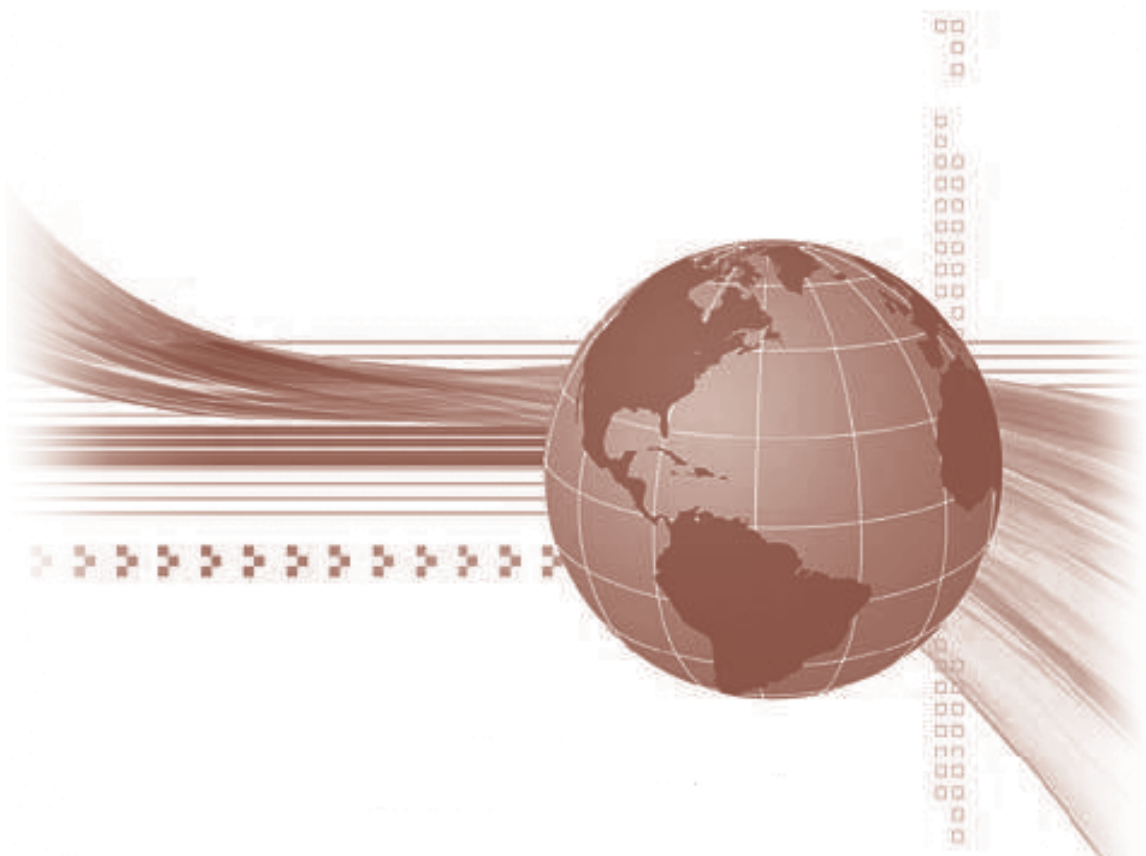
STUDIA UNIVERSITATIS
BABEŞ-BOLYAI

UNIVERSITATEA BABEŞ-BOLYAI
BABEŞ-BOLYAI TUDOMÁNYEGYETEM
BABEŞ-BOLYAI UNIVERSITÄT
BABEŞ-BOLYAI UNIVERSITY
TRADITIO ET EXCELLENTIA



INFORMATICA

1/2023

STUDIA

UNIVERSITATIS BABEŞ-BOLYAI
INFORMATICA

No. 1/2023

January - June

ISSN (online): 2065-9601; ISSN-L: 2065-9601

Published by Babeș-Bolyai University

EDITORIAL BOARD

YEAR **Volume 68 (LXVIII) 2023**
MONTH **JUNE**
ISSUE **1**

S T U D I A

UNIVERSITATIS BABEȘ-BOLYAI

INFORMATICA

1

EDITORIAL OFFICE: M. Kogălniceanu 1 • 400084 Cluj-Napoca • Tel: 0264.405300

SUMAR – CONTENTS – SOMMAIRE

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 1, 2023
DOI: 10.24193/subbi.2023.1.01

MALWARE ANALYSIS AND STATIC CALL GRAPH GENERATION WITH RADARE2

ATTILA MESTER

Abstract. A powerful feature used in automated malware analysis is the static call graph of the executable file. Elimination of sandbox environment, fast scan, function call patterns beyond instruction level information – all of these motivate the prevalence of the feature. Processing and storing the static call graph of malicious samples in a scaled manner facilitates the application of complex network analysis in malware research. IDA Pro is one of the leading disassembler tools in the industry and can generate the call graph via *GenCallGdl* and *GenFuncGdl* APIs – a tool which was used in our previous works. In this paper an alternative analysis method is presented using another disassembler tool, Radare2, an open-source Unix-based software, which is also frequently used in this domain. Radare2 has Python support (among other languages), via the *r2pipe* package, thus enabling full scalability on Linux-based servers using containerized solutions. This paper offers a detailed technical description on how to use Radare2 to generate the static call graph of a PE file and a thorough comparison with the output of IDA Pro, as well as a public dataset on which the experiments were carried out.

1. Introduction

Analyzing malware in an automated manner not only eases the workload of cybersecurity experts, but it is a necessity in this domain, due to the number of new threats rising globally on a daily basis. A key statistic provided by AV-TEST[1] is the daily emerging several tens of thousands of malicious threats. In 2022, roughly one hundred million new samples were discovered – that is ≈ 3 new malicious files per second. While these threats come from different types of attacks and exploits such as phishing campaigns, emails, attachments,

Received by the editors: 1 March 2023.
2010 *Mathematics Subject Classification.* 68P25, 68P30.
1998 *CR Categories and Descriptors.* D.4.6 [**Security and Protection**]: Subtopic – *Invasive software*.
Key words and phrases. malware analysis, static call graph, radare2, IDA Pro.
[1] `https://www.av-test.org`

executables, android apps, etc., the leading source of malicious attacks comes from Windows executable files (i.e. PE).

PE files can carry a vast amount of different attack techniques, hence they can also contribute enormously to the process of gathering threat intelligence about the origin of the attack. One such exceedingly valuable piece of information is called *attribution* in literature. In its highly comprehensive book entitled *Attribution of Advanced Persistent Threats* [27] (APT) published in 2020, Steffens explains why it is so important to attribute an attack, as well as offers some detection ideas in this regard. There are fundamentally two options in this domain: static and dynamic analysis. These methods assume the use of either a sandbox environment – which is often expensive and time-consuming, or a disassembler tool such as IDA, Radare2, Ghidra, etc.

In this work, we present a malware analysis framework using Radare2 to extract the static call graph of a PE file and offer a detailed comparison with an alternative disassembler, IDA Pro 6. Our previous work [17, 18, 19] relies solely on IDA Pro 6 – this experience led to the need to try out an alternative disassembler tool which enables containerized, parallel processing of samples. Other alternatives were taken into consideration as well, but due to its popularity in the literature – as described in Section 2, our tool of choice became Radare2.

The paper is structured as follows. Section 2 covers the key directions in the literature of PE analysis based on static call graph features, using IDA or Radare2 tools. Our proposed framework for the generation of the static call graph using Radare2 is described in Section 3. We then compare the results of our analysis with the ones obtained with IDA, in Section 4. Our conclusions are presented in Section 5, as well as possible future research ideas using Radare2.

2. Related work

A recent survey paper [28] offers an ample overview on the literature of automated malware analysis using various machine learning techniques. A multitude of research papers are presented from the past decade, and it is clearly shown that one particular feature is by far the most frequently used in this domain – the static call graph. Our previous work [17] presents a detailed overview on the literature of PE analysis, based on this survey paper, visualizing the distribution of research work with histograms of the features and methods applied. The motivation to use one particularly interesting static feature, the call graph, is that it includes both topological information of an executable file regarding function call sequences, and also the x86 assembly instruction list of each local subroutine – one presumption of the analysis process

is that each of these local subroutines may be an original code of a malicious actor or APT group. There is a multitude of potential use cases of this feature. Using only the topological structure of the call graph, graph matching or graph edit distance (GED) may be applied [7, 22, 2, 13]. Attaching a feature vector to graph, based on n-grams, is also common practice [5, 8, 26], as well as applying graph embedding methods [23, 11]. The abundance of research work using this feature raises the importance of analyzing malicious code in a fast and scalable manner, preferably with a free, open-source tool which enables the automated extraction of the call graph.

In this section, we present some technical disadvantages of the disassembler tool used in our previous research work, the IDA Pro 6. This is one of the global leading solutions [20, 31] when it comes to static malware analysis, however, it has its limitations as well. One key aspect to mention is that IDA is commercial software, it offers scripted functionality only in its paid version, IDA Pro. Another major blocking issue using the scripted functionality of this tool is the series of unexpected runtime errors, which cause unnecessarily slow analysis – making it impossible for real-time use cases for example, where one needs to process a daily flux of new malicious samples.

3. Using Radare2 to obtain the static call graph

3.1. **IDA Pro alternatives.** As a consequence of the drawbacks of the IDA tool listed in Section 2, a list of potential alternative disassemblers was analyzed. Fortunately, there is a multitude of such tools: Binary Ninja [24], Hopper [1], Relyze [29], x64dbg[2], ODA[3], etc. One of the most popular alternatives is Ghidra[4], available on Windows/Linux, developed by NSA's Research Directorate under Apache License (FOSS) is a leading alternative to IDA Pro [25, 14]. The downside of this tool which made our choice of another alternative is the difficulty in using its scripted API call/graph generation.

Radare2[5] is also available on Windows/Linux (FOSS), and offers a lightweight alternative to Ghidra, while being able to integrate Ghidra decompiler *r2ghidra*[6]. It can be used from command-line interface (CLI) and also GUI, offered by Cutter[7]. A major power of this tool is the Python binding *r2pipe*[8], which offers extensive APIs for static analysis, including call graph inspection.

[2] `https://x64dbg.com/`
[3] `https://github.com/syscall7/oda`
[4] `https://ghidra-sre.org/`
[5] `https://www.radare.org/`
[6] `https://github.com/radareorg/r2ghidra`
[7] `https://cutter.re/`
[8] `https://r2wiki.readthedocs.io/`

Radare2 (also referred to as *r2*) has also great popularity in the cyber tech domain [16, 4, 9, 3, 12].

3.2. **Radare2 usage and commands.** Radare2[5] offers a clear description of its installation on their Github page[9] and has plenty of documentation and community support on their Wiki page[10] and their official e-book [21]. After installation, Radare2 can be invoked using the *radare2* or *r2* commands, specifying a path to a PE file.

In this CLI, a variety of commands is offered for analyzing sections, imports, exports, entry point information, blocks, function calls, for seeking certain parts of the binary, and much more – also, each command has a helper interface invocable by appending "?" after the respective command. Radare2 works with the concept of flags, i.e a bookmark at an offset like "fcn." or "sym.imp", meaning that every offset considered as interesting by Radare2 will be assigned a corresponding flag to it, e.g. strings, functions, imports, and much more. Analysis of a binary PE file can be started by the command "aaa", which analyzes all the flags in the file. Since this work focuses on the analysis of the static call graph, we will detail commands which are related to the analysis of the call sequences, function blocks, and entry points.

The majority of these *r2* commands have multiple output formats, available by specifying a formatter at the end of the command – such as the default ASCII art, or "j" for *json*, "d" for *dot*, "b" for "Braile art" i.e. short overview/bird's eye plot, or "w" for an interactive plot – highly useful for debugging purposes, similar to a *matplotlib* plot.

As mentioned in Section 3.1, Radare2 has also a GUI tool, Cutter, which offers a definitely positive usage experience due to its intuitive and simple interface.Even though Cutter makes it easy to analyze samples manually on a daily basis, for us a huge advantage of Radare2 comes from its CLI, which is clearly documented and offers fast analysis performance when called from Python scripts, enabling the continuous analysis of the samples on a real-time income flux.

3.3. **Generating the static call graph.** When generating the static call graph of a PE binary using Radare2, multiple *r2* commands are leveraged to obtain the final graph object. Radare2 offers Python bindings via the *r2pipe* package, which simply enables the pipeline of multiple *r2* commands without the need to open and load the file each and every time. Some of the commands mentioned here are detailed in Section 3.2. We start the analysis by calling "aaa" command. Then, entry point nodes are collected (i.e. function blocks)

[9] `https://github.com/radareorg/radare2`

[10] `https://r2wiki.readthedocs.io/`

```
import r2pipe
import pygraphviz
import networkx as nx                                              [...]
r2 = r2pipe.open(self.file_path)
r2.cmd("aaa")
entrypoint_info = r2.cmd("ie").split("\n")                         [...]
agCd = r2.cmd("agCd")
agRd = r2.cmd("agRd")                                              [...]
nx = nx.drawing.nx_agraph.from_agraph(pygraphviz.AGraph(agCd))
                                                                   [...]
for addr, data in nx.nodes.items():
    block = r2.cmd(f"agfd {addr}")                                 [...]
```

LISTING 1. Creating the call graph in Python using Radare2

by calling "ie". The *r2* commands that are used for call graph analysis are part of the "ag" command group.

The structure of the call graph is provided by the "agC" command, where the desired DOT format is specified using the "d" flag. The full reference graph (e.g. imports) is offered by "agR" command. It is important to mention that none of these two commands include node-level information regarding to the instruction list. In order to obtain the assembly code of each subroutine, "agf" command is called on each node of the call graph. In a similar manner to generating the call graph using IDA Pro 6, merging the output of "GenCallGdl" and "GenFuncGdl" [17, 18, 19], the same logic applies in Radare2 as well. Both the global function graph ("agC") and global references graph ("agR") is needed to be analyzed, furthermore, each function block ("agf") must be processed in order to obtain the final, complete call graph.

One key difference between IDA Pro 6 and Radare2 is that in the former, only 2 APIs have to be called, while in the latter, a multitude of *r2* commands are needed – $O(n)$ where n is the number of function blocks. The unexpected revelation is that despite all these aspects mentioned, Radare2 scans the binaries much faster and in a way more reliable way than IDA – scan time information is provided in Section 4 (note: this may be due to the environmental circumstances of the scripted analysis).

4. Comparing Radare2 with IDA Pro

The experiments were run on multiple machines, thus a reliable comparison of the runtime cannot be provided yet. IDA Pro 6 was run on *Windows Server 2012 R2*, while Radare2 was run on *Ubuntu 22.04*. An example output of the disassembler tools is shown in Figures 1 and 2, where the structure of the call

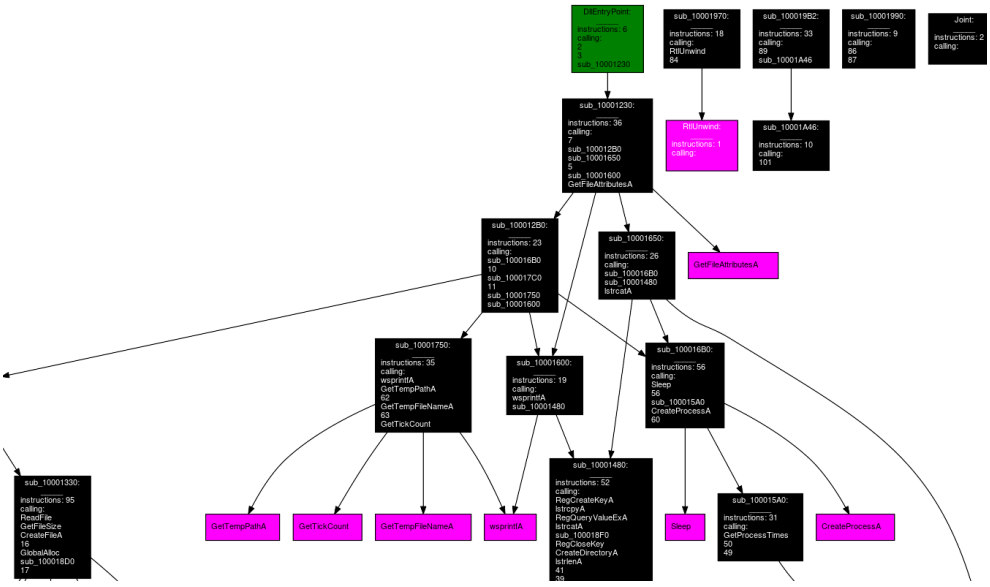


FIGURE 1. Call graph obtained with IDA Pro 6 ("GenCallGdl").

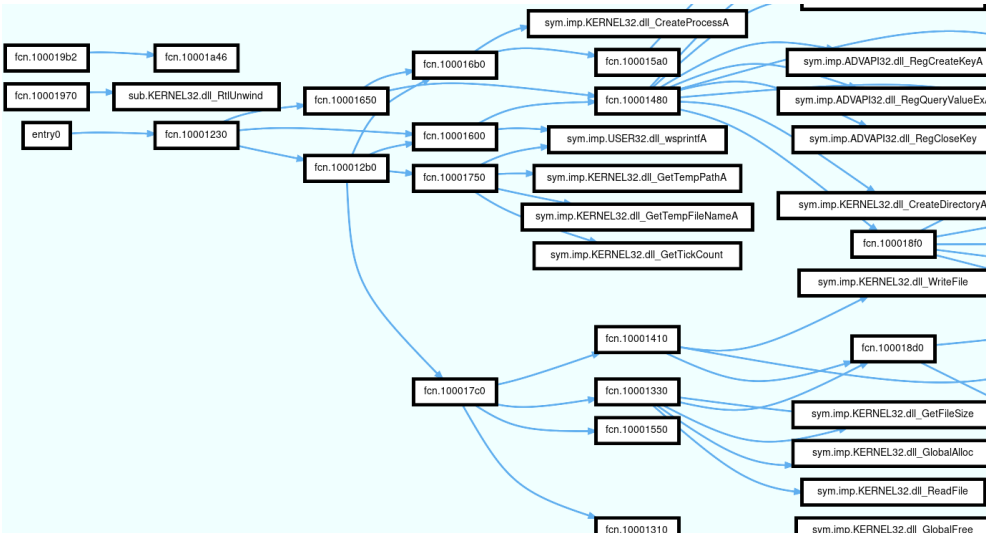


FIGURE 2. Call graph obtained with Radare2 ("agCd").

graph of the same executable file is depicted, dumped in DOT file format by each tool, and converted to SVG image.

The comparison of the call graph of a binary file is carried out on both topological level (edges, i.e. function/import calls) and node-level (x86 instruction list of the functions), and follows the following steps. The PE file is scanned with IDA Pro 6, using the method described in our previous works [17, 18, 19] – DFS traversal is applied on the control flow graph obtained by *GenFuncGdl* and it is then merged with the*GenCallGdl* API's output. The file is then scanned using Radare2, with the method described in Section 3.3. A series of normalizing steps are taken into consideration, e.g. IDA Pro subroutine labels follow the structure of "sub_0XXXXXX" (i.e. a capitalized RVA address), while Radare2 names its function blocks "fcn.0xxxxxx". Another example where normalization must be applied is on the instruction level: IDA prefers to use conditional jump instructions with the notation of *jump if zero* (e.g. "jz", "jnz", "repz", etc.), while Radare2 uses the form of *jump if equal* ("je", "jne", "repe"). These instructions have the same meaning, so they should not account in the node-level comparison of the call graphs.

4.1. **Comparison metrics.** The topological similarity of the call graphs is expressed with the Jaccard similarity of the edge set – an edge being represented by the name of its endpoints. For example, if the call graph from IDA has two edges, namely

$$[sub\_40010A \to sub\_400200, sub\_400200 \to sub\_400300],$$

and Radare2's call graph has also two edges, namely

$$[fcn.40010a \to fcn.400200, fcn.400200 \to sym.imp.kernel32.dll\_WriteFile],$$

then their topological similarity will be 0.33. Similarly, another topological similarity is calculated, referring to the node labels – the Jaccard score between the function label sets obtained from IDA and Radare2.

The node-level similarity is expressed using the similarity between the instruction lists (precisely, the mnemonic list) of each matching subroutine of the call graphs (in the sense of their label matching). For this purpose, several metrics are calculated, i.e. Levenshtein distance, relative Levenshtein similarity, Jaro distance, and Jaro-Winkler distance on each matching function block, and statistics are gathered regarding the minimum, maximum, average, median and 75% percentile of the values.

4.1.1. *Levenshtein distance.* The Levenshtein distance [15] is a commonly used distance metric in information theory, and it measures the number of edits needed to obtain one string from the other one. The edits permitted are insertion, deletion, and substitution. This is a naturally good distance metric in our application because we want to know how many instructions differ between

two function blocks, taking into consideration their place as well. Naturally, if this metric is 0, it means that the instruction lists match completely.

Since the function blocks may have varying lengths of instruction lists, a relative similarity should be expressed as well – two instruction lists having 100 assembly mnemonics that differ in only one instruction should have a higher similarity score than two functions having 2 mnemonics, differing in only one instruction. The relative distance, i.e. similarity is expressed in Equation 1, and its values are bound to the interval $[0, 1]$.

$$L_r(a, b) = 1 - L(a, b)/max(len(a), len(b)). \tag{1}$$

4.1.2. *Jaro distance.* The Jaro distance metric [10] is specifically designed for short strings, names, measuring the number of matching characters while taking into consideration the distance between them as well.

4.1.3. *Jaro-Winkler distance.* The Jaro-Winkler metric [30] is a variant of the Jaro distance. In addition to the former one, this metric takes into consideration not only the matching characters but also some scaling factor, i.e. the length of the common prefix. This way, it will have a higher similarity value for strings that are similar at the beginning – in contrast to the Jaro metric which considers the characters' position equally important. In this paper, all the results referring to Jaro and Jaro-Winkler distances are expressed as a similarity score in the $[0, 1]$ interval – 1 marking the perfect match. A detailed comparison between various distance metrics is described in [6].

4.2. **Dataset.** The dataset consists of publicly available samples, in order to increase the transparency of the experiments. The samples are part of a Kaggle competition[11]. 435 binary files were analyzed with the comparison method described in Section 4. The dataset was extracted from the Kaggle competition[11], and can be viewed on our page[12].

4.3. **Results.** To demonstrate the efficiency of the Radare2 scanner, a histogram of runtime values is presented in Fig. 3.

In each of the following images, Figs. 4, 5, 6, 7, 8, 9, 10, two sets of plots are shown, regarding the dimensions of the original IDA call graphs – plots referring to graphs having nodes in the $[0, 100)$ and $[100,)$ intervals, respectively. This was necessary in order to offer relevant statistics divided by the category in which they are measured.

Figures 6, 7, 8, 9, 10 represent histogram plots of the minimum, maximum, average, median and 75% percentile value of the respective metrics, which are measured on a set of nodes. The first row represents values measured on a

[11] `https://www.kaggle.com/competitions/malware-detection/data`

[12] `https://attilamester.github.io/call-graph/studia2022.html`

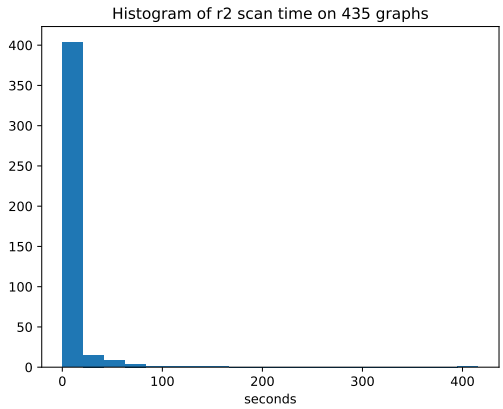


FIGURE 3. Runtime of the Radare2 scanner programme.

set of smaller graphs. Each of these graphs has a set of nodes (i.e. function blocks) – the metrics are calculated on these sets of nodes. The second row is calculated using the same logic applied on larger graphs. It should be noted that these images refer to node-level statistics, and some smaller graphs having under 10 nodes contain only functions marked as *sys.imp* (i.e. import functions) – thus, these graphs are excluded from these plots. That is the reason why these images contain statistics of only 425 graphs.

Fig. 4 has the purpose of showing the sizes of the call graphs which are examined in this paper. The upper and lower images show that the sizes range from almost empty graphs to enormously huge ones, topping at around 15 thousand nodes (i.e. functions) and 70 thousand edges (i.e. function calls). This fact highlights the need to separate each of the statistics into different categories. It can be also concluded that the majority of the dataset consists of call graphs having under one thousand nodes and edges – this is the information shown by the lower two rows of plots. Another conclusion could be that very few graphs have under 10 nodes or edges.

The topological similarities, as described in Section 4.1, are shown in Fig. 5. When measured on smaller graphs, in the upper row, it can be observed that the Jaccard is either 1, or a rather small value. Meanwhile, on larger graphs, this score barely reaches 1, which is natural, it is highly improbable that a sample whose call graph has hundreds of nodes will have the same scanning result in IDA and Radare2 as well. On the contrary, what can be confirmed is that the size of the graphs does not affect negatively this score – as the graphs grow, the average Jaccard still remains in the $[0.4, 0.6]$ interval. It should

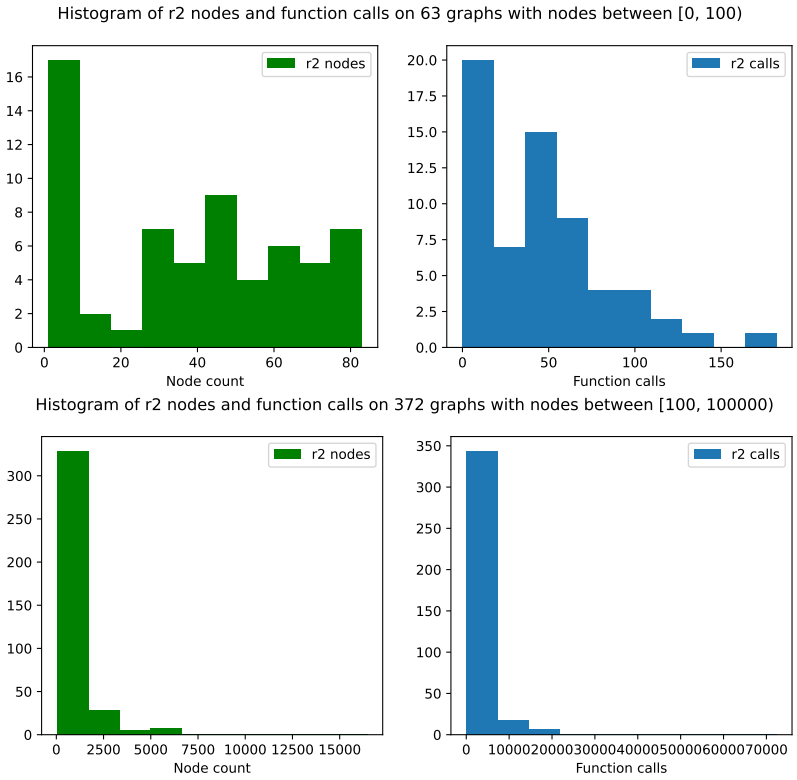


FIGURE 4. Histogram of call graph sizes using Radare2 (nodes and calls).

be mentioned here that this does not mean that Radare2 obtains different nodes than IDA – it can happen that the nodes are assigned other labels, but their instruction content may still be the same. This is a key aspect, which highlights the fact that the real similarity between IDA Pro and Radare2 scan results are higher than the values measured.

Fig. 6 aims to show us the size of the local subroutines in the graphs – i.e. the length of the assembly instruction list in a subroutine. One can observe that the average and median values (around $100 - 200$) are not so much affected by the size of the graphs, but the maximum values are heavily affected ($6000 - 20,000$): the larger the graph in node count, the longer its functions may become. This may be an unwanted effect of metamorphic malware samples, which fill their sections with garbage code from one generation to another.

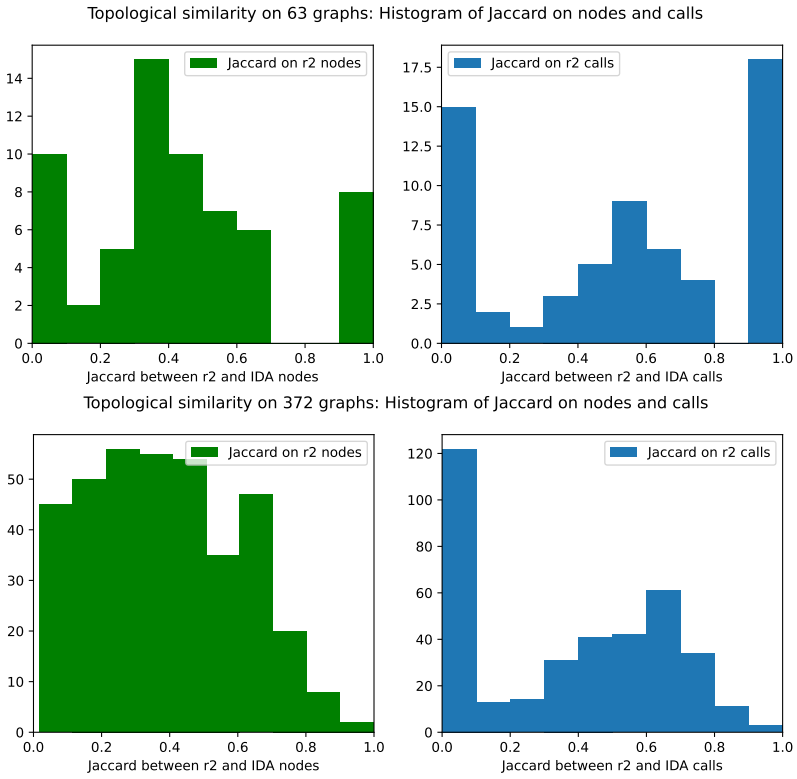


FIGURE 5. Topological similarity: histogram of Jaccard score between IDA and Radare2 nodes and calls.

Figures 7, 8, 9, 10 show histograms of the distance and similarity metrics between the instruction mnemonic lists of the nodes within the call graphs obtained with IDA and Radare2. In Fig. 7 we can see one of the most valuable conclusions of this paper: while call graphs grow, their nodes' instruction list grow, the Levenshtein distances' average value still remains fixed in the range of $10 - 20$. This conclusion is reinforced by Fig. 8, where relative Levenshtein similarities converge to 1 even in the case of large graphs. This observation remains valid in the case of the remaining plots, shown in Figures 9 and 10, depicting the histograms of Jaro and Jaro-Winkler similarity scores, respectively. The fact that the median values, especially the 75% percentile values are close to 1 means that even though the content of the functions may change from IDA to Radare2 and vice-versa, this change is insignificant.

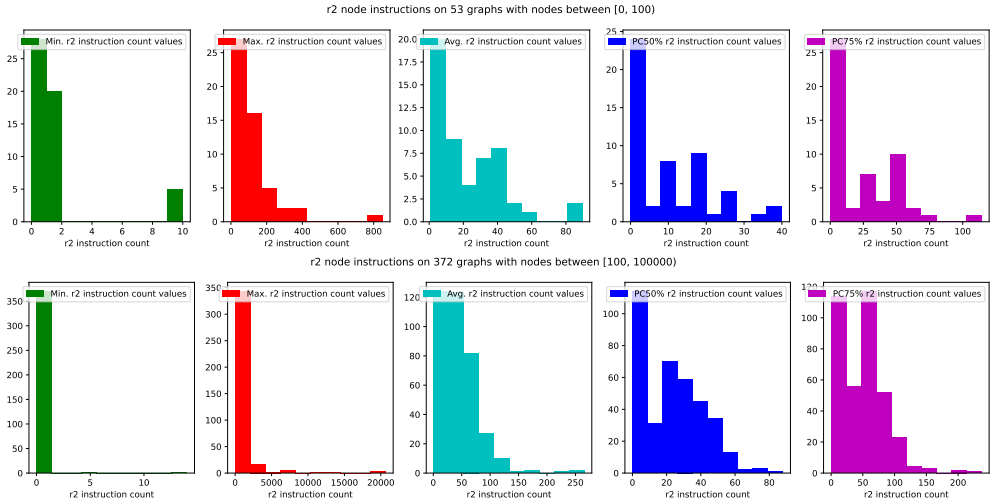


FIGURE 6. Histogram of Radare2 nodes' instruction count.

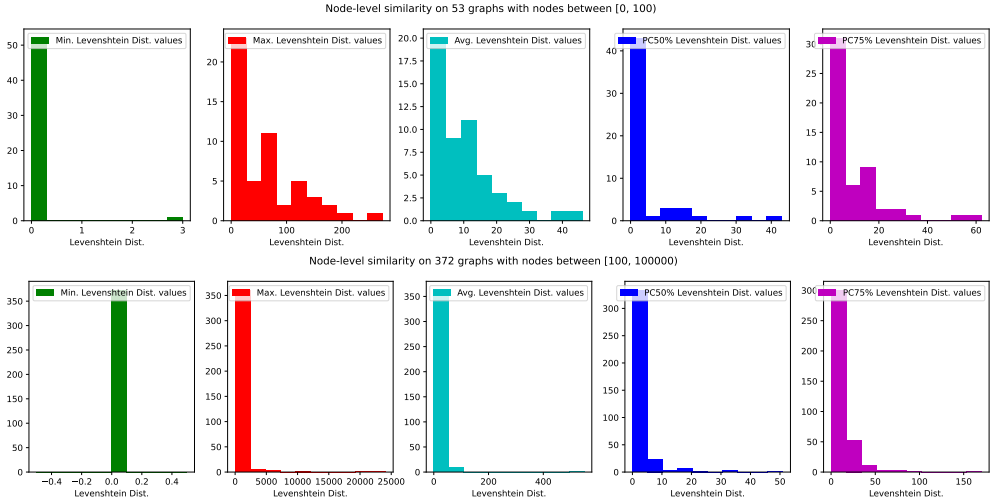


FIGURE 7. Histogram of Levenshtein distances between Radare2 and IDA nodes.

5. Conclusions and future work

This paper presents a novel comparison between IDA Pro 6 and Radare2 disassembler tools, by analyzing a dataset of malicious files using both of these, and comparing their output. The subject of the analysis is the static

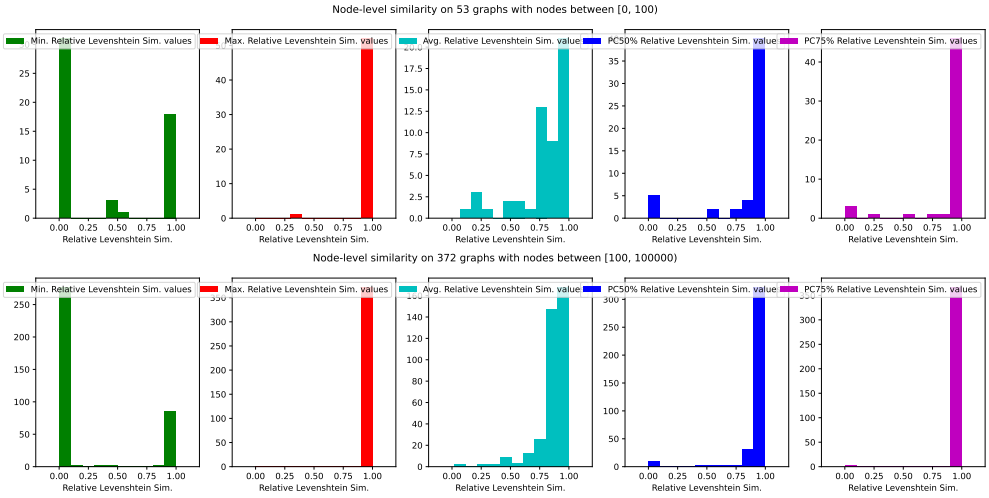


FIGURE 8. Histogram of relative Levenshtein similarities between Radare2 and IDA nodes.

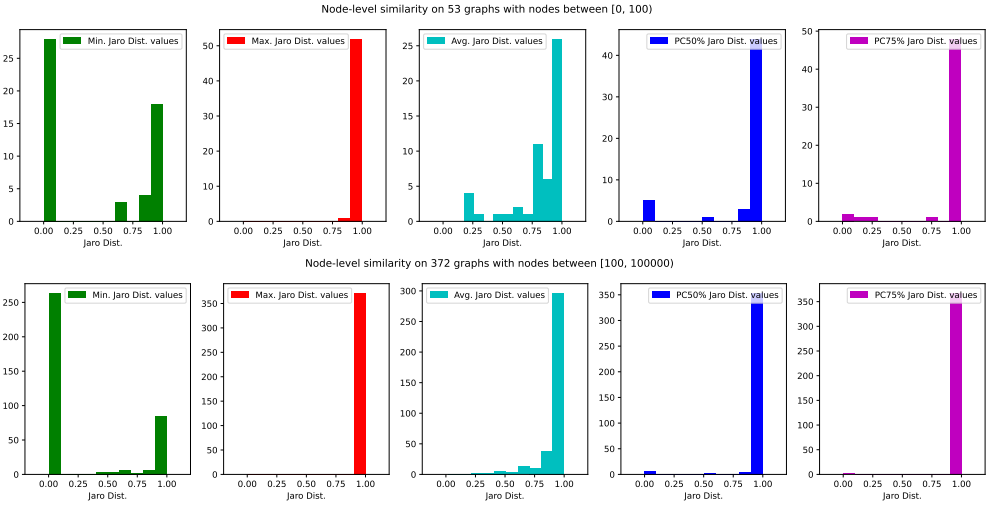


FIGURE 9. Histogram of Jaro distances between Radare2 and IDA nodes.

call graph, which is generated by using the tools' scripted APIs and processing the output to create the final, global call graph. In the experiments, a public dataset is used in order to offer full transparency of the results. The call graphs are compared from various perspectives, both topological aspects i.e.

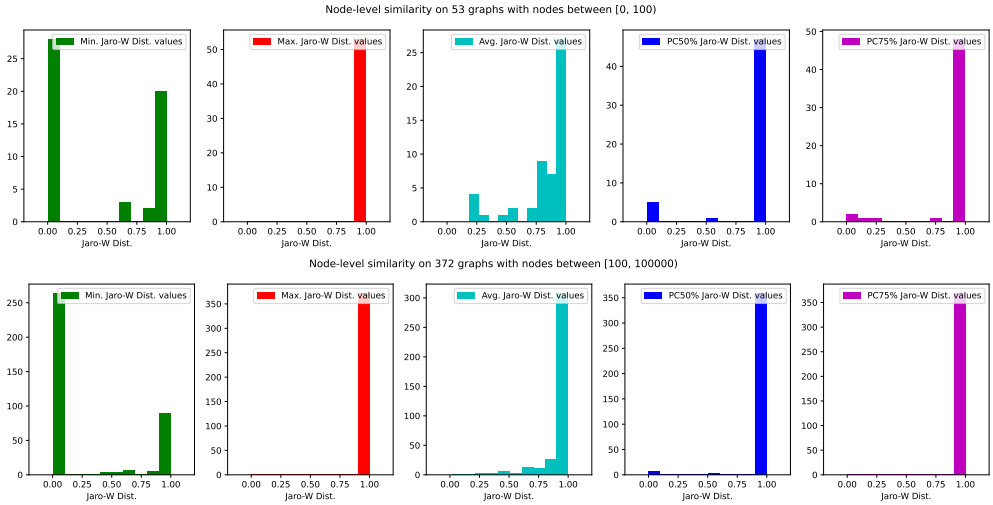


FIGURE 10. Histogram of Jaro-Winkler distances between Radare2 and IDA nodes.

the function calls, and also node-level criteria i.e. the instruction list of each subroutine. Our results claim that there is no significant change in the output of IDA and Radare2 disassemblers, however, the latter offers a faster, more stable way of scripted analysis which is suitable for a production environment where performance is a key aspect.

Future ideas include and are not limited to the use of Radare2 in order to analyze the call graphs of a larger dataset, with the aim of attribution classification, clustering, or other threat intelligence retrieval.

ACKNOWLEDGEMENTS

This project was supported by Bitdefender, offering the infrastructure for malware analysis –special thanks to my colleagues, Ovidiu Ardelean and Adrian Nandrean, for helping in the dataset collection process.

I want to thank my scientific tutor, dr. Zalán Bodó, for all his assistance during our work.

References

[1] Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., and Bos, H. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium* (2016), pp. 583–600.

[2] Bai, J., Shi, Q., and Mu, S. A malware and variant detection method using function call graph isomorphism. *Security and Communication Networks 2019* (2019), 1–12.

[3] Cohen, I. Deobfuscating apt32 flow graphs with cutter and radare2. Tech. rep., 2019.

[4] Cunningham, E., Boydell, O., Doherty, C., Roques, B., and Le, Q. Using text classification methods to detect malware. In *AICS* (2019).

[5] Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), IEEE, pp. 3422–3426.

[6] del Pilar Angeles, M., and Gamez, A. E. Comparison of methods hamming distance, jaro, and monge–elkan. *DBKDA 2015* (2015), 73.

[7] Elhadi, A. A. E., Maarof, M. A., and Barry, B. I. Improving the detection of malware behaviour using simplified data dependent api call graph. *International Journal of Security and Its Applications 7*, 5 (2013), 29–42.

[8] Faruki, P., Laxmi, V., Gaur, M. S., and Vinod, P. Mining control flow graph as api call-grams to detect portable executable malware. In *Proceedings of the Fifth International Conference on Security of Information and Networks* (2012), pp. 130–137.

[9] Gibert, D., Mateu, C., and Planes, J. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications 153* (2020), 102526.

[10] Jaro, M. A. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association 84*, 406 (1989), 414–420.

[11] Jiang, H., Turki, T., and Wang, J. T. Dlgraph: Malware detection using deep learning and graph embedding. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)* (2018), IEEE, pp. 1029–1033.

[12] Kilgallon, S., De La Rosa, L., and Cavazos, J. Improving the effectiveness and efficiency of dynamic malware analysis with machine learning. In *2017 Resilience Week (RWS)* (2017), pp. 30–36.

[13] Kinable, J., and Kostakis, O. Malware classification based on call graph clustering. *Journal in computer virology 7*, 4 (2011), 233–245.

[14] Koo, H., Park, S., and Kim, T. A look back on a function identification problem. In *Annual Computer Security Applications Conference* (2021), pp. 158–168.

[15] Levenshtein, V. I., et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (1966), vol. 10, Soviet Union, pp. 707–710.

[16] Massarelli, L., Di Luna, G. A., Petroni, F., Baldoni, R., and Querzoni, L. Safe: Self-attentive function embeddings for binary similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (Cham, 2019), Springer International Publishing, pp. 309–329.

[17] Mester, A. Scalable, real-time malware clustering based on signatures of static call graph features. Master's thesis, Babeş–Bolyai University, Faculty of Mathematics and Computer Science, Cluj-Napoca, Romania, 2020.

[18] Mester, A., and Bodó, Z. Validating static call graph-based malware signatures using community detection methods. In *Proceedings of ESANN* (2021).

[19] Mester, A., and Bodó, Z. Malware classification based on graph convolutional neural networks and static call graph features. In *Advances and Trends in Artificial Intelligence. Theory and Practices in Artificial Intelligence: 35th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2022, Kitakyushu, Japan, July 19–22, 2022, Proceedings* (2022), Springer, pp. 528–539.

[20] Nar, M., Kakisim, A. G., Yavuz, M. N., and Soğukpinar, İ. Analysis and comparison of disassemblers for opcode based malware analysis. In *2019 4th International Conference on Computer Science and Engineering (UBMK)* (2019), IEEE, pp. 17–22.

[21] Org., R. The official radare2 book. https://book.rada.re/.

[22] Park, Y., Reeves, D., Mulukutla, V., and Sundaravel, B. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research* (2010), pp. 1–4.

[23] Pektaş, A., and Acarman, T. Deep learning for effective android malware detection using api call graph embeddings. *Soft Computing 24* (2020), 1027–1043.

[24] Priyanga, S., Suresh, R., Romana, S., and Shankar Sriram, V. The good, the bad, and the missing: A comprehensive study on the rise of machine learning for binary code analysis. In *Computational Intelligence in Data Mining: Proceedings of ICCIDM 2021*. Springer, 2022, pp. 397–406.

[25] Shaila, S., Darki, A., Faloutsos, M., Abu-Ghazaleh, N., and Sridharan, M. Disco: Combining disassemblers for improved performance. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (2021), pp. 148–161.

[26] Singh, A., Arora, R., and Pareek, H. Malware analysis using multiple api sequence mining control flow graph. *arXiv preprint arXiv:1707.02691* (2017).

[27] Steffens, T. *Attribution of Advanced Persistent Threats*. Springer, 2020.

[28] Ucci, D., Aniello, L., and Baldoni, R. Survey of machine learning techniques for malware analysis. *Computers & Security 81* (2019), 123–147.

[29] Wenzl, M., Merzdovnik, G., Ullrich, J., and Weippl, E. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR) 52*, 3 (2019), 1–37.

[30] Winkler, W. E. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage.

[31] Yin, X., Liu, S., Liu, L., and Xiao, D. Function recognition in stripped binary of embedded devices. *IEEE Access 6* (2018), 75682–75694.

Faculty of Mathematics and Computer Science, Babeş–Bolyai University of Cluj-Napoca

Email address: attila.mester@ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 1, 2023
DOI: 10.24193/subbi.2023.1.02

MALICIOUS WEB LINKS DETECTION - A COMPARATIVE ANALYSIS OF MACHINE LEARNING ALGORITHMS

COSTE CLAUDIA-IOANA

Abstract. One of the most challenging categories of threats circulating into the online world is social engineering, with malicious web links, fake news, clickbait, and other tactics. Malware URLs are extremely dangerous because they represent the main propagating vector for web malware. Malicious web links detection is a challenging task because the detection mechanism should not influence the consumers' online experience. The proposed solutions must be sensitive enough, and fast enough to perform the detection mechanism before the user accesses the link and downloads its content.

Our paper proposes three goals. The main purpose of this paper is to refine a methodology for malicious web links detection that may be used to experiment with machine learning algorithms. Moreover, we propose to use this methodology for training and comparing several machine learning algorithms such as Random Forest, Decision Tree, K-Nearest Neighbor. The results are compared, justified, and placed in the malicious web links literature. In addition, we propose to identify the most relevant features and draw some observations about them.

1. Introduction

Starting with the early 2000, most services: media and news, education, public administration, shopping etc. have moved their content, and customers online. Now, almost every household with an Internet connection needs to surf the Internet to satisfy its basic needs. Thus, consumers are more susceptible to becoming victims of malicious links and web-malware in general. Malicious web links are used to trick the users into giving away personal information. Through malicious links, consumers may be compelled to give access

Received by the editors: 1 March 2023.

2010 *Mathematics Subject Classification.* 68T99, 68U99.

1998 *CR Categories and Descriptors.* 68T99 [**Artificial Intelligence**]: Applications and Expert Systems – *Experiments for Malicious Web Links Detection*; 68U99 [**Management of Computing and Information Systems**]: Security and Protection – *Malicious Web Links*.

Key words and phrases. malicious web links detection, web-malware, artificial intelligence.

to their computer's resources or to consume low-quality and fake web content, increasing the incomes of content providers with views, clicks, and page visits. According to Cofense [4], in 2021, 38% of all phishing emails analyzed contained a malicious link. In addition to that, shortened URLs have become a real threat, since they are difficult to be identified as malicious [5]. Moreover, as stated by [4], 50% of credential phishing attacks were done using *.com* domains and 84% of all phishing sites use SSL and HTTPS protocol [2].

A web link represents a Uniform Resource Locator (URL), an identifier for a web page resource. A malicious link can be an attack vector for many types of threats such as: phishing attacks, cross-site request forgery (CSRF), cross-site scripting (XSS), drive-by-downloads, redirections without user's consent to cloned web pages etc. Most attacks target credentials theft, which may lead to important data breaches, and it may have financial advantages for the attackers.

Our identified problem in the domain of malicious web links is to detect malware or benign links. The problem is a binary classification with the URL as input and the class (malicious or benign) as output. The problem definition is mathematically defined as: $f : URLs \rightarrow \mathbb{R}^d$, f(url) = ($x_1$, x_2, x_3, ..., x_d), where d is the number of features and x_d, for each d $\in \mathbb{N}$ is a feature. We address the problem of malicious web links detection with three machine learning (ML) algorithms: K-Nearest Neighbor (KNN), Decision Tree (DT), and Random Forest (RF). The experiments made follow the next steps: parameters calibration and feature importance for DT and RF. We propose a methodology for refining the parameters values through experiments. Moreover, we would like to compare our models and analyze how they relate with other solutions presented in literature, especially with models proposed by Islam et al. [7], since it is using the same dataset. Another aim for the present paper is to investigate the feature importance in the case of DT and RF models.

The present article is structured in the following four sections. Malicious Web Links Detection presents the previous work done in our research niche. Proposed Methodology contains relevant details about the methodology we followed when driving the experiments. Results and Discussions has our metrics, comparisons and critical analysis on the experiments delivered. Conclusions and Future Work draws the final conclusions and discusses future directions of research.

2. Malicious Web Links Detection

Most previous work done in the malicious web links detection field can be split into two categories: dynamic and static. The dynamic approaches involve malicious code execution, and the static ones predict maliciousness of a link

based on features, without code execution. In the execution-based category, there are included honey pots, sandbox-based systems and ML approaches that use features related to the code execution. Static detection systems contain blacklists solutions, signature-based ones, approaches based on complex networks and ML solutions taking into consideration static features.

The features used for link classification can be split into blacklists features, host-based ones, lexical characteristics, and content-based [14]. Blacklists features are extracted from public blacklists and lists with trusted domains. Host-based characteristics are usually extracted through web crawling and are related to IP address, port, protocol, HTTP/HTTPS headers, WHOIS and DNS information, geo-location etc. Lexical features involve traditional lexical properties (e.g., Bag-of-Words, N-grams) [14]. Moreover, lexical features may also involve the number of different special characters, number of words, URL length, query parameters number, domain name, etc. Content-based features are formed by characteristics extracted from the web page content, mostly code: HyperText Markup Language (HTML), Cascading Style Sheets (CSS), and JavaScript. In addition, content-based features include visual features, used for catching consumers' attention and metadata characteristics used for search engine optimization.

There are more problem types when discussing malicious web links detection. Some approaches consider a binary classification (most approaches) and others a multi-class problem ([8, 16]). There are articles ([10], [11]) testing the proposed models across multiple datasets to prove their adaptability.

Regarding multi-class solutions, Johnson et al. [8] experiments with RF, DT, KNN, Support Vector Machine (SVM), Logistic Regression (LR), etc., and two deep learning models developed with Fast.ai and TensorFlow-Keras. Best results for both multi-class and binary-class prediction are obtained by RF and the two deep models. Finally, RF is seen as the most suitable option since it does not require many computational resources. Tung et al. [16] is solving the multi-class problem as well, having four classes: benign, spam, malware, and phishing. The classification step is done using DT and RF models. By comparison, the RF algorithm outperforms the DT model with an accuracy of 97.49% for each class. They concentrate their effort in the feature selection process, where the solutions prove an improvement when adding three host-based features.

The classification done by Oshingbesan et al. [11] is binary and it is an experiment across multiple different datasets with multiple artificial intelligence algorithms (SVM, DT, RF, KNN etc.). According to [11], KNN is the most robust model that achieved a high performance in a cross-dataset environment. Similarly, Naveen et al. [10] is implying usage of tree-based models (DT, RF)

and other algorithms (LR, Linear SVM, KNN etc.) for experiments across multiple datasets. The model that distinguishes its performance is KNN.

Taking into consideration binary classification with multiple ML models, Shantanu et al. [15] is using many ML models for experiments: SVM, DT, RF, KNN, LR, Naive Bayes (NB), and Stochastic Gradient Descent. The best metrics are obtained by RF. Similarly, Ibrahim et al. [6] analyzes malicious websites that deliver drive-by downloads attacks, using NB, JRip and J48. In the same direction of research, Catak et al. [3] develops two models, a RF with default parameters and a Gradient Boosting classifier. The best accuracy of 98.6% is obtained by RF, which proves to be faster than its counterpart algorithm. Implying multiple ML algorithms, Pakhare et al. [12] is using: LR, SVM with linear, RBF, and sigmoid kernels, KNN, RT and DT. But besides this, there are proposed ensembles formed out of these algorithms. The best performance is achieved by the ensemble formed with DT, KNN, and SVM. Islam et al. [7] is proposing solutions on the dataset found in [17] with the following ML algorithms: KNN, DT, RF and Multilayer Perceptron. They achieved 90% F1 score for KNN model, 83% for Neural Net and 99% for DT and RF model.

3. Methodology

Our research purpose is to propose an experimental methodology for approaching the malicious web links detection problem. This methodology will be exemplified with three ML algorithms: KNN, DT and RF. We aim to calibrate their parameters, accordingly, compare them and analyze the feature importance obtained. The methodology contains the next steps: dataset selection, pre-processing, counteracting the imbalancement problem, the classification step, which includes in total three phases of experiments for parameter calibration and feature importance. The methodology will be detailed in the next paragraphs.

The dataset we selected is a free available dataset [17], having 1781 records (216 malicious, 1565 benign). We propose to first experiment with little data such that we could easily get an insight on how we should approach this malicious links classification problem, without involving many computation resources. Moreover, the dataset has 17 already extracted attributes: lexical (URL length, number of special characters) and host-based ones (WHOIS & DNS information, content length, charset, server, number of ports open on the server, TCP packets count, number of bytes transported over the network, number of IPs connected to the server etc.).

Next step was preprocessing the data. Firstly, we cleared the records from missing or not defined values. Categorical features were transformed into

numbers using supervised ratio algorithm (total number of samples with the category present in the positive class divided by the total number of records [7]) and weight of evidence algorithm (equation 1 [7]). The positive class is annotated with 1, indicating the malicious records, while the negative one is labeled with 0.

$$
\begin{aligned}
&X_{new} = \ln \frac{\frac{P_i}{TP}}{\frac{N_i}{TN}}; where \\
&P_i \ - \ number\ of\ records\ with\ positive\ class\ value\ for\ the \\
&categorical\ attribute\ value\ in\ dataset; \\
&N_i \ - \ number\ of\ records\ with\ negative\ class\ value\ for\ the \\
&categorical\ attribute\ value; \\
&TP \ - \ total\ number\ of\ records\ with\ positive\ class\ value; \\
&TN \ - \ total\ number\ of\ records\ with\ negative\ class\ value.
\end{aligned} \tag{1}
$$

As the final preprocessing step, data normalization was done using two implementations: the Min-Max and Standard Scaler from Sklearn library [13]. The Min-Max Scaler normalizes the data on feature range (0,1). The Standard Scalar is transforming data based on the difference between data and the mean of data divided by the standard deviation [13]. Even though, Islam et al. [7] is not mentioning a normalization step, we chose to add it since we consider it to be important to balance the values in our dataset. For instance, timestamps for datetime features are represented by a long number, while the URL length is represented with a number between 16 and 33.

Since the data of the dataset is unbalanced, the problem is approached by using specific metrics: ROC-AUC score, precision, recall and F1 score. Moreover, when splitting the data into training and testing sets, it was parameterized with the *stratify* argument to keep the initial ratio between the two classes. In addition to this, the *class weight* parameter was used for DT and RF models. This parameter takes into consideration the imbalancement of the data when executing the algorithm and making a node split.

The algorithms we have chosen for this experimental research were KNN, DT and RF. We considered these algorithms because they are efficient, and do not require a lot of configuration and training time. Moreover, these algorithms are often used in other literature articles proposing solutions for malicious web links detection.

3.1. **K-Nearest Neighbor.** KNN is a statistical algorithm, used in [7], where it achieves an F1 score of 90%. Shantanu et al. [15] is using KNN along other ML algorithms, yet all models are outperformed by RF. Johnson et al. [8] is experimenting with multiple algorithms, including KNN, which scores 97.47 accuracy for the binary classification. In [11] and [10], KNN proves to be the

most flexible and adaptive model across multiple datasets. Pakhare et al. [12] uses KNN in the best performing ensemble model, including SVM and DT. This ensemble is the one outperforming the others with an accuracy of 94.93%.

3.2. **Decision Tree.** A DT model is a directed connected acyclic graph with a root node, child nodes and leaves. In [7], DT achieves a 99% accuracy, being one of their best models. Also, DT is used in the best ensemble model with SVM and KNN in [12]. Johnson et al. [8] experiments with DT for binary classification and achieves an accuracy of 97.63. For Oshingbesan et al. [11] and Naveen et al. [10], DT performs well for classifying links based on lexical features but is not the best performing model. Tung et al. [16] achieves a multi-class accuracy of 96% for the DT model.

3.3. **Random Forest.** RF is an ensemble learning method characterized by a voting system and formed with multiple Decision Trees. For Islam et al. [7], RF is one of the best models deployed, considering their experiments. Adas et al. [1] employs a Decision Forest reaching 99.8% accuracy. Similarly, as in the DT case, RF model achieves good accuracy for lexical features according to Naveen et al. [10]. In [8], RF reaches a performance of 98.68 accuracy for the binary classification case and outperforms more complex models. RF's multi-class accuracy is 96.26, being close to the accuracy of the fast.ai model. In the experiments done by Pakhare et al. [12], RF has the worst accuracy of 87.34%. Tung et al. [16] is deploying a RF model, with 97.49% accuracy, surpassing its comparing algorithm (DT). Catak et al. [3] get its best performance with a RF (98.6% accuracy).

The implementations of the ML algorithms (KNN, DT, RF) used in our experiments were from Sklearn Python Library [13], version 1.0.2. For running the experiments, we used Jupyter notebooks with Google Colab and the Python version was 3.7.15.

The experiments done followed three stages. The first and second stages of experiments consisted of calibrating the parameters for all algorithms. The values used for parameters were chosen based on a preliminary set of experiments (first stage). These experiments were made using as a starting point the configuration provided by Islam et al. [7], then we varied each parameter at a time. The resulting configuration was sorted based on an average metric computed as the average mean of all metrics (precision, recall, F1 score and ROC AUC score * 100). We chose the best 20 configurations and extracted from them the parameters and values relevant. Next, for the selected parameters and values we tried all the combinations possible (second stage of experiments).

	Model by [7]	**Preliminary stage**	**Second stage**
scaler	not mentioned	Min-Max & Standard	Min-Max & Standard
n neighbors	5 (default)	{1, 2, ..9} ∪ {10, 20, . . . , 90}	{1, 2, .., 10 }
weights	uniform (default)	{uniform, distance}	{distance}
algorithm	auto (default)	{brute, ball tree, kd tree, auto}	{ball tree, kd tree, auto}
leaf size	30	{1, 6, . . . 16} ∪ {20, 39} ∪ {40, 50, . . . , 90}	{1,3, .., 13} ∪ {20, 22, .., 34} ∪ {43, 44, 45, 46} ∪ {50, 51, .., 56} ∪ {65, 66, .., 69} ∪ {85, 86, .., 91}
p	2	{1, 2, . . . , 10} (with metric = minkowski)	{1, 2, . . . , 10} (with metric = minkowski)
metric	minkowski	{euclidean, chebyshev, manhattan, minkowski }	{manhattan, euclidean, minkowski}
		Configurations: 376	**Configurations: 68400**

TABLE 1. KNN algorithm with the parameters we calibrated and their values

	Model by [7]	**Preliminary stage**	**Second stage**
scaler	not mentioned	Min-Max & Standard	Min-Max & Standard
criterion	gini	{gini, entropy}	{gini, entropy}
min samples leaf	32	{1, 2, . . . 99}	{1, 2, .., 20} ∪ {30, 31, . . . , 35}
min samples split	2 (default)	{2, 22, 42, . . . , 1762}	{2,3, .., 10}
max depth	12	{1, 2, . . . 99}	{5, 6, .., 15}
max leaf nodes	None (default)	{2, 4, . . . , 98}	{18, 20, 22, 24}
max features	None (default)	{1, 2, . . . , 19} ∪ {log2, sqrt, auto}	None (default) ∪ sqrt
min weight fraction leaf	0.0 (default)	{0, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5}	0.0 (default)
min impurity decrease	0.0 (default)	{0, 20, 40, . . . , 1780}	0.0 (default)
ccp alpha	0.0 (default)	{0.0, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.75, 0.8, 0.9, 1}	0.0 (default)
		Configurations: 1882	**Configurations: 82368**

TABLE 2. DT algorithm with the parameters we calibrated and their values

In the tables 1, 2 and 3, there are aggregated all parameters for each algorithm and the values chosen for both stages of the experiments. Finally, the best configuration was chosen out of the top 20 configurations resulting from the second stage of experiments, which were run multiple times. The scores for the best configuration are computed as an average of all execution scores.

The last stage of our experiments was about analyzing feature importance in the case of the DT or RF model. The feature importance was computed on the best model with Mean Decrease in Impurity (MDI) and Mean Decrease Accuracy (MDA) formulas, both being integrated in Sklearn library [13]. We proposed to analyze the feature importance such that we could give relevant

	Model by [7]	Preliminary stage	Second stage
scaler	not mentioned	Min-Max & Standard	Min-Max & Standard
n estimators	100	{1, 11, . . . , 71} ∪ {80, 85, . . . , 115} ∪ {120, 130, . . . , 240} ∪ {250, 275, . . . , 475} ∪ {500, 550, . . . , 950}	{50, 51, 90, 100, 110}
criterion	gini	{gini, entropy}	{gini, entropy}
min samples leaf	32	{1, 6, . . . 96}	{1, 3, 5} ∪ {11, 13, . . . , 21} ∪ {32}
min samples split	2 (default)	{2, 3, 23, 43, . . . , 1763}	{2, 4, .., 10}
max depth	12	{None} ∪ {1, 2, . . . , 15} ∪ {16, 18, . . . , 98}	{10,11, . . . , 15} ∪ {50, 51, . . . , 54}
max leaf nodes	None (default)	{2, 22, 42, . . . , 1762}	{None, 680, 681, 682, 683, 684, 1560, 1561, 1562, 1563, 1564}
max features	sqrt (default)	{1, 3, . . . , 19} ∪ {log2, sqrt}	sqrt (default)
min weight fraction leaf	0.0 (default)	{0, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5}	0.0 (default)
min impurity decrease	0.0 (default)	{0, 20, 40, . . . , 1780}	0.0 (default)
bootstrap	True (default)	{true, false}	True (default)
oob score	False (default)	{true, false}	False (default)
ccp alpha	0.0 (default)	{0.0, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.75, 0.8, 0.9, 1}	0.0 (default)
max samples	None (default)	{0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1}	None (default)
		Configurations: 1816	**Configurations: 99000**

TABLE 3. RF algorithm with the parameters we calibrated and their values

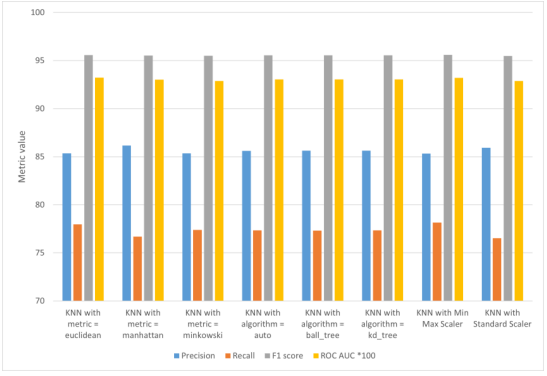
and interesting tips based on how to spot the malicious link and explain how the tree models work in links classification.

4. Results and discussions

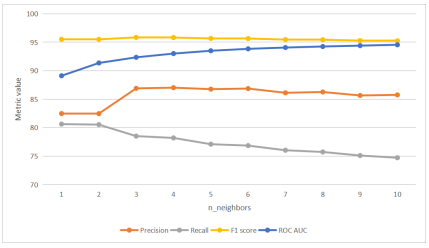
In the current section, we will present the results for all our experiments described in Section 3. The first stage of experiments was the preliminary phase. The second stage of experiments was represented by the parameter's calibration. Both phases were carried out for all three algorithms. In the next paragraphs, we will present and analyze the results obtained for each algorithm. For the preliminary stage we ran 376 configurations for KNN, 1882 for DT and 1816 for RF. Based on the results obtained from these experiments we chose the interval values for relevant parameters in our model. The values and parameters were presented in detail in Tables 1, 2 and 3.

4.1. **Results obtained for KNN.** KNN achieved the best on average performance (see Table 4). The best configuration of KNN was *weights = distance, metric = Euclidean, algorithm = ball tree, n neighbors = 3, leaf size= 86,*

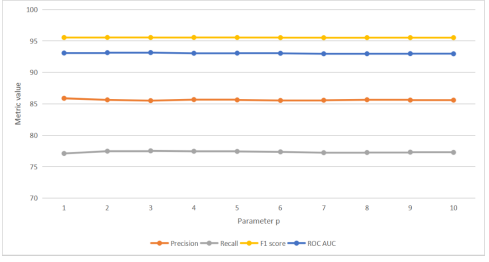
scaler = Min-Max. Parameter calibration for KNN is presented in Figures 1. As observed for DT and RF as well, there was little to no difference between the normalization methods (Figure 1a). From some of the graphics associated with the calibration of parameters it cannot be determined the best value for our model (Figures 1d, 1a, and 1c). Best KNN had the distance value for *weights*, representing that closer neighbors have a higher influence on the classification task. The Euclidean metric got the best recall and F1 score, while the Manhattan metric had the highest precision and Minkowski reached the best ROC AUC score as can be observed in Figure 1a. In the case of the Euclidean metric, the value of the p parameter is not taken into consideration. Ball tree algorithm got the best precision, but the difference to the other algorithm was not significant. On the other hand, the *n neighbors* value could be deducted from the Figure 1b, value 3 achieving the highest precision. *Leaf size* is relevant for the algorithm ball tree chosen, but its optimal value is hard to be observed in Figure 1d.



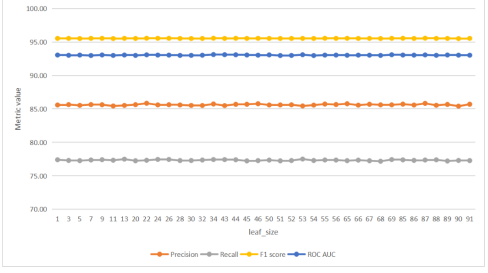
(A) Calibrating algorithm & metric & scaler



(B) Calibrating n neighbors



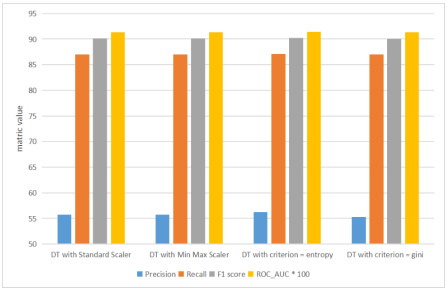
(C) Calibrating p



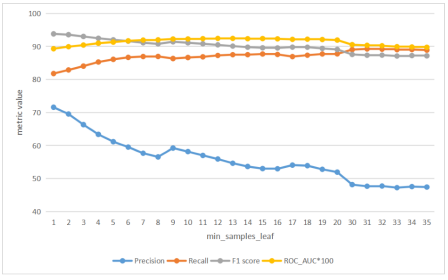
(D) Calibrating leaf size

FIGURE 1. KNN - calibrating parameters

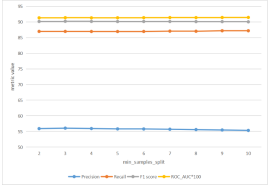
4.2. **Results obtained for DT.** The results for DT confirm that there was little to no difference in using the Min-Max or the Standard Scaler for normalization, as seen in Figure 2a. Considering the *criterion*, in Figure 2a it can be observed that entropy was the better option. In figures 2d, 2c and 2e the problem of choosing the best parameter is quite difficult, not much of a difference between each value. On the contrary, Figure 2b is presenting the best option to be chosen as parameter, the precision score being very sensitive to the variations of *min samples leaf* parameter, where the optimum value was 1. The graphics obtained are strengthened by the selection of the best DT model configuration. This configuration involved: *criterion = entropy, min samples leaf = 1, max depth = 8, max leaf nodes = 20, min samples split = 7, scaler = Standard, class weight = balanced, max features = None.*



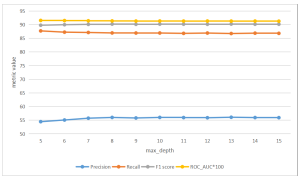
(A) Calibrating scaler & criterion



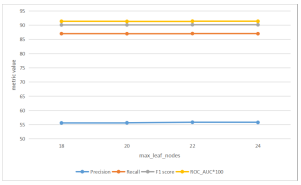
(B) Calibrating min samples leaf



(C) Calibrating min samples split



(D) Calibrating max depth

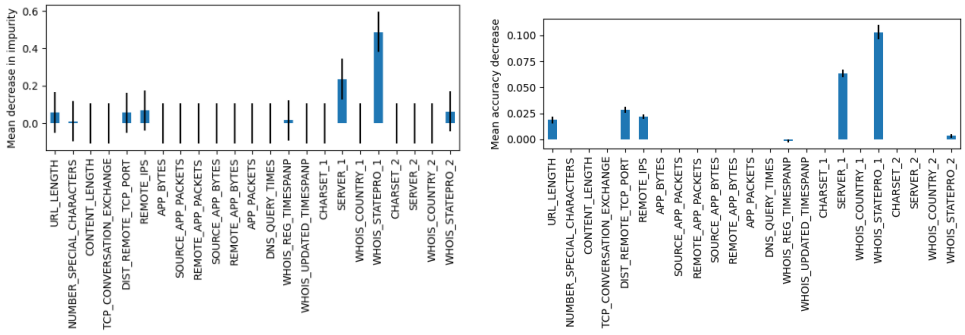


(E) Calibrating max leaf nodes

FIGURE 2. DT - calibrating parameters

Considering feature importance analysis, the results for the DT model are presented in Figure 3a for MDI and in Figure 3b for MDA. Features suffixed with "_1" are categorical features preprocessed with the supervised ratio algorithm. The ones suffixed with "_2" are categorical features transformed with weight of evidence algorithm (Equation 1). Taking into consideration the results, we observed a high relevancy for host-based features such as: *SERVER* and *WHOIS STATEPRO*. *SERVER* refers to the operation system running

on the Web Server. *WHOIS STATEPRO* is represented by the state (approximate geo-location) from where the server responded to the request. Thus, malicious link attacks might be a geographically segregated attack. *SERVER* feature indicates importance since an outdated operating system can be more vulnerable to threats. It can be observed that categorical features processed with the supervised ratio algorithm reached a higher degree of importance. From the lexical features, *URL LENGTH* was the most important one, its score being comparable with other host-based characteristics: *DIST REMOTE TCP PORT* and *REMOTE IPS*. These last two attributes refer to the number of ports detected to be opened on the server (excluding the current connection port when data was collected) and respectively, the number of IPs addresses connected to the web server. These two attributes were relevant since they show a high activity on the server.

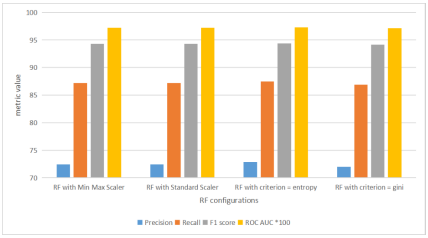


(A) Mean Decrease in Impurity (MDI) (B) Mean Decrease Accuracy (MDA)

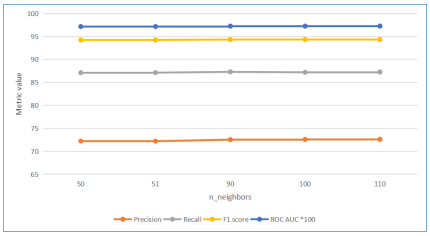
FIGURE 3. DT - feature importance

4.3. **Results obtained for RF.** RF on average managed to improve performance compared to DT (Table 4). The best calibration of RF model was: *criterion = entropy, n estimators = 110, max depth=53, min samples leaf=3, max leaf nodes=1564, min samples split=4, class weight=balanced, scaler = Min-Max*; which outperformed the others. The best model configuration was confirmed by the graphics presenting parameters calibration, even though in the cases of *max depth*, *max leaf nodes*, *min samples split* and *n estimators* as can be observed from Figures 4e, 4f, 4d and respectively Figure 4b, there was little to no difference between the parameters values. Considering the *criterion* used, entropy was the value scoring the highest value along all metrics as seen in Figure 4a. Entropy was the value used in the best model calibration as well. Considering the *scaler* used for the normalization step, just the

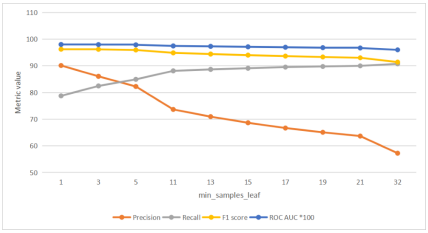
same conclusion here as for KNN model or DT model, there was almost no difference between them. Still, the best performance was achieved using Min Max scaler, which scored a higher precision and F1 score than its counterpart. Regarding the calibration of *min samples leaf* parameter, as can be seen quite easily in Figure 4c the value, which significantly increased the recall is 3, this one being used in the best model configuration.



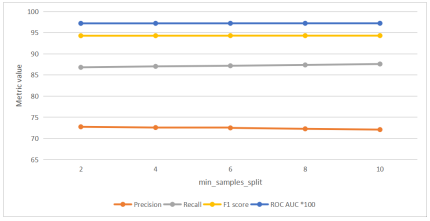
(A) Calibrating scaler & criterion



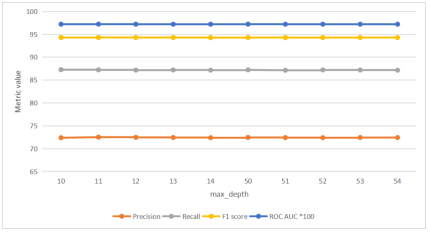
(B) Calibrating n estimators



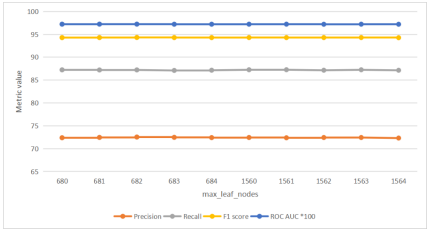
(C) Calibrating min samples leaf



(D) Calibrating min samples split



(E) Calibrating max depth

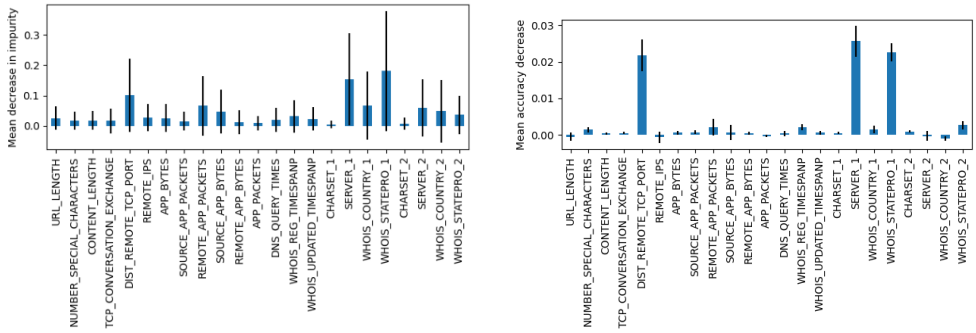


(F) Calibrating max leaf nodes

FIGURE 4. RF - calibrating parameters

The scores of the features were computed on the best RF model configuration. Results for the RF model showed us that the importance score is more distributed along more features, unlike the DT model case. Feature scores were computed based on the MDI (Figure 5a) and MDA (Figure 5b). Similarly, as in the case of the DT model, categorical features were transformed in

numbers using the supervised ratio algorithm and the weight of evidence algorithm. Best features were *SERVER*, *WHOIS STATEPRO*, *DIST REMOTE TCP PORT*, all of them being host-based features. *SERVER* represents the operating system of the web server serving the HTTP request, which is relevant to predict the vulnerability of an outdated system. *WHOIS STATEPRO* refers to the approximate geo-location of the server and it was expected to be relevant since most cyber-attack are geographically segregated. *DIST REMOTE TCP PORT* counts the open ports waiting for a connection on the web server and it shows a high activity on the server, maybe even serving multiple web pages. From the lexical features, *URL LENGTH* was the attribute achieving the highest score.



(A) Mean Decrease in Impurity (MDI) (B) Mean Decrease Accuracy(MDA)

FIGURE 5. RF - computing feature importance

	Precision	Recall	F1 score	ROC AUC * 100
KNN on average	85.62	77.33	95.53	93.03
Best KNN	86.6	80.43	96.02	92.78
DT on average	55.73	87.01	90.11	91.35
Best DT	75.23	83.62	94.7	90.51
RF on average	72.43	**87.19**	94.29	97.21
Best RF	**87.6**	82.55	**96.39**	**98.08**

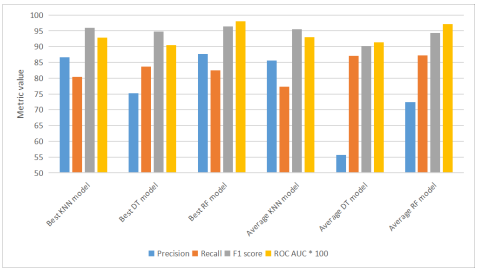
TABLE 4. Results for KNN, DT and RF models

4.4. **Discussions.** From the experiments done and the statistics made, we observed that precision is a highly sensitive metric to parameters' calibration. We considered it to be highly relevant for our binary classification because it was computed on the malicious class. By comparing our models, their average and best performance as presented in Figure 6a and Table 4 it can be observed that the highest precision score was obtained for the best RF

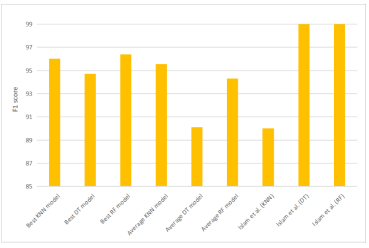
model. Moreover, this model managed to get the highest F1 score and ROC AUC score. The second most precise model was the best KNN model, having comparable results with the best RF model. On average, KNN performed best having the highest precision and F1 score across the average models. Unfortunately, the DT model performed the worst, on average achieving just five percent above a random baseline model.

When comparing our results with the performance denoted by Islam et al. [7] models, we managed to significantly improve the KNN model. In the experiments from [7], KNN gets 90% F1 score, while our best KNN model had 96.02% F1 score and on average 95.53% F1 score. For the case of DT and RF models, their approach has a better performance of 99% F1 score. Our DT model had a 94.7% F1 score at best and RF reached 96.39% at best. Still, we consider that our results were comparable with their experiments.

To our perspective, we manage to improve the methodology needed for running experiments in this benign/ malicious links classification field. Thus, the purpose of our paper was not to improve state-of-the-art detection algorithms for malicious links detection, but to try and perfect the strategy followed when finding solutions for this problem.



(A) All models comparison



(B) All models comparison with Islam et al. models [7]

FIGURE 6. Comparing models

If we compare our approach to other literature solutions that experiment with the same algorithms, our results are outperformed by theirs. This is the case in [15], where the authors deliver experiments on multiple ML algorithms, such as KNN (96.2% precision), DT (99% precision), RF (99.8% precision) etc. Reaching a higher score than ours may be because of using a 450,000 records dataset, with the URL and the label. Having a larger dataset can be helpful in properly fitting the models. Shantanu et al. [15] manually engineers lexical features from the URL, while we mostly used host-based features and two lexical ones, which are as well included in their model. Another disparity comes from our dealing with data imbalancement, while in [15] there are no mentions

about the actions taken to counteract it. Moreover, we computed the metrics as an average of multiple runs. Comparing our approach on the RF model with the Decision Forest proposed by Adas et al. [1] (99.8% accuracy), the dissimilarity comes from using another methodology and dataset. In [1], the dataset has over 2.4 million URLs and the ratio between malicious and benign samples is not mentioned. Besides that, the features used in classification are different and their training step includes a cross validation step, which may be very effective since their dataset is numerous.

Comparing the last step of the experiments, about feature importance, we managed to confirm the results within other articles. The experiments done in [11] concludes that no particular features dominated the detection algorithm. On the contrary, Johnson et al. [8] manage to prove the relevance of URL related attributes through chi-squared test, as our lexical URL features achieved a medium accuracy score. As well, in [9] the relevant attributes are URL-based, while the host-based ones (WHOIS information) are not as relevant. This is in contrast with our results that proved a higher score for host-based characteristics in general. The relevance computed with information gain in [6] denotes that the length of the URL is a top feature together with the count of dots, which is part of our special characters count attribute.

5. Conclusions and future work

In conclusion, malicious web links detection is a complex domain from an experimental point of view. There was almost no difference between the two normalization techniques we applied. Our main contribution is that we managed to propose an experimental methodology for malicious web links detection. Moreover, we got to improve the score metric of KNN model compared to other literature solutions and the RF model achieved the best precision (87.6%). Regarding feature importance analysis, we observed a high score for host-based features. Considering future work, we propose experimenting with more simple and complex ML algorithms on the same dataset such that we could draw a more relevant and complete overview from our experiments. Moreover, we plan to develop a real-time framework that will help users report malicious links. The collected samples of links can be further analyzed, other features can be extracted, especially the host-based ones.

References

[1] Adas, H., Shetty, S., and Tayib, W. Scalable detection of web malware on smartphones. In *2015 international conference on information and communication technology research (ICTRC)* (Abu Dhabi, UAE, 2015), IEEE, IEEE, pp. 198–201.

[2] APWG. Phishing activity trends report - q4, 2020. Tech. rep., APWG, USA, 2021.

[3] Catak, F. O., Sahinbas, K., and Dörtkardeş, V. Malicious url detection using machine learning. In *Artificial intelligence paradigms for smart cyber-physical systems*. IGI Global, Papua New Guinea, Turkey, 2021, pp. 160–180.
[4] Cofense. Annual state of phishing report. Tech. rep., Cofense, Leesburg, VA, USA, 2021.
[5] Cook, S. Phishing statistics and facts for 2019–2022, Oct 2022.
[6] Ibrahim, S., Herami, N. A., Naqbi, E. A., and Aldwairi, M. Detection and analysis of drive-by downloads and malicious websites. In *International Symposium on Security in Computing and Communication* (Trivandrum, India, 2019), Springer, Springer, pp. 72–86.
[7] Islam, M., Poudyal, S., Gupta, K. D., et al. Mapreduce implementation for malicious websites classification. *International Journal of Network Security & Its Applications (IJNSA) Vol 11* (2019).
[8] Johnson, C., Khadka, B., Basnet, R. B., and Doleck, T. Towards detecting and classifying malicious urls using deep learning. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl. 11*, 4 (2020), 31–48.
[9] Kumi, S., Lim, C., and Lee, S.-G. Malicious url detection based on associative classification. *Entropy 23*, 2 (2021), 182.
[10] Naveen, I. N. V. D., Manamohana, K., and Verma, R. Detection of malicious urls using machine learning techniques. *International Journal of Innovative Technology and Exploring Engineering 8*, 4S2 (2019), 389–393.
[11] Oshingbesan, A., Okobi, C., Ekoh, C., Richard, K., and Munezero, A. Detection of malicious websites using machine learning techniques. *preprint none*, none (06 2021), 1–5.
[12] Pakhare, P. S., Krishnan, S., and Charniya, N. N. Malicious url detection using machine learning and ensemble modeling. In *Computer Networks, Big Data and IoT*. Springer, Singapore, 2021, pp. 839–850.
[13] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.
[14] Sahoo, D., Liu, C., and Hoi, S. C. Malicious url detection using machine learning: A survey. *arXiv preprint arXiv:1701.07179* (2017).
[15] Shantanu, Janet, B., and Kumar, R. J. A. Malicious url detection: A comparative study. In *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)* (Tamil Nadu, India, 2021), IEEE, IEEE, pp. 1147–1151.
[16] Tung, S. P., Wong, K. Y., Kuzminykh, I., Bakhshi, T., and Ghita, B. Using a machine learning model for malicious url type detection. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems* (Cham, 2022), Y. Koucheryavy, S. Balandin, and S. Andreev, Eds., Springer International Publishing, pp. 493–505.
[17] Urcuqui, C., Navarro, A., Osorio, J., and García, M. Machine learning classifiers to detect malicious websites. *SSN 1950* (2017), 14–17.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeș-Bolyai University, Cluj-Napoca, Romania
Email address: claudia.coste@ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 1, 2023
DOI: 10.24193/subbi.2023.1.03

DETECTING PROGRAMMING FLAWS IN STUDENT SUBMISSIONS WITH STATIC SOURCE CODE ANALYSIS

PÉTER KASZAB AND MÁTÉ CSERÉP

Abstract. Static code analyzer tools can detect several programming mistakes, that would lead to run-time errors. Such tools can also detect violations of the conventions and guidelines of the given programming language. Thus, the feedback provided by these tools can be valuable for both students and instructors in computer science education. In our paper, we evaluated over 5000 student submissions from the last two years written in C++ and C# programming languages at Eötvös Loránd University Faculty of Informatics (Budapest, Hungary), by executing various static code analyzers on them. From the findings of the analyzers, we highlight some of the most typical and serious issues. Based on these results, we argue to include static analysis of programming submissions in automated and assisted semi-automatic evaluating and grading systems at universities, as these could increase the quality of programming assignments and raise the attention of students on various otherwise missed bugs and other programming errors.

1. Introduction

The demand for IT professionals is constantly increasing, as software development and maintenance is required in various fields, ranging from the finance sector through energy and manufacturing to healthcare [9]. As a direct consequence, more and more people are enrolling each year in computer science degree programs and other IT and programming related courses at universities [14]. This increment of students significantly increases the workload of university teachers and makes the manual grading of each student submission

Received by the editors: 01 March 2023.

2010 *Mathematics Subject Classification.* 68U99, 68Q55, 97Q70.

1998 *CR Categories and Descriptors.* F.3.2 [**Theory of Computation**]: Logics and Meanings of Programs – *Semantics of Programming Languages*; D.3.4 [**Software**]: Programming Languages – *Processors*; K.3.2 [**Computing Milieux**]: Computers and Education – *Computer and Information Science Education*.

Key words and phrases. static code analysis, C++, C#, student submission, computer science education, programming flaw.

unsustainable. As a result, the usage of automatic grading systems for programming assignments have gained focus in the past years. Whether they are developed commercially, open-source or in many cases as an internal project at a university, these systems are becoming indispensable for instructors [11].

Non-trivial run-time errors in programming submissions are often missed by instructors and automatic testers, because these kinds of errors are not always easy to find and reproduce. Furthermore, there are solutions with functionally correct and bug-free code which do not follow the conventions and guidelines of the given programming language. These kinds of errors can be avoided and the application of the given guidelines can be forced using static code analyzers [4, 12, 18, 22].

Static analyzer tools can be utilized to check or flaws in other programs. This can be achieved by evaluating the source code, the byte code or the binaries [2]. These tools can help developers to identify a wide range of issues from incorrect styling and formatting to serious security issues [10]. In Section 2 we review the previous applications of such tools in higher education.

For our research, we evaluated over 5000 student submissions from the last two years written in C++ and C# programming languages at the Eötvös Loránd University Faculty of Informatics (*ELTE FI*), by executing various static code analyzers on them. In Section 3 we introduce the evaluated courses, the used analyzer tools and the criteria for filtering analyzer results. Then, in Section 4 and 5 the most typical and interesting findings are presented for the C++ and C# submissions with simple code examples. We showcase our prototype implementation for an automated evaluator system in Section 6. Finally, the conclusion and future work is described in Section 7.

2. Related work

2.1. **Automatic evaluation of submissions.** Static analyzers can be used in education in order to help the learning process of the students and speed up the evaluation of the submission.

Michael Striewe and Michael Goedicke [22] reviewed the static analysis approaches that can help in providing feedback to the submitted solutions. They highlighted the following requirements for a system that evaluates submissions with static analysis:

- check for mistakes and violated coding conventions in syntactically correct source code;
- check for source code which is correct, but contains element that are not allowed in the context of the given course or exercise;
- check for missing code structures;
- give hints on how to solve the previously mentioned issues.

J. Walker Orr [19] proposes a rule-based tool for Java and Python that provides feedback on predefined rules. The checked design principles are formalized as logical functions, and they are applied to the subtrees of the abstract syntax trees. The implemented rules are designed to meet the needs of students. The system was hosted as a standalone web service where students could submit their solutions. There were no limitations to the number of uploads, and the execution of the tests was instant. Thus, this increased the transparency of the grading progress. On average, the rate of design quality flaws dropped 47.84% on different assignments.

Blau et al. developed a tool called *FrenchPress* [4] which is an Eclipse-plugin designed for students with intermediate knowledge of the Java programming language. It focuses on silent flaws that often get overlooked by students, because IDEs and compilers do not catch them. The advantage of the IDE integration, that the students can get feedback while they work in their code without leaving the development environment. The authors emphasize that the feedbacks should be relevant for the situation of the students and should be easy to understand. Also, it is important to minimize the number of false positives, as they could be more problematic than false negatives for inexperienced users. In the end, the percentage of cases when *FrencPress* motivated the users to improve their programs varied from 36% to 64% depending on the course.

In contrast to the previously mentioned tools, *Hyperstyle* [3] uses existing professional code analyzers to evaluate the submissions. It currently supports Java, Kotlin, JavaScript and Python, but it can be extended easily with analyzers for other languages. The possible errors are split into the following categories: code style, code complexity, error-proneness, best practices, and minor issues. Based on the findings, it gives grades for the solutions on a four level scale: excellent, good, moderate, bad. Additionally, it provides custom messages for some issues, because students often need more detailed errors messages than the output of the professional tool. *Hyperstyle* was tested on the *Stepik* and *Jetbrains Academy* platforms, but it can be added to other MOOC systems as well. For Java solutions, the number of students who made fewer mistakes increased, and the number of who made six or more mistakes decreased. For Python solutions, the number of students without code quality issues increased four times and the number of students who made two or more errors decreased.

2.2. **Analyze solutions from previous semesters.** While the previously reviewed papers present solutions that provide feedback to students or grade their submissions automatically, analyzing datasets of existing submission can provide valuable information on several aspects. Moreover, checking older

solutions can also help to evaluate the code analyzer tools, and adapt their results to the needs of the students and teachers.

Molnar et al. [18] evaluated Python assignments from an introductory programming course using *Pylint*. They also developed a custom tooling that is able to visualize and list findings for: a specific student, a given assignment, or an assignment corresponding to multiple students. Their study showed that *Pylint* provided meaningful information regarding code style and logical errors.

Keuning et al. [12] investigated code quality issues in Java programs. The analyzed source files were collected from the *Blackbox* database, which contains Java solutions written in the *BlueJ IDE*. The database stores multiple versions of the source files, and also collects events and usage statistics from the *BlueJ IDE* (e.g., enabled plugins). They used the *PMD* tool for analysis and categorized the errors into the following categories: flow, idiom, expression, decomposition, and modularization. Some detectable issues by *PMD* needed to be discarded, because they were too advanced for novice programmers, or they were too specific for a library or platform. Errors from categories like presentation and documentation were completely discarded. They also examined the most commonly fixed errors. Empty if statements and singular fields were the most commonly fixed issues. Probably, because the initial uploads were not finished. Issues like too many fields or methods were fixed in less than 5% of the cases. So, the overall fixing rate was relatively low.

Similarly to the previous paper, Edwards et al. [6] also analyzed Java programs from four different courses for students with different skill-levels. The dataset contained nearly 10 million errors produced by 3691 students. They used the *CheckStyle* and *PMD* open-source tools for static code analysis, but they created their own categories for errors: braces, coding flaws, documentation, excessive coding, formatting, naming, readability, style, and testing. The most common categories were documentation, formatting, style, and coding flaws. The coding flaws category could indicate that the student is struggling. Usually, the solutions with lower scores had more coding flaws in them. Also, it is possible that some students ignore the warnings produced by the analyzers, if they are dominated by documentation and formatting issues. This factor should be considered when analyzers are used in automatic evaluator systems.

3. Methodology

The first batch containing 3433 solutions written in C++ were collected from the *Object-oriented programming* course. While the students have to develop command-line interface applications, they have to manage memory

manually and use advanced object-oriented techniques, like polymorphism. In addition, 226 C++ projects were added from the *GUI programming with QT* course, where the participants have to develop complex graphical application with layered architecture (Model-View).

The C++ submissions were analyzed with *Clang-Tidy*, *Clang Static Analyzer* and *Cppcheck* [1, 13, 15, 16]. To run the previously mentioned tools and visualize their results, we have used *Ericsson CodeChecker* [8].

In the case of C# projects, 2148 programming submissions were collected from the *Event-driven programming* course, where students have to develop Windows Forms, WPF and Xamarin/MAUI graphical applications. Similarly to the *GUI programming with QT* course, the usage of layered architecture (Model-View and Model-View-ViewModel) is mandatory. For these programs, we have used both first-party (*Microsoft NetAnalyzers*) and third-party (*Roslynator Analyzers* and *SonarAnalyzer CSharp*) analyzers built on top of APIs provided by the *Microsoft Roslyn* compiler platform [23, 24].

Table 1 summarizes the previously described tools, courses, and analyzed submissions.

Language	**Analyzer tools**	**Course**	**Submissions**
C++	Clang Tidy, Clang Static Analyzer, Cppcheck	Object-oriented programming	3433
		GUI programming with QT	226
C#	Microsoft NetAnalyzers, Roslynator Analyzers, SonarAnalyzer CSharp	Event-driven programming	2148

TABLE 1. Summary of the used analyzers and evaluated submissions

From the findings of the analyzers, we have selected the presented errors according to the following criteria:

- We have included the most common and typical errors.
- Some errors only occurred in a handful of submission, but they indicated serious design flaws or lack of understanding.
- We excluded styling errors. While code-styling is important, there were no enforced styling guidelines for the assignments. Also, these rules often require detailed configuration in real-world projects.
- For the C# programs a significant part of findings reported by the Roslyn-based analyzers were possible refactorings, those were also discarded.

4. C++ RESULTS

In this section, we present the selected errors from the C++ solutions. Figure 1 shows the number of solutions from both courses where the those errors occurred.

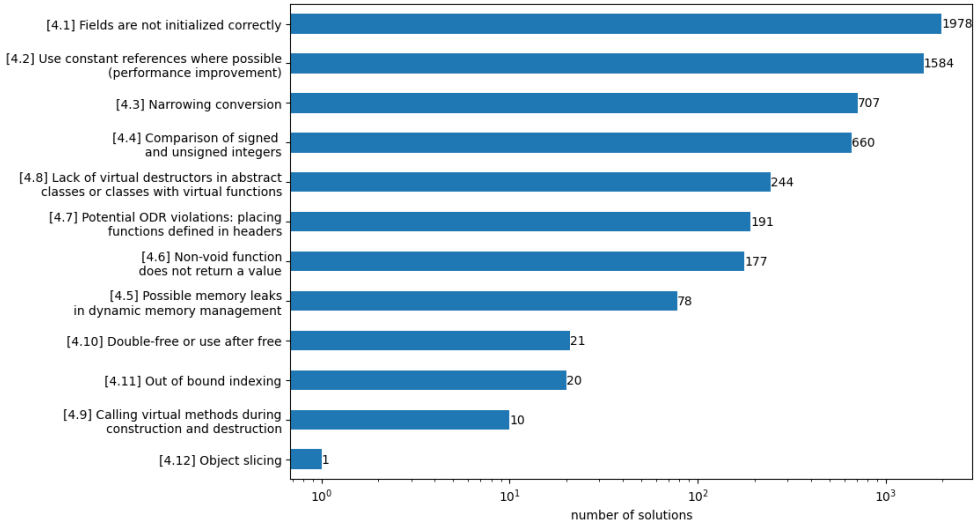


FIGURE 1. The number of C++ solutions with errors

4.1. **Fields are not initialized correctly.** Fields are usually not initialized automatically in C++. However, they could be initialized during debug compilation on some platforms, misleading students (as in Listing 1). It is generally a good practice to give sensible starting values to fields during construction.

```
class Example {
private:
    int _x, _y, _z;
public:
    Example(int x, int y): _x(x), _y(y) {}
    void method() {
        if (_z > 0) { /* ... */ }
        // _z not guaranteed to be initialized to zero
    }
};
```

LISTING 1. Field initialization

4.2. **Use constant references where possible (performance improvement).** While function `f` in Listing 2 is functionally correct, it has two potential performance problems:

(1) the `vec` vector is copied every time `f` is called;
(2) the `curr` vector is copied in every iteration.

Using `const` references for the parameter and the loop variable improves performance of this program.

```
void f(vector<vector<int>> vec) {
    for(vector<int> curr : vec) { /* ... */ }
}
void f_improved(const vector<vector<int>>& vec) {
    for(const vector<int>& curr : vec) { /* ... */ }
}
```

LISTING 2. Perfomance can be improved with constant references

However, sometimes copying the values of the parameters is the desired behavior. Code analyzers are smart enough to give hints based on the context. So, these warning are only showed if marking the parameter to a `const` reference would not break the given program.

4.3. **Narrowing conversion.** Conversion from a wider data type to a narrower can lead to data loss (e.g., `float` → `int`) and/or integer overflow (e.g., `unsigned int` → `int` in Listing 3).

```
void search(int elem, bool& found, int &ind) {
  found = false;
  for (unsigned int i = 0; i < vec.size() && !found; ++i) {
    if(vec[i] == elem) {
      found = true;
      ind = i; // Could cause integer overflow
    }
  }
}
```

LISTING 3. Possible integer overflow, because of narrowing conversion

4.4. **Comparison of signed and unsigned integers.** Direct comparison between signed and unsigned integers is not safe in C++. In most cases this error occurred, when students compared a loop-variable with a size of a container that has `std::size_t` type which is an unsigned integer type (Listing 4). While this have not caused problems in the submitted solution, it is still considered a bad practice, because `vec.size()` can be greater than the maximum value of `int` on the given platform.

```
for (int i = 0; i < vec.size() /* std::size_t */ ; ++i) {}
```

LISTING 4. The maximum size of `int` might be smaller than `vec.size()`

Comparison of signed and unsigned integers could also be problematic if the signed integer value is negative. In Listing 5, we would expect that it will print `0` as i is not greater than j, but the value of i is also cast to `unsigned int` and it underflows.

```
int i = -4;
unsigned int j = 5;
std::cout << (i > j) << std::endl; // Expected 0, but prints 1
```

LISTING 5. `int i` is casted to `unsigned int`

4.5. **Possible memory leaks in dynamic memory management.** Freeing allocated dynamic memory is often missed by students. Consider the `Stack` class in Listing 6, where the writer of the code allocates memory for the array, but the destructor is missing. Thus, the memory will not be freed after s is not used anymore.

```
class Stack {
private:
  int _top, _size;
  int* _arr;
public:
    Stack(int size)
    : _top(-1), _size(size), _arr(new int[size]) {}
    // ...
};
```

LISTING 6. Memory leak: missing destructor

4.6. **Non-void functions does not return a value.** Reaching the end of the body of a non-void function without returning a value is will not generate a compiler error by default, but it is an undefined behavior in C++. A good example of this, a stack class where the `pop` method of a stack that removes the item from the top of the stack and returns its value. The implementation in Listing 7 of the `pop` method is error-prone, because the user of the class can call the method on an empty stack.

```
int Stack::pop() {
    if (!isEmpty()) { return _vec[--_top]; }
}
```

LISTING 7. Empty stacks are not handled

4.7. **Potential ODR violations: placing function in headers.** One Definition Rule (ODR) means that non-inline functions and types must have only one definition in the entire program [21]. For instance, placing functions in headers can lead to ODR violations. This does not necessarily mean that the solution does not compile or run until it is only included in one source file. However, if the student had included it in two or more sources, the compiler would not have accepted the solution.

Consider the scenario illustrated in Listing 8, while

- the `g++ main.cpp first.cpp` command will compile the program successfully;
- the `g++ main.cpp first.cpp second.cpp` command will fail.

```
/// Contents of helpers.h:
int square(int x) { return x * x; }

/// Contents of first.cpp:
#include "helpers.h"
void first_calculation() { int res = square(2); /* ... */ }

/// Contents second.cpp:
#include "helpers.h"
void second_calculation() { int res = square(3); /* ... */ }

/// Contents main.cpp:
// helpers.h is not included in main.cpp
```

LISTING 8. Functions in headers

4.8. **Lack of virtual destructors in abstract classes or classes with virtual functions.** It is possible that the student implemented all necessary destructors, but they are not marked as virtual when needed. In Listing 9, if the destructor of `Base` is not marked as virtual and `delete` is called on a variable with static type of `Base`, then the destructor of `Derived` will not be called.

```
struct Base {
    virtual void method() = 0;
    ~Base() {std::cout << "base "; }  // Should be virtual
};
struct Derived: public Base {
    void method() override {  }
    ~Derived() { std::cout << "derived "; }
};
void f() {
    Base* d = new Derived;
    delete d; // outputs: base
}
```

LISTING 9. Destructors should be `virtual`

4.9. **Calling virtual methods during construction and destruction.** During construction and destruction, the virtual call mechanism is disabled. Therefore, the implementation from the current class is used, as illustrated in Listing 10 with the call of `f`. Calling virtual methods in the constructor is not necessarily a problem, but the student might not aware of the previously described behavior.

```
struct Base {
  Base() { f(); } // Prints base
  virtual void f() { std::cout << "base"; }
};
struct Derived: public Base {
  void f() override { std::cout << "derived"; }
};
```

LISTING 10. Virtual calls in constructors

4.10. **Double-free and use after free.** In C++ `delete` should be called only once for the same reference and the reference should not be used after `delete` called on it. Listing 11 counts not only as double-free, but an infinite loop, because `~Example` will always get called again, recursively. This is a good example of how reported errors can also indicate lack of understanding from students.

```
struct Example {
    ~Example() { delete this; }
};
```

LISTING 11. Incorrect usage of `delete`

4.11. **Out of bound indexing.** Out of bound indexing is often missed by beginner programmers. It usually results in a *memory segmentation fault*. The provided example (Listing 12) is relatively simple: the student made a small mistake and wrote `<=` instead of `<`. Fortunately, the used code analyzers can spot possible out of bound indexing in more complex scenarios.

```
int arr[10];
for (int i = 0; i <= 10; ++i) { arr[i] = i; }
```

Listing 12. Out of bound indexing

4.12. **Object slicing.** Slicing happens when copying a derived object into a base object: the members of the derived object (both member variables and virtual member functions) will be discarded [5]. In Listing 13, slicing object from type `Derived` to `Base` discards override `method`.

```
struct Base {
    virtual void method() { std::cout << "base"; }
};
struct Derived: public Base {
    virtual void method() { std::cout << "derived"; }
};
void f(Base obj) { obj.method(); }
int main() {
    Derived d;
    f(d); // prints base
    return 0;
}
```

Listing 13. Object slicing

5. C# results

In this section, we present the selected errors from the C# solution. Figure 2 shows the number of solutions from both courses where the those errors occurred, categorized by tasks. It is worth to note that the number of solutions containing the highlighted errors are really similar for the WinForms, WPF, and Xamarin/MAUI assignments. This is because the students have to develop the same software for all three tasks, and they are encouraged to reuse layers from their previous solutions. Exams are different, because student have to develop new applications from scratch, but reusing their existing materials is still allowed.

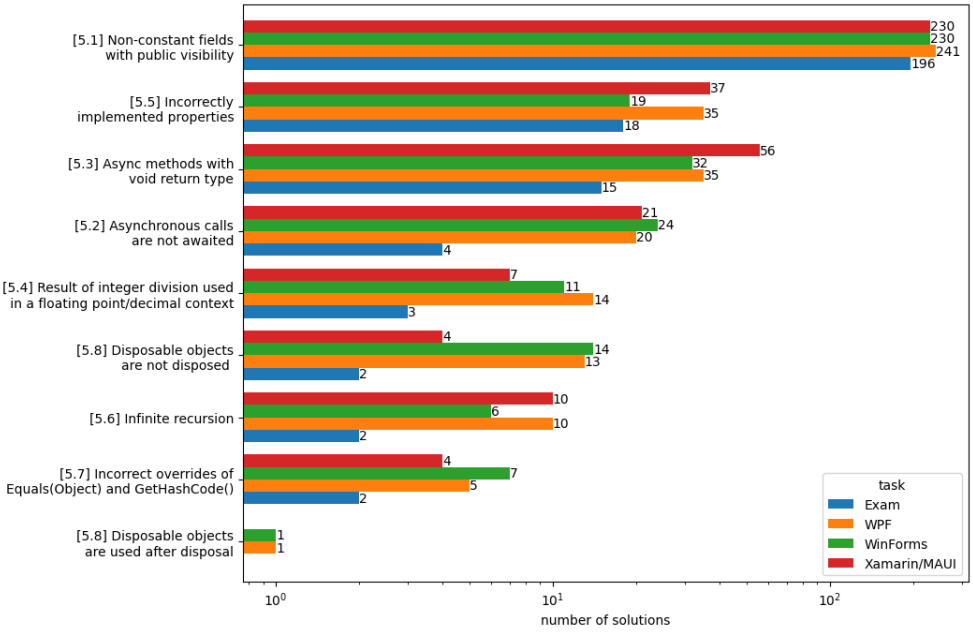


FIGURE 2. The number of C# solutions with errors

5.1. **Non-constant fields with public visibility.** Using public mutable fields are generally considered a bad practice and against guidelines in C#. There are several alternatives:

- mark the field `readonly` or `const`;
- use auto-implemented properties instead;
- make it private and access it with a property or method.

5.2. **Asynchronous calls are not awaited.** In Listing 14, `NewGameAsync` is an async function, but it is not awaited. Thus, the state of the `model` object might be incorrect when `AdvanceGame` is called.

```
GameModel model = new GameModel();
model.NewGameAsync();
model.AdvanceGame();
```

LISTING 14. `model.NewGameAsync()` is not awaited

5.3. **Asynchronous methods with void return type.** Asynchronous function should return `Task` or `Task<T>`, because they cannot be awaited and exceptions cannot be caught from them (Listing 15). Event handlers are the only

exceptions according to the Microsoft Learn guidelines, because they usually have to return `void` [17].

```
public async void LoadAsync(string filePath) {
    FileContent = await File.ReadAllTextAsync(filePath);
}
```

LISTING 15. `LoadAsync` cannot be awaited

5.4. **Results of integer division should not be assigned to floating point/decimal variables/parameters.** In Listing 16, if the result of `size / 2` is positive, then it is already floored because of the integer division. Therefore, calling `Ceiling` will not return the expected result.

```
int size = // ...
if ((int)decimal.Ceiling(size / 2) == x) { /* ... */ }
```

LISTING 16. Integer disivision

5.5. **Incorrectly implemented properties.** The getters and setters of the properties should access the correct backing fields. As shown in Listing 17, the student may want to write a read-only property, but the setter is still present with an empty body. The correct solution would be a property without a setter, because assigning a value to a property will not generate a compile-time error and the user may think that the property is writable. In contrast, if the setter is not present, then both the compiler and the IDE will show an error upon assignment.

```
private int property;
public int Property { get { return property; } set {} }
```

LISTING 17. Read-only property: `set` should be omitted

5.6. **Infinite recursion.** A trivial example of this error, when the setter tries to assign the vale to the property itself (Listing 18). This may be the result of a typo in the source code, as backing fields often have the same name as the property, but with a different case. Calling the setter of such a property would lead to an infinite recursion.

```
private int property;
public int Property {
    get { return property; }
    set { Property = value; }
}
```

LISTING 18. Incorrectly implemented setter: infinite recursion

5.7. **Incorrect overrides of Equals(object) and GetHashCode().** When overriding `Equals(object)` and `GetHashCode()` certain rules should be followed, such as:

- `Equals(object)` and `GetHashCode()` should be overridden in pairs.
- Classes directly extending `object` should not call `base` in `GetHashCode` or `Equals`. The implementation in `object` are based on object reference.

In Listing 19, the student overrides both methods, but the `GetHashCode` calls the implementation from the `object` class.

```
class ClassName {
    public override int Equals() {
        // correct implementation
    }
    public override int GetHashCode() {
        base.GetHashCode(); // Calls GetHashCode from object
    }
}
```

Listing 19. Incorrect override of `GetHashCode()`

5.8. **Mistakes related to disposable objects.** While C# has automatic garbage collector, the unmanaged resources taken by certain classes should be freed. For instance (Listing 20), if a file opened by the `StreamReader` class, then it should be closed after usage. The `Dispose` should be called (or `Close` in this case), preferably in a `finally` block. A `using` statement or declaration would be an even better option, as it ensures the correct usage of disposable objects.

```
List<string> values = new List<string>();
StreamWriter sw = new StreamWriter("output.txt");
foreach (string line in values) { sw.WriteLine(line); }
```

Listing 20. The file is not closed after usage

It is also important that the objects cannot be used after they are disposed. In the example in Listing 21, the `Dispose` method is automatically called after the execution leaves the block of the using statement. So, the returned `StreamReader` instance will not be able to read the file, as its methods will throw an `ObjectDisposedException`.

```
public StreamReader CreateReader(string filename) {
    using (StreamReader sr = new StreamReader(filename)) {
        return sr;
    }
}
```

LISTING 21. `sr` is returned after disposal

6. INTEGRATION WITH AUTOMATED EVALUATOR SYSTEMS

We developed a prototype-implementation as part of the open-source *TMS* task management system developed at ELTE FI [7], which already contains a custom developed Docker-based automated evaluator and integrates the static analysis tool *CodeCompass* [20], but for code comprehension purposes.

We extended the evaluator system with the tools mentioned in Section 3. *CodeChecker* is an important part of our solution, because apart from running static analysis on C/C++ solutions with *Clang SA*, *Clang Tidy*, and *Cppcheck* it can also process the output of more than 20 third-party analyzers and convert it to its own format. Thus, it is enough if TMS can process only one report format. Moreover, we could take advantage of the additional tools provided by *CodeChecker*, such as the HTML report viewer.

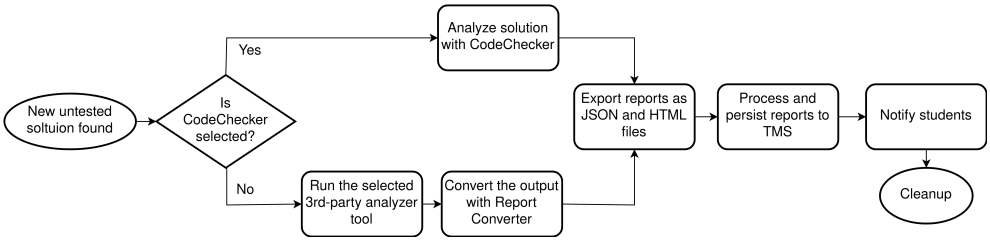


FIGURE 3. The workflow of automated static code analysis in TMS

The Docker environment and the selected tools can be individually configured for each assignment. So, instructors can adapt the tools to the need of their courses. The system regularly checks for new submissions from tasks with valid evaluator configuration. When there is a new untested solution, the system runs the selected tools in Docker containers and converts the reports to a common report format if necessary. Finally, when the reports are available in the required format, TMS persists them for the given solution, and notifies the student about the completion of the analysis. The described workflow is illustrated in Figure 3.

Both the students and the instructors can view the results on the page of the given submission. It is important that the results should be presented in a legible format. They are prioritized by severity to help students identify the more serious issues first. The HTML reports produced by CodeChecker are also available from the user interface, so the reports can be viewed without downloading the submissions.

7. Conclusions and Future Work

In our paper, we have evaluated over 5000 student submission written in C++ and C#, by running static code analyzer tools on them. We have found violations of conventions and various programming bugs which could have been filtered with static analysis, but were overlooked by the teachers, probably due to the high number of student submissions they had to evaluate and grade. In these cases, the feedback provided by the analyzers could help students to fix their mistakes before the deadlines and learn from them. Furthermore, the usage of these tools would allow a more thorough assessment by teachers and speed up the grading process.

While analyzing solutions from previous semesters helped us to create the initial prototype implementation, choose the right tools, and configure them, we believe our solution can be improved further by testing it during an academic semester. First, it should be observed what is the impact of such a tool on the students' behavior, how much the quality of their submissions improved. Furthermore, it should also be determined if the students really understand and use correctly the provided feedback. The current implementation shows the reports to both students and instructors, but some tips might be applied too easily without understating them. It might be beneficial to make the detail of the feedback configurable or add an option to hide it from the students completely, so instructors can choose according to their preferences. Another possible approach could be adding an option to limit the number of possible uploads, thus students have to rethink twice before re-upload their solutions just to check if they managed to solve the reported problems. Finally, after the testing is completed and the previous questions are issued, we aim to introduce our solution for other courses at ELTE FI.

References

[1] Babati, B., Horváth, G., Májer, V., and Pataki, N. Static analysis toolset with Clang. In *Proceedings of the 10th International Conference on Applied Informatics* (2017), pp. 23–29.

[2] Bardas, A. G., et al. Static code analysis. *Journal of Information Systems & Operations Management 4*, 2 (2010), 99–107.

[3] Birillo, A., Vlasov, I., Burylov, A., Selishchev, V., Goncharov, A., Tikhomirova, E., Vyahhi, N., and Bryksin, T. Hyperstyle: A tool for assessing the code quality of solutions to programming assignments. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2022), SIGCSE 2022, Association for Computing Machinery, pp. 307—313.

[4] Blau, H., and Moss, J. E. B. Frenchpress gives students automated feedback on java program flaws. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2015), ITiCSE '15, Association for Computing Machinery, pp. 15—-20.

[5] Clang Team. LLVM - Clang-tidy - cppcoreguidelines-slicing. https://releases.llvm.org/13.0.0/tools/clang/tools/extra/docs/clang-tidy/checks/cppcoreguidelines-slicing.html, Accessed: 2023-02-25.

[6] Edwards, S. H., Kandru, N., and Rajagopal, M. B. Investigating static analysis errors in student java programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (New York, NY, USA, 2017), ICER '17, Association for Computing Machinery, pp. 65—-73.

[7] ELTE. TMS – Task Management System. https://tms-elte.gitlab.io/, Accessed: 2023-02-27.

[8] Ericsson Ltd. CodeChecker. https://codechecker.readthedocs.io/, Accessed: 2023-02-25.

[9] Eurostat. ICT education - a statistical overview. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=ICT_education_-_a_statistical_overview, Accessed: 2023-04-16.

[10] Gomes, I., Morgado, P., Gomes, T., and Moreira, R. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal* (2009).

[11] Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (New York, NY, USA, 2010), Koli Calling '10, Association for Computing Machinery.

[12] Keuning, H., Heeren, B., and Jeuring, J. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (New York, NY, USA, 2017), ITiCSE '17, Association for Computing Machinery, pp. 110—-115.

[13] Kremenek, T. Finding software bugs with the clang static analyzer. *Apple Inc* (2008).

[14] Loyalka, P., Liu, O. L., Li, G., Chirikov, I., Kardanova, E., Gu, L., Ling, G., Yu, N., Guo, F., Ma, L., Hu, S., Johnson, A. S., Bhuradia, A., Khanna, S., Froumin, I., Shi, J., Choudhury, P. K., Beteille, T., Marmolejo, F., and Tognatta, N. Computer science skills across china, india, russia, and the united states. *Proceedings of the National Academy of Sciences 116*, 14 (2019), 6732–6736.

[15] Marjamäki, D. Cppcheck. https://cppcheck.sourceforge.io/, Accessed: 2023-02-23.

[16] Martignano, M., and Spazio, I. A new static analyzer: The compiler. *ADA USER 40*, 2 (2019), 99–103.

[17] Microsoft. Async return types (C#). https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-return-types, Accessed: 2023-02-23.

[18] Molnar, A.-J., Motogna, S., and Vlad, C. Using static analysis tools to assist student project evaluation. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on Education through Advanced Software Engineering and Artificial Intelligence* (New York, NY, USA, 2020), EASEAI 2020, Association for Computing Machinery, pp. 7—12.

[19] Orr, J. W. Automatic assessment of the design quality of student python and java programs. *arXiv e-prints* (2022).

[20] Porkoláb, Z., Brunner, T., Krupp, D., and Csordás, M. Codecompass: an open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension* (2018), pp. 361–369.

[21] Quinlan, D., Vuduc, R., Panas, T., Haerdtlein, J., and Saebjoernsen, A. Support for whole-program analysis and the verification of the one-definition rule in C++. In *Static Analysis Summit 2006* (6 2006).

[22] Striewe, M., and Goedicke, M. A review of static analysis approaches for programming exercises. In *Computer Assisted Assessment. Research into E-Assessment* (07 2014), vol. 439, Springer, pp. 100–113.

[23] Sundström, J. Assessment of Roslyn analyzers for Visual Studio, 2019.

[24] Vasani, M. *Roslyn Cookbook*. Packt Publishing Ltd., 2017.

ELTE Eötvös Loránd University, Faculty of Informatics, Department of Software Technology and Methodology, H-1117 Budapest, Pázmány P. sny 1/C.

Email address: t5mop2@inf.elte.hu

Email address: mcserep@inf.elte.hu

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 1, 2023
DOI: 10.24193/subbi.2023.1.04

DOMAS: DATA ORIENTED MEDICAL VISUAL QUESTION ANSWERING USING SWIN TRANSFORMER

TEODORA-ALEXANDRA TOADER

Abstract. The Medical Visual Question Answering problem is a joined Computer Vision and Natural Language Processing task that aims to obtain answers in natural language to a question, posed in natural language as well, regarding an image. Both the image and question are of a medical nature. In this paper we introduce DOMAS, a deep learning model that solves this task on the Med-VQA 2019 dataset. The method is based on dividing the task into smaller classification problems by using a BERT-based question classification and a unique approach that makes use of dataset information for selecting the suited model. For the image classification problems, transfer learning using a pre-trained Swin Transform based architecture is used. DOMAS uses a question classifier and seven image classifiers along with the image classifier selection strategy and achieves 0.616 strict accuracy and 0.654 BLUE score. The results are competitive with other state-of-the-art models, proving that our approach is effective in solving the presented task.

1. Introduction

Visual Question Answering (VQA) is a task that combines both the Natural Language Processing (NLP) Field and Computer Vision (CV). The inputs of a VQA model are an image and a question addressed in natural language, question that can be answered from the given image. The output is of course the answer returned in natural language as well. Medical Visual Question Answering (MVQA) is a task that evolved from the VQA task by constraining the domain of the image and question to be the medical domain. Therefore, the images can take the form of pictures obtained using medical imaging, such as X-rays, MRIs, CT scans, as will be in our case while the questions can enquire about different aspects associated with the image. Using intelligent

Received by the editors: 14 June 2023.

2010 *Mathematics Subject Classification.* 68T45, 68T50.

1998 *CR Categories and Descriptors.* I.2.7 [**Artificial Intelligence**]: Natural Language Processing – *Language parsing and understanding*; I.2.7 [**Artificial Intelligence**]: Applications and Expert Systems – *Medicine and science*.

Key words and phrases. Medical Visual Question Answering, Swin Transformer.

algorithms to solve the MVQA tasks could benefit the medical field immensely as such a model could provide a second opinion to medical professionals and could also make medical investigations more accessible.

One big challenge of MVQA is the limited amount of data that is available compared to the general VQA task that has been more widely explored. The lack of data for such an extensive task can lead models to overfit and not provide enough generalization. One recent approach that has been successfully used in the domain as a solution to the problems caused by small amounts of data is transfer learning, which focuses on using information gained from solving one task on a second related task. An architecture that proved to be very successful in association with transfer learning is the transformer architecture introduced by Vaswani et al. in [17]. Transformers are deep learning models based on the attention mechanism that proved to be very efficient in both NLP and CV, especially when pretrained on large amounts of data and then fine-tuned for specific tasks.

In this paper we introduce DOMAS, a deep learning model that solves the MVQA task on the Med-VQA 2019 dataset. The architecture is based on transforming the complex MVQA task into smaller image classification problems by selecting the image model using a model based on BERT architecture [6] applied on the questions and our dataset knowledge and then solving the image classifications using models based on an impressive computer vision backbone, introduced by Liu et al. called the Swin Transformer. [10]. Our approach achieves a 0.616 accuracy and 0.654 Bleu score on the VQA-Med-2019 test set which makes it comparable with other state-of-the-art models. The purpose of this paper is to answer the following research questions:

- Can the Swin architecture be an alternative to the more commonly used CNN based networks on this task?
- Does using information about the dataset improve the classification for modality models, especially for questions with affirmative or negative answers?

The structure of the paper will be as follows. Section 2 will present other approaches from the literature. Section 3 will provide a detailed description of the dataset. Our approach will be described in Section 4, while the experiments will be detailed in Section 5. The last section will provide the conclusions of the study as well as possible ideas for future work.

2. Related Work

The recent advancements in computer vision and natural language processing also led to advancement in the joined image and language task that is VQA. Most state-of-the-art models such as the ones of Chen at al. [4],

Wang et al. [19] and Bao et al.[3] make use of the transformer architecture for vision-language pre-training and for other tasks such as [4] which uses the Vision Transformer for feature extraction as well. The general VQA task has the advantage of large datasets, performant models being pre-trained on millions of images and text samples which is not currently possible for MVQA. However, other methods and architectures can also be used for MVQA.

Many approaches have been proposed for the ImageCLEF 2019 Med-VQA dataset, some of them during the competition that proposed the dataset. The two highest-ranking teams at the ImageCLEF 2019 competition for the VQA-Med task combined features extracted from image and text using a fusion algorithm. Yan et al. [20] used a VGG-16 [14] inspired network combined with Global Average Pooling for image feature extraction and the basic BERT [6] model as the question encoder. The fusion of the two types of features extracted was achieved by using multi-modal factorized bilinear pooling with co-attention [21]. Minh Vu et al. [18] also use a CNN based network, namely ResNET-152 [7], to extract image features and BERT for question features. The features are fused using an attention mechanism and global image features are obtained, while the question features are also linearly transformed to obtain global question features. The global features are then further combined using a bilinear transformation. Some contestants also made use of the nature of the dataset and divided the problem into four different problems, one for each type of question. Zhou et al. [22] propose a different type of model for the plane, organ and modality questions, where a classifier is used to get the answer, and for abnormality questions where a generative method is used. The simple classifier consists of an Inception ResNet-V2 [16] for image feature extraction and BERT for question embeddings. The features are combined through a Multi-Layer Perceptron (MLP). The generator used for the abnormality questions consists of a sequence-to-sequence model. The encoder part is similar to the classifier while the decoder consists of a long short-term memory network (LSTM), and it continuously generates the probability distribution of the next word. Another interesting submission is the one of the JUST team [2] which creates an individual model for each type of question as well. They consider the questions to be repetitive and therefore each model is in fact an image classification model or a combination of image classification models. We can observe that all proposed models used pre-trained networks for feature extraction as the models benefit tremendously from transfer learning given the dimension of the dataset. Particularly, the proposed models also use CNN based networks for image feature extraction.

More recent approaches on the ImageCLEF 2019 VQA-Med dataset also make use of the transformer architecture and obtain improved results. Ren at

al. [12] propose CGMVQA, a model that can switch between a classification and a generative mode, by changing only the loss function and the output layer, in order to better fit the approached problem. They divide the task into five subtasks depending on the type of question: yes/no questions, organ, plane, modality and abnormality. To obtain the image features they extract from different convolutional layers of a ResNet-152. The questions are tokenized, and token, segment and positional embeddings are used to obtain the final features. These two types of features are used in the classification mode, for the generative mode, masked answers are also added as the method used for the generation is masking position by position. To get the final outputs, the features are fed into a slightly changed Transformer network. Another method that makes greater use of transformer capabilities is proposed by Khare et al. [8] where the authors propose MMBERT (Multimodal Medical BERT), a BERT like architecture that is pre-trained using self-supervised learning. The model is pretrained on medical images and their corresponding captions using MLM. The image features are extracted as in [12] and the captions are modified by replacing medical terms with the [MSK] token and then the embeddings are obtained using BERT. The obtained embeddings are passed through a BERT-like encoder and then a classifier is used to predict the initially masked word.

3. Dataset Description

The dataset we are going to use in our experiments is the VQA-Med-2019 dataset, introduced at the ImageCLEF 2019 competition for the VQA task [1]. The dataset contains 4200 images selected from MedPix database and 15 292 corresponding questions divided in the following way. For training 3200 images were allocated as well as 12 792 question-answer pairs; for validation 500 images with 2000 question answers pairs and for the test dataset, 500 images and questions.

The questions were divided into four different categories: Organ, Plane, Modality and Abnormality. The plane category includes images in 16 different planes, namely Axial; Sagittal; Coronal; AP; Lateral; Frontal; PA; Transverse; Oblique; Longitudinal; Decubitus; 3D Reconstruction; Mammo-MLO; MammoCC; Mammo-Mag CC and Mammo-XCC. The organ category has the smallest number of classes, the possible answers to all the questions belonging to a set of ten organs and organ systems namely: Breast; Skull and Contents; Face, sinuses, and neck; Spine and contents; Musculoskeletal; Heart and great vessels; Lung, mediastinum, pleura; Gastrointestinal; Genitourinary; Vascular and lymphatic. The modality category is slightly more complex than the previous two. There are 36 modalities, and the question can refer to the type

of modality used, either what or yes/no questions. There are also questions related to contrast/noncontract in the image, what type of contrast is used and specifics of MRIs (if the images are t1-weighted, t2-weighted or flared). In total, there are 44 possible answers for all modality questions. Therefore, the modality category includes yes/no questions, what question and other closed questions. The abnormality category includes both yes/no questions, that inquire about the state of the image; if it is normal/abnormal and what questions, that inquire about the abnormality shown in the picture. The abnormality category is the most complex, with 1485 possible answers in the training set. One concern that we will consider with this category is the large number of answers to the validation questions that are not found through the answers in training. This could be a possible issue when treating this problem as a classification since the model will not be able to learn the classes that are not present while training.

4. Proposed Approach

Following the lead of papers such as the ones proposed by Zhou et al. [22] and Al-Sadi et al.[2], we propose a model that divides the complex MVQA task into smaller and more manageable problems. By making use of the dataset knowledge, we can treat each individual problem as an image classification one and obtain comparable results with the current approaches from the literature. Unlike the presented approaches which use CNN based networks for image related tasks we choose to make use of an attention based state-of-the-art model for image classification, the Swin Transformer. We use pre-trained versions of Swin and finetune them to our specific classifications in order to obtain our results. We aim to see if the abilities of this model perform as well on these downstream tasks and moreover, if this transformer-based architecture can surpass the widely used CNN-based architectures. In order to apply the image classification models we need to divide the types of questions in a way that makes sense from the point of view of the created classes, therefore, we chose to create individual models for organ, as all answers are a type of organ or organ system, plane, as all answers in this category are planes in which the image is taken and abnormality. For the modality questions we created four models depending on the type of questions and possible answers that would create the classes. Therefore, we obtained a contrast model, used for questions that inquire about the way the image was taken, with contrast or noncontrast; a contrast type model, used for questions that inquire about the type of contrast used, possible answers being gi/iv/gi and iv; a type of weight model, for questions that examine whether the image is t1, t2 or flair

weighted and finally the modalities model which predicts classes representing the type of image modalities.

4.1. **Model Overview.** DOMAS is therefore a model that joins two types of models: a question classification model and several image classification ones. An overview of the flow model can be observed in Figure 1.

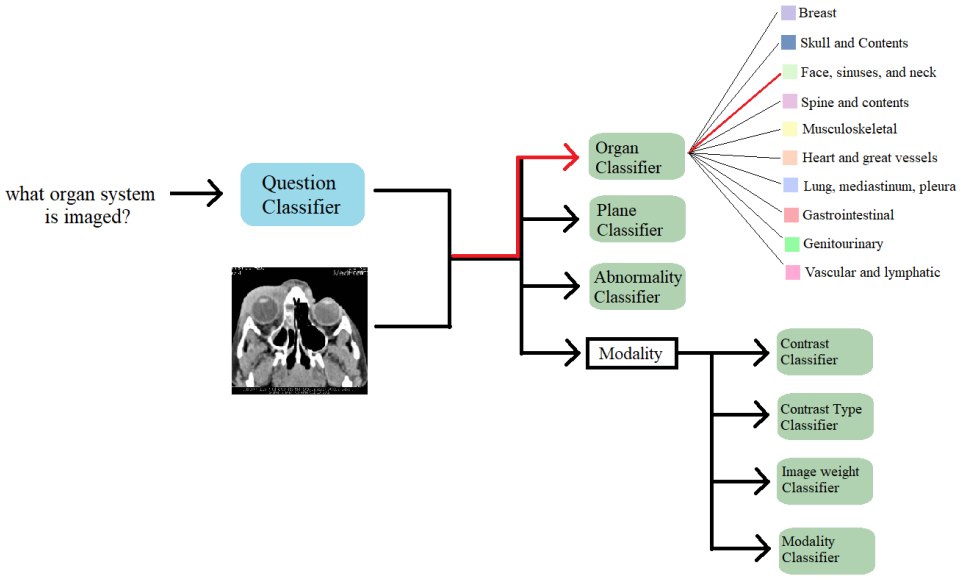


FIGURE 1. Model overview

As it can be seen, in Figure 1, the question is first passed through the question classifier which predicts which type of question it is organ, plane, modality or abnormality. Based on the predicted class an image classification model is selected. For example, in Figure 1 the question is classified as an organ question therefore the organ model is selected. For organ, plane and abnormality the corresponding model is selected. For modality, four models are available. To select the type of modality model we make use of dataset analysis. We observe that there is a limited number of questions for each of the four modality models in both train and validation that have as answer our classes. All questions that do not fit in one of these question lists and start with "is" or "was" are closed-ended questions, meaning the answer is either yes or no, and the remaining questions are image modality related questions. To handle yes or no questions, instead of treating them as classes and creating new models we further analyze them and make use of the initial four models.

We observe that each of the yes/no questions refers, in fact, to the information obtained using the previously mentioned models. Therefore, we create a function that extracts the type of modality model from the question and also the expected answer. For example, from the question "is this a t1 weighted image?" we extract the type of model, which is the weighting model, and also the expected class which is "t1". Therefore, we fed the image into the weighting classification model and if the class predicted by the model matches the expected one, we predict the answer "yes", and "no" otherwise. For modality type closed-ended questions we notice that the questions only enquire about "MRI" and "CT" scans, we return either "CT" or "MR" as the expected class. However, these answers are not classes for the modality model on their own. For that reason, we replace the perfect match for the yes answer with an inclusion. For example, if the expected class is "CT" and the predicted class is "CT - myelogram" we return the answer "yes" as "CT" is contained in the answer.

After the correct model has been selected, the image is given as input to the model and the predicted class is obtained and transformed from its numeric representation used by the model into the textual one using the corresponding model dictionary completing the inference.

4.2. **Models Architectures.** The total of eight models, one for text and seven for images, have been trained individually. More details about the architectures and training process are available in this subsection.

The question classification model is used for differentiating between the four major types of questions. It is based on a pretrained BERT [6] model that we finetune for our classification. The question is pre-processed by applying the BERT tokenizer. The model consists of the pre-trained BERT, followed by a dropout layer and a linear layer produces the final prediction. Lastly, ReLU activation is applied. One concern regarding this model was that it might affect the overall accuracy of the model by misclassifying the questions which would result in an erroneous result from the start. Fortunately, the model achieves 100% accuracy on the task thus eliminating the concern and providing the model with a greater generalization power than a classification based on pattern matching.

For the image classification models we expand the dataset by using image augmentations. After testing with several augmentation types and excluding the ones that could alter the image in such way that the label would no longer be fitting, such as random rotations, horizontal and vertical flips, we decided to use random resized crop with a size of 224, which randomly crops the image and that resize it to the given size, as it was the version that yielded the best improvements.

For each type of image model we experimented with a Swin-based classifier. The architecture of the model can be observed in Figure 2.

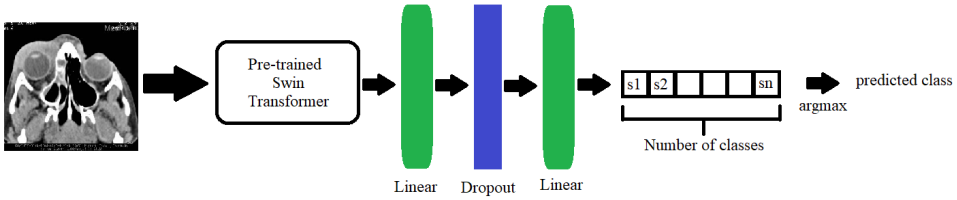


FIGURE 2. Image classification model architecture

As can be seen, the resized image enters the Swin model pretrained on the ImageNET dataset [5]. We experiment with different model versions such as tiny, small and base. The head of the model is modified in order to change the ImageNet classification task with our task. Therefore, the output of the Swin backbone is passed first through a linear layer. Next, ReLU activation and a dropout layer [15] are applied and finally the last linear layer obtains the final class. After analyzing the dataset, we observed that some classes are not present in the training dataset but appear in the validation dataset. We eliminated these classes, namely, "pet-CT fusion" from the modality split and "Mammo-XCC" from the plane classes. After this process the number of classes were 10 for organ classifier, 14 for plane, 1485 for abnormality and 44 for modality which were split into two for contrast mode, three for weighted model, three for contrast type model and 34 for modalities; the remaining two were the yes/no answers.

We chose Cross Entropy Loss since we performed multi-class classification and the Adam [9] and SGD optimizers depending on the case. Parameter settings and other implementation details will be further detailed in Section 5.

More details about hyperparameters settings as well as the results obtained by our model will be presented in Section 5

5. EXPERIMENTS AND RESULTS

5.1. **Experimental Setup and Results.** We trained and evaluated our models using a Google Colaboratory environment. The training was completed for each model using the integrated Nvidia T4 GPU. As mentioned before, we trained each model individually and selected the best models based on the classification accuracy. For evaluating the model we used two metrics namely the strict accuracy and Bleu score [11]. The parameter settings of the

final models can be seen in Table 1. The parameters were chosen empirically. To select the Swin version for each model we performed experiments with the pre-trained tiny, small and base versions and selected the best performing one based on accuracy and F1-score. If different versions obtained the same results we selected the smallest model between them.

Model	**Swin Version**	**Optimizer**	**Linear Layer Output Dim**	**Activation Function**	**Dropout Rate**
Organ	Tiny	Adam	384	ReLU	0.5
Plane	Tiny	Adam	384	ReLU	0.5
Abnormality	Small	Adam	1536	ReLU	0.5
Modality Contrast	Small	SGD	384	ReLU	0.5
Modality Contrast Type	Tiny	Adam	128	ReLU	0.5
Modality Weighting	Base	Adam	384	ReLU	0.5
Modality Modalities	Small	Adam	384	ReLU	0.5

TABLE 1. Parameters Setting for the employed models

We used the models with the configurations presented in Table 1 in combination with the question classifier, which achieved 100% accuracy and obtained the final results which are presented in table 2

Metric	**Organ**	**Plane**	**Modality**	**Abnormality**	**Overall**
Strict Accuracy	0.744	0.824	0.824	0.072	0.616
BLEU Score	0.789	0.838	0.774	0.215	0.654

TABLE 2. Model Results

As we can see from the results, our model obtains a 61.6% score in strict accuracy and 65.4% BLEU score. The lowest performing model is as expected the abnormality model since the number of classes is indeed the largest. We can also observe that some models have high BLEU scores compared to accuracy which could mean that the model does not give a perfect answer but could give a close one.

5.2. **Discussion and comparison to related work.** Our model achieves promising results on the MVQA 2019 dataset, especially for the plane, organ and modality questions. For the abnormality model the high number of classes corresponding to the answers as well as the existence of many classes in both validation and test dataset which are not found in the train set lead to a lower performance. For the other models we will further discuss the results in order to better understand the strong points and the shortcomings of the models.

For the organ model, we found that some questions in the validation and test dataset have more than one organ system given as answers. More specifically, in the test dataset there are nine such answers out of the 125. This is not a case that we treated during training or as postprocessing, therefore the model cannot make a correct prediction from the perspective of strict accuracy. However, when analyzing these answers compared to the predictions, we found out that in seven out of the nine cases our model predicted an organ system that was part of the answer which partially increased the Bleu score. In order to analyze if there is a pattern in the misclassifications of the model, we constructed the confusion matrix, Figure 3 (left), from which we left out the answers that have more than one organ system. We can observe that the most frequent misclassifications are between face, sinuses and neck and skull and contents or gastrointestinal and genitourinary and vascular and lymphatic, but most classifications are indeed correct.

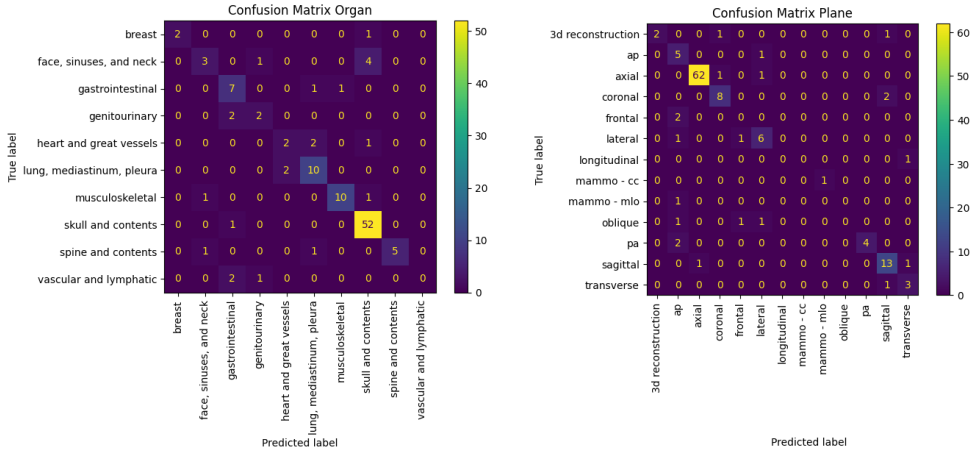


FIGURE 3. Confusion Matrices for Organ and Plane

We constructed a confusion matrix for the plane model, Figure 3 (right), classification on the test set as well. As we can see the classes found while testing are fewer than the ones found in the train dataset. For this model there

are fewer misclassifications, as expected after seeing the metrics. However, we can observe that the mistakes are more frequent for the less represented classes.

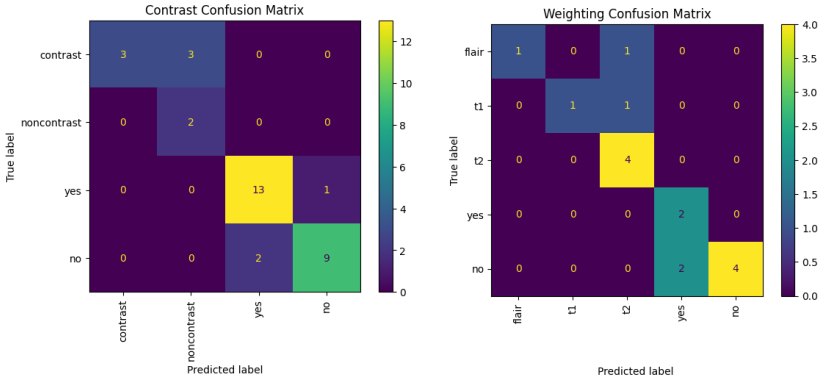


FIGURE 4. Confusion Matrices for Contrast and Weighting classifications

For the modality models we took a closer look at the results of the contrast, weighting and modality models. Even though our models do not treat yes and no as classes in the classification we constructed the matrices based on the final answer given by the modality model which was constructed to give an affirmative or negative answer as described in Section 4. For the contrast classification, one class that did not appear in training or validations set was found when testing therefore we removed the singular answer. We can see in Figure 4, that the model tends to predict the noncontrast class instead of contrast one rather than the other way around, which we also found to be true when analyzing more deeply the true meaning of the yes and no answers. For the weighting model we observed that there is a tendency to predict 't2' type which is the best represented class out of the modality weighting types.

For the modalities confusion matrix, Figure 5, we used again the test classification data. We notice that many discrepancies between the predicted and true classes are generated by the different types of CT scans which explains the good prediction for yes or no answers since they only inquire whether the image is or not a CT scan or an MR image meaning that the prediction includes more classes which are usually only confused with one another. We can also see confusions between different types of ultrasounds or similar classes which explains why the Bleu score is higher that the strict accuracy.

After analyzing the results of the model overall as well as individually for each component we can affirm that using Swin as an image classification method for this dataset offers promising results especially for the organ, plane and individual modality models. Moreover, our approach to the models for

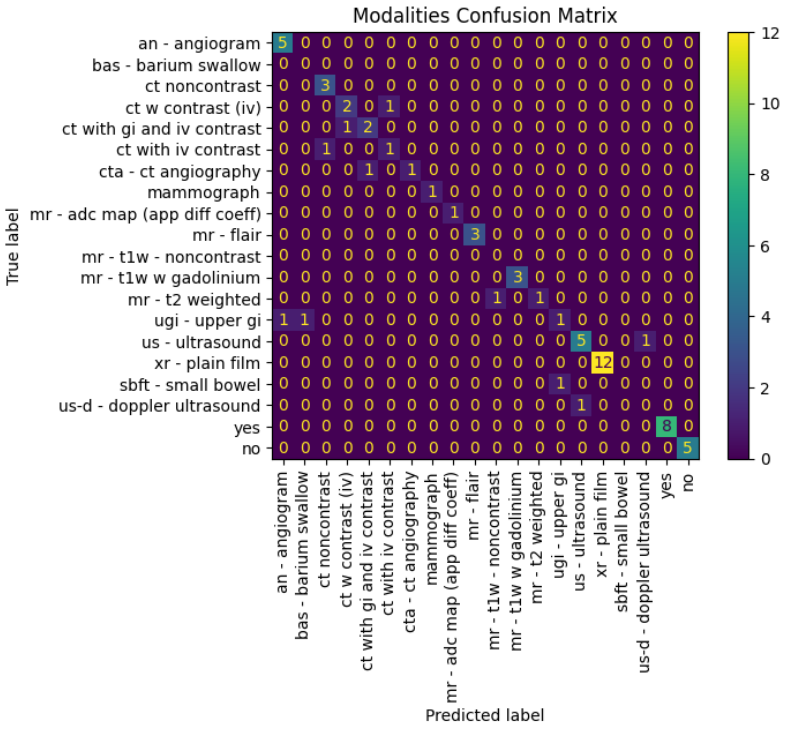


FIGURE 5. Confusion Matrix for Modalities classification

the modality questions makes these sub-tasks suitable for image classification by giving each model meaningful classes to discern from. By not using the yes and no answers as classes by themselves, but rather understanding from the question what the desired information is and constructing the inference accordingly we created models that can in fact predict the yes and no answers correctly, most of the time, without the need for extra classes or models.

In order to have the best understanding of where our results stand, we compared our model with other literature approaches, namely the first five teams of the Image-CLEF 2019 competition according to the dataset as well as the JUST [2] team since they also used an image classification approach and the two transformer-based approaches we mentioned in Section 2. The comparative results can be observed in Table 3, the results obtained by our proposed model being highlighted. As the test set was the same for all the papers, we did not replicate the experiments but rather got the corresponding results from each paper.

Model	Organ	Plane	Modality	Abnormality	Overall Accuracy	Overall BLEU
Henlin [22]	0.736	0.768	0.808	0.184	0.624	0.644
Yan [22]	0.736	0.768	0.808	0.168	0.62	0.640
Minhvu [18]	0.76	0.776	0.84	0.088	0.616	0.634
TUA1 [22]	0.792	0.816	0.744	0.072	0.606	0.633
UMMS [13]	0.736	0.76	0.672	0.096	0.566	0.593
JUST [2]	0.704	0.728	0.64	0.064	0.534	0.591
CGMVQA [12]	0.784	0.864	0.819	0.044	0.64	0.659
MMBERT [8]	0.768	0.864	0.833	0.14	0.672	0.69
DOMAS	**0.744**	**0.824**	**0.824**	**0.072**	**0.616**	**0.654**

TABLE 3. Results compared with literature approaches

Compared to the models submitted for the competition our model achieved the highest Bleu score and achieved the third score in accuracy. As for the individual models, it achieves the highest accuracy for the plane classification, ranks second for modality and third for plane. Compared to the JUST team which had an image classification approach as well but used VGG as a backbone for classification, our results rank higher in all the categories which proves that the Swin Transformer is a very suitable option for this task and could potentially be seen as a good alternative to the CNN based networks that are very popular choices for the MVQA task. Compared to the two transformer-based models, our model does not obtain better results, but the results are quite close. Our model surpasses the CGMVQA model for modality and abnormality results and it obtained a very close Bleu score. The MMBERT does perform better in all categories, which shows the great impact of extra training on more data.

Overall, our model obtains competitive results with the state-of-the-art models in all categories except the abnormality one where the high number of classes and small amount of data take a toll on the model's ability to effectively classify all the abnormality answers. We plan on further improving these results using various methods that are detailed in Section 6.

6. Conclusions and Future Work

In conclusion, we proposed DOMAS, a model that breaks down the complex MVQA task into multiple image classification tasks by processing the questions using a BERT-based architecture and knowledge we extract while performing exploratory data analysis on our dataset, Med-VQA 2019. Our approach proposes a Swin Transformer backbone for the image classification models as well as a unique way to select which models need to be developed based on the nature of the question in a way that all classes make sense from an image classification point of view. Our model achieves 0.616 score in strict accuracy and 0.654 BLEU score which ranks it the third in accuracy and first in BLEU among the participants in the ImageClef 2019 competition. The obtained results are also comparable with current state-of-the-art transformer-based model.

Our method shows that using the Swin Transformer architecture for working with images is beneficial in this task and could be seen as a viable alternative for the more popular CNN based networks which answers our first research question. Our model performs well for the organ, plane and modality models and we observe that our original approach of splitting the modality questions into four subcategories and obtaining the yes and no answers as a postprocessing of the model's output based on dataset knowledge rather than treating the answers as classes drastically improves the results in this category, making the response to our second research question an affirmative one. Moreover, using a BERT-based classifier, as opposed to a simpler pattern matching, for the questions also provides a certain generalizing power to the model even though the questions provided by the dataset are limited and quite redundant. However, the model also has some shortcomings such as lower level of robustness since we do use dataset specific knowledge and also treat the problem as a classification which makes the answer dependent on the training classes. We can observe this issue in the abnormality model where one cause of the lower performance could be the large number of answers in the test and validation datasets that are not present while training.

Future work plans include addressing some of these drawbacks. We plan on replacing the pattern matching methods used in the modality model classification with an intelligent approach that would predict the correct model as in the case of the question classification. Moreover, for yes and no questions we would aim to obtain the model as well as the expected class that would later be compared with the prediction in order to obtain the affirmative or negative answer. We also believe that the organ model could be improved by treating the case of multiple organ systems given as answer. As there is no information in the questions that could indicate that the expected answer is a compound

one, we believe that one viable approach would be to return all answers which confidence exceeds a certain threshold. Finally, to improve the abnormality model, we would like to explore the possibility of using a generative model instead of a classification one or to use extra data for training this current classification model.

References

[1] Abacha, A. B., Hasan, S. A., Datla, V. V., Liu, J., Demner-Fushman, D., and Müller, H. VQA-Med: Overview of the medical visual question answering task at ImageCLEF 2019. *CLEF (working notes) 2*, 6 (2019).

[2] Al-Sadi, A., Talafha, B., Al-Ayyoub, M., Jararweh, Y., and Costen, F. JUST at ImageCLEF 2019 Visual Question Answering in the Medical Domain. In *CLEF (working notes)* (2019).

[3] Bao, H., Wang, W., Dong, L., Liu, Q., Mohammed, O. K., Aggarwal, K., Som, S., and Wei, F. VLMo: Unified Vision-Language Pre-Training with Mixture-of-Modality-Experts, 2022.

[4] Chen, X., Wang, X., Changpinyo, S., Piergiovanni, A., Padlewski, P., Salz, D., Goodman, S., Grycner, A., Mustafa, B., Beyer, L., et al. PaLI: A Jointly-Scaled Multilingual Language-Image Model. *arXiv preprint arXiv:2209.06794* (2022).

[5] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), pp. 248–255.

[6] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805* (2018).

[7] He, K., Zhang, X., Ren, S., and Sun, J. Deep Residual Learning for Image Recognition, 2015.

[8] Khare, Y., Bagal, V., Mathew, M., Devi, A., Priyakumar, U. D., and Jawahar, C. MMBERT: Multimodal BERT Pretraining for Improved Medical VQA. In *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)* (2021), IEEE, pp. 1033–1036.

[9] Kingma, D. P., and Ba, J. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).

[10] Liu, Z., Lin, Y., Cao, Y., Hu, H., Wei, Y., Zhang, Z., Lin, S., and Guo, B. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. In *Proceedings of the IEEE/CVF international conference on computer vision* (2021), pp. 10012–10022.

[11] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (2002), pp. 311–318.

[12] Ren, F., and Zhou, Y. CGMVQA: A New Classification and Generative Model for Medical Visual Question Answering. *IEEE Access 8* (2020), 50626–50636.

[13] Shi, L., Liu, F., and Rosen, M. P. Deep Multimodal Learning for Medical Visual Question Answering. In *CLEF (working notes)* (2019).

[14] Simonyan, K., and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015.

[15] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The journal of machine learning research 15*, 1 (2014), 1929–1958.

[16] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Proceedings of the AAAI conference on artificial intelligence* (2017), vol. 31.

[17] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention Is All You Need. *Advances in neural information processing systems 30* (2017).

[18] Vu, M., Sznitman, R., Nyholm, T., and Löfstedt, T. Ensemble of Streamlined Bilinear Visual Question Answering Models for the ImageCLEF 2019 Challenge in the Medical Domain. In *CLEF 2019-Conference and Labs of the Evaluation Forum, Lugano, Switzerland, Sept 9-12, 2019* (2019), vol. 2380.

[19] Wang, W., Bao, H., Dong, L., Bjorck, J., Peng, Z., Liu, Q., Aggarwal, K., Mohammed, O. K., Singhal, S., Som, S., et al. Image as a Foreign Language: BEiT Pretraining for All Vision and Vision-Language Tasks. *arXiv preprint arXiv:2208.10442* (2022).

[20] Yan, X., Li, L., Xie, C., Xiao, J., and Gu, L. ImageCLEF 2019 Visual Question Answering in the Medical Domain. *Zhejiang University* (2019).

[21] Yu, Z., Yu, J., Fan, J., and Tao, D. Multi-modal Factorized Bilinear Pooling with Co-Attention Learning for Visual Question Answering. In *Proceedings of the IEEE international conference on computer vision* (2017), pp. 1821–1830.

[22] Zhou, Y., Kang, X., and Ren, F. TUA1 at ImageCLEF 2019 VQA-Med: a Classification and Generation Model based on Transfer Learning. In *CLEF (Working Notes)* (2019).

Department of Computer-Science, Faculty of Mathematics and Computer Science, Babeș-Bolyai University, Cluj-Napoca, Romania

Email address: teodora.toader@stud.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 1, 2023
DOI: 10.24193/subbi.2023.1.05

FIELD EXPERIMENT OF THE MEMORY RETENTION OF PROGRAMMERS REGARDING SOURCE CODE

ANETT FEKETE AND ZOLTÁN PORKOLÁB

Abstract. Program comprehension is a continuously important topic in computer science since the spread of personal computers, and several program comprehension models have been identified as possible directions of active code comprehension. There has been little research on how much programmers remember the code they have once written. We conducted two experiments with a group of Computer Science MSc students. In the first experiment, we examined the code comprehension strategies of the participants. The students were given a task to implement a minor feature in a relatively small C++ project. In the second experiment, we asked the students 2 months later to complete the same task again. Before starting the clock, we asked the students to fill a questionnaire which aimed to measure program code-related memory retention: we inquired about how much the students remembered the code, down to the smallest relevant details, e.g. the name of functions and variables they had to find to complete the task.

After the second experiment, we could compare the solution times of those students who participated in both parts. As one result, we could see that these students could solve the task in shorter time than they did in the first experiment. We also looked at the results of the questionnaire: the vast majority of students could not precisely remember more than two or three identifiers from the original code. In this paper, we will show how this result compares to the forgetting curve.

1. Introduction

Software development is a knowledge-intensive and complex task that demands programmers to master and utilize vast amounts of information. Programmers need to have a deep understanding of programming languages, algorithms, and design patterns, among other things. Moreover, the retention of knowledge and memory of previously written and read code is essential for

Received by the editors: 1 March 2023.

2010 *Mathematics Subject Classification.* 68U99.

1998 *CR Categories and Descriptors.* I.m [**Computing Methodologies**]: Miscellaneous; J.m [**Computer Applications**]: Miscellaneous .

Key words and phrases. code comprehension, memory retention, experiment.

program comprehension and efficient coding. The memory retention of coding concepts is important for programmers' productivity, as they must be able to recall previous code when creating new programs.

The ability to remember code and programming concepts is a critical component in the development process, but it is not always clear how long the retention lasts, or how it impacts performance. To address these questions, this paper investigates the effect of forgetting on source code comprehension and task solving time. We also examine whether programmers tend to remember code details or larger units, such as functions or algorithms. By answering these questions, we can gain a better understanding of the cognitive processes involved in programming and provide insights into how programmers can optimize their performance by retaining and recalling code more effectively.

In order to measure memory retention regarding source code, we conducted two experiments with Computer Science MSc students, in which the participants had to solve the same programming task and answer memory-related questions. The experiments took place two months apart. After the second experiment, we investigated the memories the participants had of the task, and how that influenced their solution time. We asked the students to describe their memories with as much details as possible in an essay question, and asked them to fill a multiple-choice question which targeted the remembrance of exact source code details.

In this paper, we attempt to answer the following research questions in connection with source code comprehension and memory retention:

- **RQ1:** How does forgetting affect source code comprehension and task solving time?
- **RQ2:** Are programmers more likely to remember the details of the code, or larger units like functions or algorithms?

The rest of the paper is structured as follows: In Section 2, we present earlier research about memory retention and programming experiments. In Section 3, we describe the details of both experiments, putting more focus on the second one. Section 4 contains the results of the second experiment. In Section 5, we mention the possible threats to the validity of our study. Finally, we conclude the paper in Section 6.

2. Related work

Our work is focused on the memory retention of programmers regarding source code through two experiments in which the participants were given a programming task to solve. In this section we present related research to show how other studies conducted experiments that were centered around the work

of software developers, and attempts to measure how programmers remember source code.

2.1. **Program comprehension experiments.** Programming tasks require cognitive effort and mental models from programmers, and can affect them physically. Many experiments have investigated program comprehension and computer science students, examining various aspects. For example, Nakagawa et al. measured cerebral blood flow during program comprehension and found that more complex code increased mental workload [14]. Andrzejewska and Skawińska tracked eye activity and found that external conditions and cognitive load affected comprehension speed [1]. Krüger et al. examined feature traceability and program decomposition and found that feature traces were helpful in solving tasks more quickly, while program decomposition hindered it [10]. They also found based on developer interviews that self-assessments are reliable sources of developer-related information, and programmers tend to be correct when they recall memories on project-related questions that they consider important [11]. Their findings confirm the study of Koenemann and Robertson who investigated the analysis methods of professional developers, and found that they focus on the software parts that they perceive as relevant to them [9]. Cornelissen, Zaidman, and Dursen investigated trace visualization and found that it could speed up task solving by 22% [3]. Kather and Jan found that program comprehension and algorithm comprehension are not the same, and that domain knowledge, experience, and abstract knowledge can help solve tasks more quickly [8].

2.2. **Memory retention and forgetting.** In 1885, Ebbinghaus defined the so-called "forgetting curve" [4] (see Figure 1) after a series of experiments in which his subjects tried to remember randomly selected words. The most important factor of the formula is time. The original experiments of Ebbinghaus were since then replicated, confirming the correctness of the formula with some small modification in its smoothness [13], and it has also been investigated in the context of brain function [16]. The psychological experiments of Averell and Heathcote [2] confirmed that the exponential curve is the best fit to model human forgetting.

Forgetting and memory retention has been scarcely researched from a software development and source code aspect. Some studies that utilize the forgetting curve include the work of Xu et al. who investigated the concreteness and readability of identifiers in the source code based on how easily programmers remembered them [18]. One study that is closer to our goals is the work of Ünal et al. who looked at how repeated exposure to the same source code helps solving programming tasks [17]. Kang and Hahn found in their study

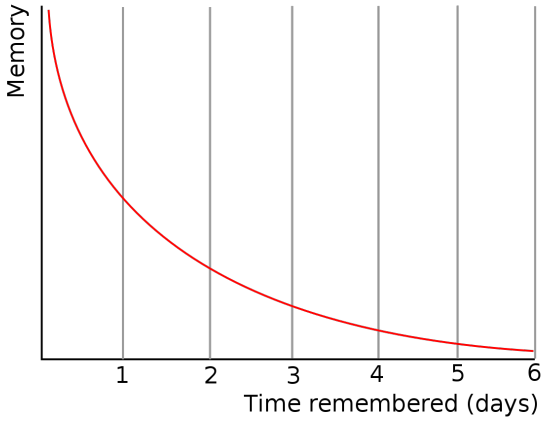


FIGURE 1. The forgetting curve as described by Ebbinghaus in the 19th century.

that forgetting affects methodological knowledge more than technology-related knowledge [7]. Most similarly to our research objectives, Krüger et al. examined whether the forgetting curve is applicable in remembering source code. Their experiment covered hours and days during which the programmers were asked to recall memories of the source code [12]. Our aim is to investigate whether programmers are likely to remember abstract levels and details of the source code after a longer time period.

3. EXPERIMENTS

We planned two experiments in advance: in the first one, we targeted the code comprehension strategies of junior programmers. The goal of the second experiment was to gain an understanding of the memory retention of programmers of source code.

We asked Computer Science MSc students from Eötvös Loránd University to take part in the experiment. The students were all enrolled in the *Multi-paradigm programming* course whose main topic is advanced C++. A total of 27 students took part in the first experiment, and 16 of them took part in both. We considered MSc students to be better experimental subjects, since they generally have more experience in programming (both as a job and as an activity), and because of that, they are more conscious about code comprehension and programming tasks.

Both experiments consisted of two main parts: first, the students were asked to fill a different questionnaire. Afterwards, the students were given a small

C++ task to solve in *TinyXML2*[1], a simple XML parser which contains only three C++ source files of hundreds of lines of code. The task was focused on code comprehension rather than writing new code: TinyXML2 is case-sensitive by default regarding XML tags. The students had to make the library case-insensitive by finding a particular line of code, and replace it with calling a function that we readily provided for them. Thus, we could measure code comprehension speed because the students only had to focus on understanding the code and finding the line in question instead of spending time with writing the replacement code.

The following function definition was provided for the participants:

```
#include <ctype.h>

int my_stricmp(const char* s1, const char* s2)
{
    while (tolower((unsigned char) *s1) == tolower((unsigned
        ↪ char) *s2))
    {
        if (*s1 == '\0')
            return 0;
        s1++; s2++;
    }
    return (int) tolower((unsigned char)*s1) - (int) tolower
        ↪ ((unsigned char)*s2);
}
```

The line to be modified was line 1142 in *tinyxml.cpp*:

```
...
else if ( !XMLUtil::StringEqual( endTag.GetStr(), ele->Name
    ↪ () ) ) {
...
```

The correct solution:

```
...
else if ( my_stricmp( endTag.GetStr(), ele->Name() ) ) {
...
```

We divided the students into two groups: one group had to use CodeCompass for code comprehension activities, the other group was free to use any code editor or code comprehension tool. The latter group formed the control

[1]TinyXML2 GitHub repo: `https://github.com/leethomason/tinyxml2`

group in the first experiment. CodeCompass [15] is an open-source code comprehension framework which applies static analysis to the source code and its environment (e.g. compilation database, version control repository), and provides various textual and visual support for understanding source code both on code level and file level.

3.1. **First experiment.** In the first experiment, we investigated the usual code comprehension strategies of young programmers, and how that correlates with factors such as the amount of experience as a programmer, and language familiarity. Building on our earlier study [6], we aimed to investigate the comprehension functionality that students used during task solution.

In the questionnaire of the first experiment we inquired about the amount of their work and general programming experience, as well as the languages they were most familiar with. As mentioned above, 27 students took part in this experiment: 15 had to use CodeCompass, and 12 were free to use any other tool.

Based on the participants' solution time and their answers to the questionnaire, we concluded that while more programming experience meant quicker task solution, work experience correlated more with solution time. The students in the CodeCompass group used our demo server[2], which collects anonymous user activity using Google Analytics. The activity log in CodeCompass showed that the students were majorly using top-down comprehension strategies.

The details and results of the first experiment are elaborated in our previous study [5].

3.2. **Second experiment.** The second experiment took part cc. two months after the first one. As mentioned above, 16 students took part in both the first and the second experiment, thus their results are relevant in this study.

The students were asked to solve the same programming task as in the first experiment: find the line of code in TinyXML2 in which the function call needs to be replaced with the provided function in order to make XML parsing case-insensitive.

We asked them to fill a different questionnaire the second time. The questions were related to source code memory retention:

- Essay question: *What do you remember from the first experiment? Please provide as much information as you can, any detail can be useful.*
- Multiple-choice question: *Which identifiers were in the program that you had to modify?* For this question we listed 9 correct and 21 false

[2]Demo server: `https://codecompass.net/`

identifiers. The false ones were most of the time very similar to the correct ones, or they were made to sound relevant in an XML parser.

All students had to use the same tool for comprehension activities they used in the first experiment. Our goal was to repeat the first experiment down to every possible detail, in order to remove any additional factors that might affect measuring memory retention. Both experiments were conducted in the same university computer lab, and the machines were equipped with the same hardware and software.

4. Results and discussion

By repeating the experiment, and asking the students about their memories of the first experiment, we wanted to investigate whether participants remember source code details or structure, and whether remembering details of the actual code correctly is correlated with quicker solution time. In our previous study, we collected the common elements of code comprehension models [6]. These elements usually rely on how the code is written syntactically (e.g. beacons are "visual cues" in the code the programmer is looking for to identify the meaning of a source code unit), this is why we focus on remembering actual identifiers.

We evaluated the students' responses to the essay question. 14 out of 16 students remembered the task clearly, and 9 students described steps of their previous solution. It is worth noting, that multiple students explicitly stated in their response that they do not usually remember exact identifiers of any source code, instead they remember structural details.

Table 2 shows the number of correct and incorrect guesses of the multiple-choice question for each student. We calculated the χ-square test for the answers of the question to determine whether the distribution of responses is significantly different from what would be expected by chance. Table 1 represents the contingency table of the calculation. We divided the responses into two categories, marked and not marked.

Equation 1 shows the results of the test. The degree of freedom in the calculation was 1, and the original significance was $p < .05$. The statistic value and the calculated significance suggest that the students' guesses were influenced by the correct vs. incorrect nature of the answer. Equation 2 shows the statistics of the test with Yates correction: the results in this case did not change the conclusion, the null hypothesis (that the students' answers are independent of correctness of the answer) remains rejected.

$$\chi^2(1) = 21.42, p < .00001 \tag{1}$$

	Correct answers	**Wrong answers**
Marked	58	68
Not marked	77	247

TABLE 1. The contingency table used in the χ-square test for the evaluation of the multiple-choice question in which we examined if the students remember identifiers correctly.

$$\chi^2(1) = 20.37, p < .00001 \tag{2}$$

In Table 2 we also listed the solution times of each student who took part in both experiments. Comparing the two experiments, the solution times show an average improvement of 16 minutes and 20 seconds. If we look at the individual solution times, we can see that students performed better in all cases we knew both solution times.

The proportion of correct vs. wrong guesses was greater or equal to 1 in the case of 8 students, while this number was below 1 for the other 8 students. Comparing the solution times, the students with better guess rate improved by 16.97 minutes on average, while the other 8 students decreased their average solution time by 15.6 minutes. The cc. 1.5-minute difference between the average improvements shows that remembering identifiers better correlates with quicker solution time. However, the significant improvement in solution times for all participants suggests that remembering the process of task solution is more significant than remembering exact identifiers in the source code. We included in the rightmost column of Table 2 if a student described steps or details of the solution in the essay question. 7 students who had more correct than wrong guesses reported such memories, while only 2 students remembered any details from the solution. This result suggests that remembering steps of an algorithm and exact details from a code base are correlated.

To answer **RQ1** *(How does forgetting affect source code comprehension and task solving time?)*, our data shows that the participants who reported more memories of the first experiment - either in the form of actual identifiers or verbal descriptions of the task or the source code - performed better on average during the second experiment.

Solution times and responses to the questions suggest that the participants had statistically significant memories of the source code after two months of the initial experiment. To answer **RQ2** *(Are programmers more likely to remember the details of the code, or larger units like functions or algorithms?)*, the data suggests that there is a correlation between remembering exact details of the

source code and having more memories of the structure or steps of solving a programming task.

The findings in reply to RQ2 are complementary to a related study [11] that concluded from developer interviews that abstract knowledge of the source code is more important to remember. However, our results are somewhat contradictory of another study by Krüger et al. [12] who found that the forgetting curve applies to remembering source code. According to our results, the participants had a fairly good recollection of the solution process even after 2 months. This suggests that forgetting slows down after a certain amount of time, as we observed memory retention after two months, and the aforementioned study investigates remembering source code after some days.

Student #	Solution time #1 (mins)	Solution time #2 (mins)	Correct ids	Wrong ids	Detailed memories?
1	**11**	**3:05**	**7**	**7**	✓
2	N/A	25	5	7	✗
3	**33**	**8:03**	**3**	**2**	✓
4	19:50	N/A	2	2	✗
5	**N/A**	**14**	**6**	**4**	✗
6	11:30	8:54	5	6	✓
7	26:40	4:34	1	3	✓
8	**30**	**8**	**7**	**3**	✓
9	**7**	**1:20**	**4**	**4**	✓
10	23:48	16:54	2	7	✗
11	37	27	1	3	✗
12	59:48	19	2	3	✗
13	**24**	**2:50**	**4**	**3**	✓
14	**N/A**	**N/A**	**7**	**0**	✓
15	**12**	**4:30**	**4**	**2**	✓
16	26:47	15:35	5	12	✗

TABLE 2. The results of the second experiment. Comparing solution times we can see that all students performed better the second time which implies complex memory retention in spite of inconsistent remembrance of exact identifiers.

5. THREATS TO VALIDITY

As any research that relies on human resources and input, our study holds some obvious threats to validity.

Small number of participants. Although 27 students took part in the first experiment, only 16 of them was present during the second one. The students come from similar backgrounds considering their computer science education and programming experience. This might narrow down our research results regarding target population, making more experiments needed with a more diverse pool of participants.

Incomplete data. The questionnaire was available for the students on the Canvas learning management system. The responses of one student could not be found after the experiment. The solution times were also collected through Canvas, and some data was lost between the experiments, this is why a few solution times are missing from Table 2. The missing data is omitted in our calculations in order to avoid distorting results, hence the metrics in our results are computed for 12 students instead of 16, the total number of participants.

Short study period for the students. In our experiments, the students had one hour both times to study the source code of TinyXML2. In reality, a programmer spends much more time working on the same source code, so their memory retention of the code is probably stronger. However, our data shows that even with a short study period and after a longer intermission, the students were able to recall the comprehension process and solved the task quicker than the first time, which suggests that more time spent with the same code instills even stronger memories.

Effects of the first experiment. The students were aware that they were participating in an experiment both times which gives space to biased results as they might have paid more attention to the code and exercise than they would have had they not know about the experiment. However, at the time of the first experiment, they did not know there would be a second one so they had no direct reason to clearly remember the details of the first one after the 2-month break.

6. Conclusion

In this research, we presented the results of two consecutive experiments with Computer Science MSc students in which we investigated the effect of forgetting in source code comprehension and solving programming tasks. The students were asked to fill questionnaires and solve the same C++ programming task in both experiments. In the first experiment, we examined the code comprehension strategies of the students, and the correlation between task solution time, and work experience, general programming experience, and familiarity with programming languages. In the second experiment, we investigated how much the students remembered from the first experiment: we

asked them to describe their memories, and answer a multiple-choice question about the actual identifiers in the source code.

In total, 16 students took part in both experiments. We executed a χ-square test on the students' guesses in the multiple-choice question. The test showed that there is correlation between the correctness of an answer option and if it was guessed by a student.

The average solution time was decreased by 16.3 minutes on average among the participants. In case of the 8 students who marked at least as many correct identifiers as wrong ones in the multiple-choice question, the solution time improved by 15.6 minutes, and 16.97 minutes for the other 8 students. This result suggests that remembering the process of task solution is a more significant factor in code comprehension than remembering exact identifiers. The results of our research suggest that remembering both structural and code-level details contribute to quicker task solution, and that remembering more exact details of the source code correlates with the retention of more structural memories.

References

1. Magdalena Andrzejewska and Agnieszka Skawińska, *Examining students' intrinsic cognitive load during program comprehension–an eye tracking approach*, International Conference on Artificial Intelligence in Education, Springer, 2020, pp. 25–30.
2. Lee Averell and Andrew Heathcote, *The form of the forgetting curve and the fate of memories*, Journal of mathematical psychology **55** (2011), no. 1, 25–35.
3. Bas Cornelissen, Andy Zaidman, Arie Van Deursen, and Bart Van Rompaey, *Trace visualization for program comprehension: A controlled experiment*, 2009 IEEE 17th International Conference on Program Comprehension, IEEE, 2009, pp. 100–109.
4. Hermann Ebbinghaus, *Über das gedachtnis*, 1885.
5. Anett Fekete and Zoltán Porkoláb, *Report on a Field Experiment of the Comprehension Strategies of Computer Science MSc Students*, 2022 IEEE 16th International Scientific Conference on Informatics - Proceedings, IEEE, 2022, pp. 73–81.
6. Anett Fekete and Zoltán Porkoláb, *A comprehensive review on software comprehension models*, Annales Mathematicae et Informaticae, vol. 51, Líceum University Press, 2020, pp. 103–111.
7. Keumseok Kang and Jungpil Hahn, *Learning and forgetting curves in software development: Does type of knowledge matter?*, ICIS 2009 Proceedings (2009), 194.
8. Philipp Kather and Jan Vahrenhold, *Is algorithm comprehension different from program comprehension?*, 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), IEEE, 2021, pp. 455–466.
9. Jürgen Koenemann and Scott P Robertson, *Expert problem solving strategies for program comprehension*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 1991, pp. 125–130.
10. Jacob Krüger, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake, *Effects of explicit feature traceability on program comprehension*, Proceedings of the 2019 27th

ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 338–349.
11. Jacob Krüger and Regina Hebig, *What developers (care to) recall: An interview survey on smaller systems*, 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020, pp. 46–57.
12. Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich, *Do you remember this source code?*, Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 764–775.
13. Jaap MJ Murre and Joeri Dros, *Replication and analysis of ebbinghaus' forgetting curve*, PloS one **10** (2015), no. 7.
14. Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M German, *Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment*, Companion proceedings of the 36th international conference on software engineering, 2014, pp. 448–451.
15. Zoltán Porkoláb, Tibor Brunner, Dániel Krupp, and Márton Csordás, *Codecompass: an open software comprehension framework for industrial usage*, Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 361–369.
16. Dong Gue Roe, Seongchan Kim, Yoon Young Choi, Hwije Woo, Moon Sung Kang, Young Jae Song, Jong-Hyun Ahn, Yoonmyung Lee, and Jeong Ho Cho, *Biologically plausible artificial synaptic array: Replicating ebbinghaus' memory curve with selective attention*, Advanced Materials **33** (2021), no. 14, 2007782.
17. Utku Ünal, Eray Tüzün, Tamer Gezici, and Ausaf Ahmed Farooqui, *Investigating the impact of forgetting in software development*, arXiv preprint arXiv:2204.07669 (2022).
18. Weifeng Xu, Dianxiang Xu, and Lin Deng, *Measurement of source code readability using word concreteness and memory retention of variable names*, 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), vol. 1, IEEE, 2017, pp. 33–38.

Eötvös Loránd University,, Faculty of Informatics, Egyetem tér 1-3.,, 1053 Budapest,, Hungary
Email address: afekete@inf.elte.hu

Eötvös Loránd University,, Faculty of Informatics, Egyetem tér 1-3.,, 1053 Budapest,, Hungary
Email address: gsd@inf.elte.hu