "Empirical standards for software engineering research," *arXiv preprint arXiv:2010.03525*, 2020.

[109] B. Kitchenham, L. Madeyski, and P. Brereton, "Meta-analysis for families of experiments in software engineering: a systematic review and reproducibility and validity assessment," *EMSE*, vol. 25, no. 1, pp. 353–401, 2020.

[110] V. Tawosi, F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation: A replication study," *TSE*, vol. 48, no. 8, pp. 3185–3205, 2021.