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/`

```python
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

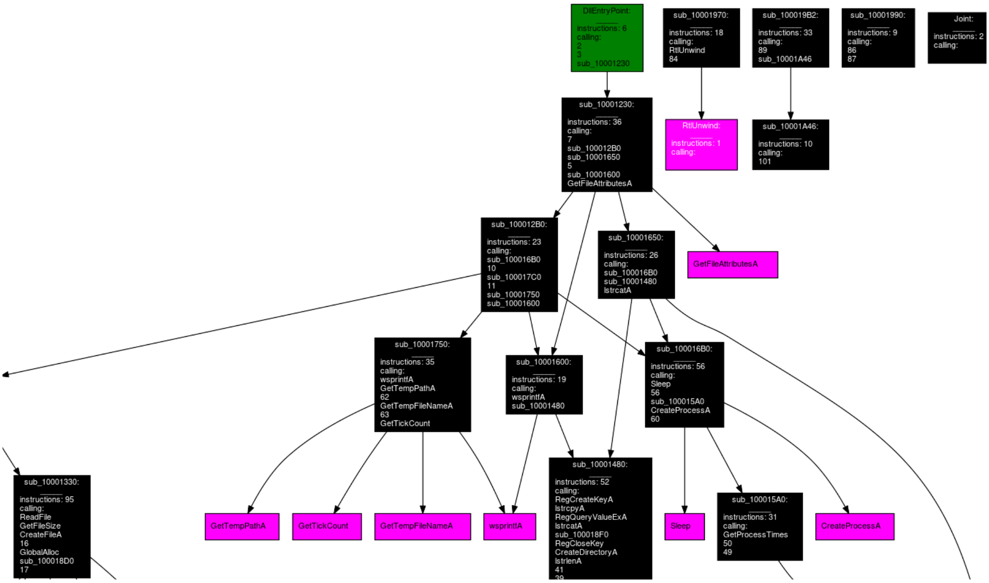

Figure 1. Call graph obtained with IDA Pro 6 (“GenCallGdl”).

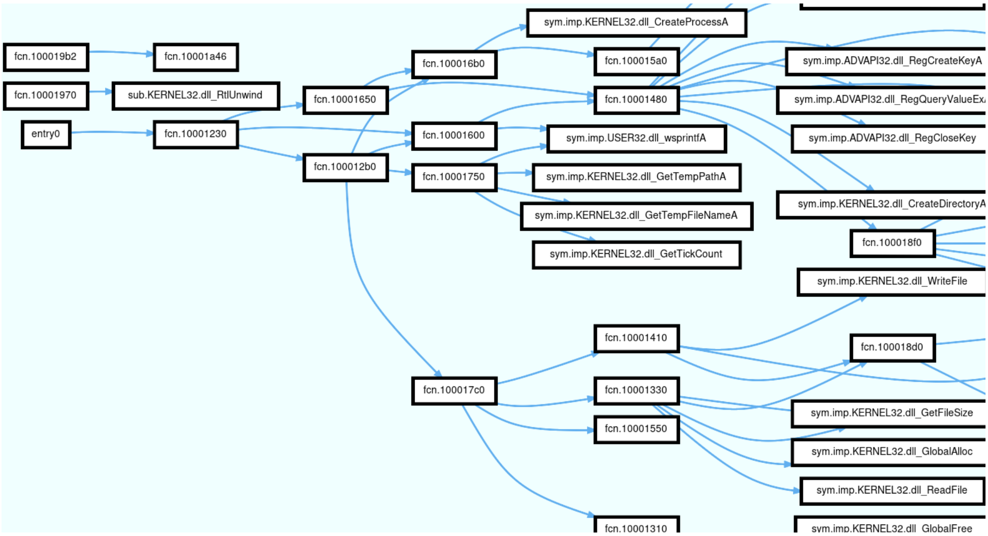
sym.imp.KERNEL32.dll_CreateProcessA
fcn.100019b2
fcn.10001a46
fcn.100016b0
fcn.100015a0
sym.imp.ADVAPI32.dll_RegCreateKeyA
fcn.10001970
sub.KERNEL32.dll_RtlUnwind
fcn.10001650
fcn.10001480
sym.imp.ADVAPI32.dll_RegQueryValueEx
entry0
fcn.10001230
fcn.10001600
sym.imp.USER32.dll_wsprintfA
sym.imp.ADVAPI32.dll_RegCloseKey
fcn.100012b0
fcn.10001750
sym.imp.KERNEL32.dll_GetTempPathA
sym.imp.KERNEL32.dll_CreateDirectoryA
sym.imp.KERNEL32.dll_GetTempFileNameA
fcn.100018f0
sym.imp.KERNEL32.dll_GetTickCount
sym.imp.KERNEL32.dll_WriteFile
fcn.10001410
fcn.100018d0
fcn.100017c0
fcn.10001330
sym.imp.KERNEL32.dll_GetFileSize
fcn.10001550
sym.imp.KERNEL32.dll_GlobalAlloc
sym.imp.KERNEL32.dll_ReadFile
fcn.10001310
sym.imp.KERNEL32.dll_GlobalFree

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

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

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

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

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`

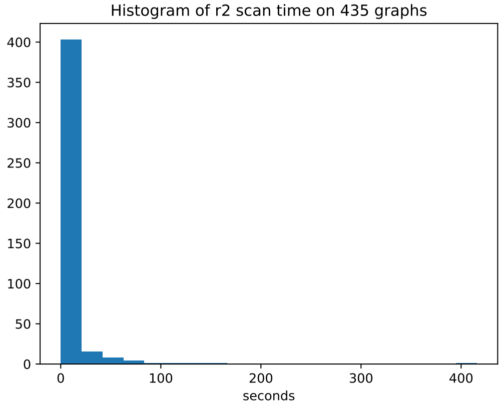

Figure 3. Runtime of the Radare2 scanner programme.

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

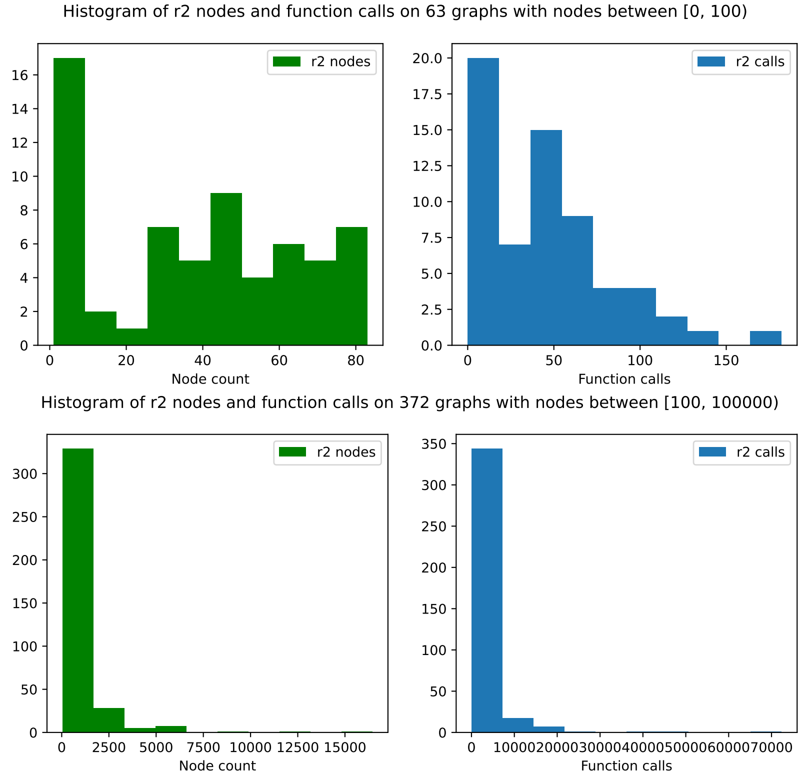

Figure 4. Histogram of call graph sizes using Radare2 (nodes and calls).

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.

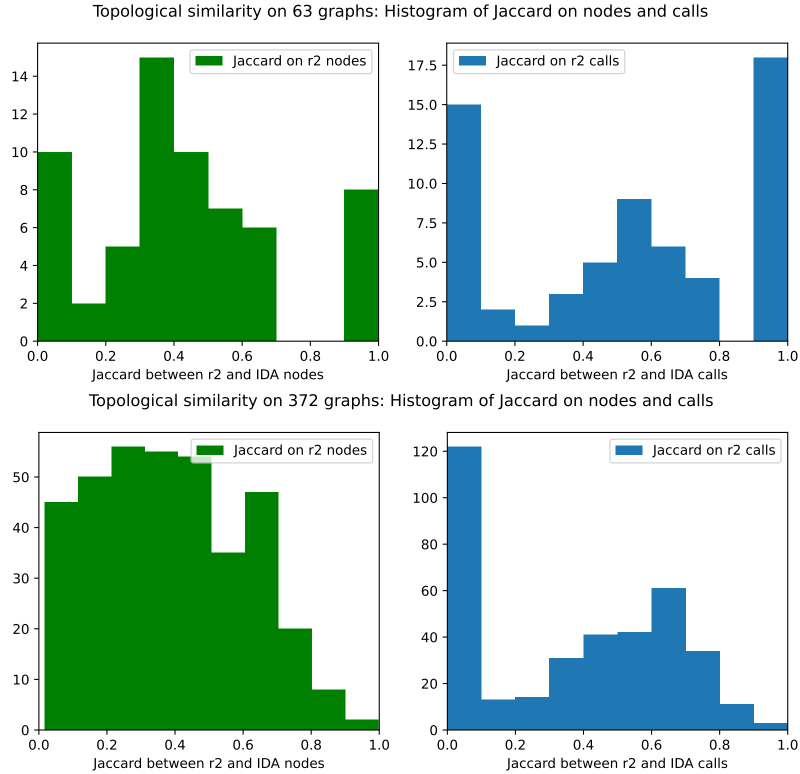

Figure 5. Topological similarity: histogram of Jaccard score between IDA and Radare2 nodes and calls.

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.

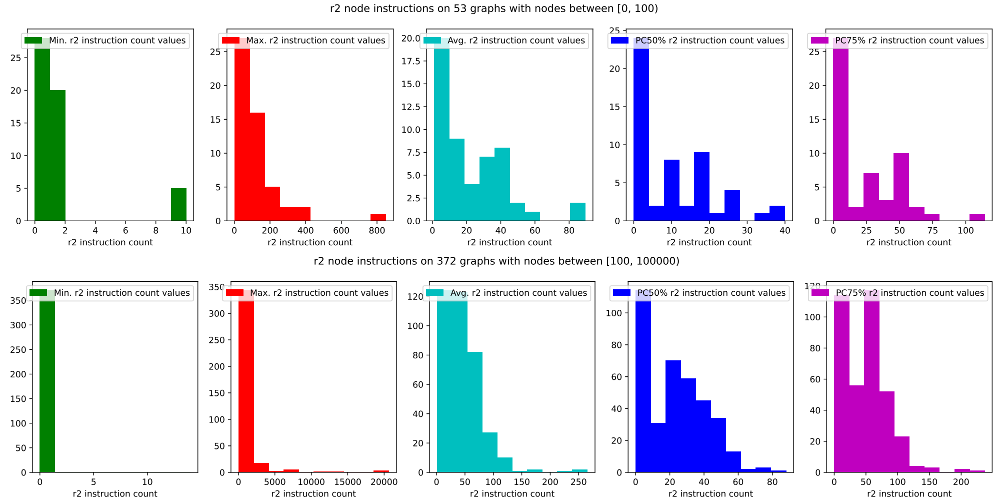

Figure 6. Histogram of Radare2 nodes' instruction count.

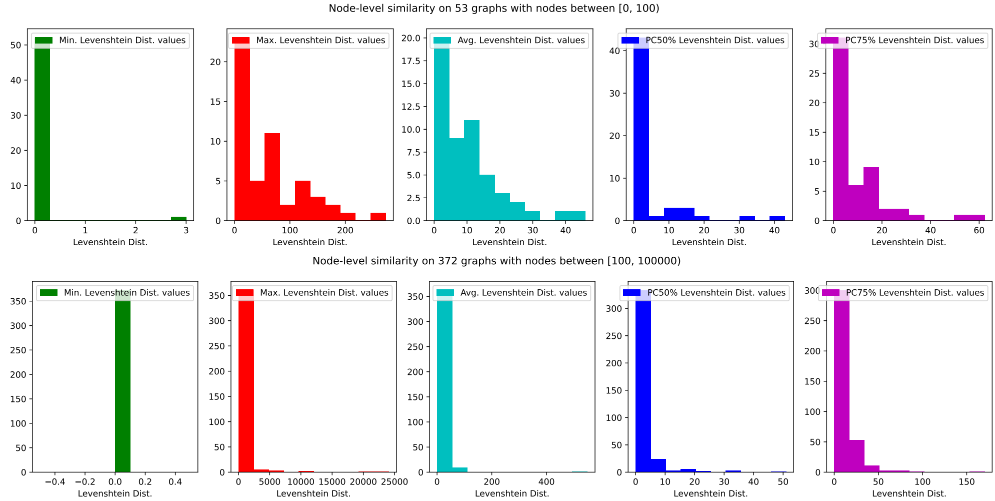

Figure 7. Histogram of Levenshtein distances between Radare2 and IDA nodes.

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

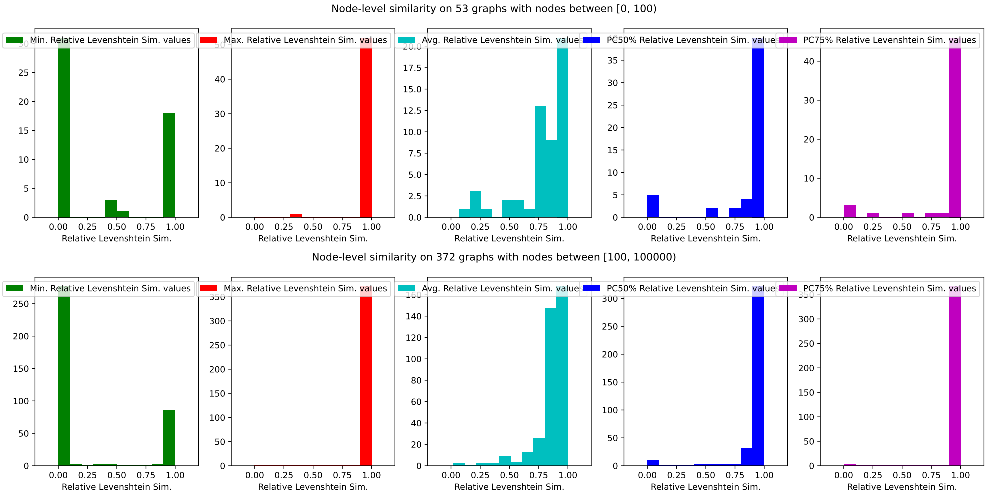

Figure 8. Histogram of relative Levenshtein similarities between Radare2 and IDA nodes.

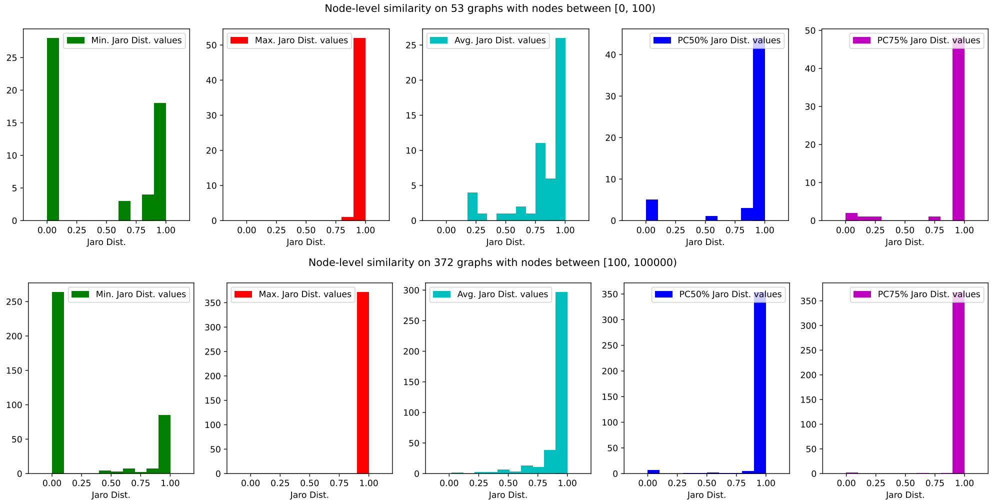

Figure 9. Histogram of Jaro distances between Radare2 and IDA nodes.

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.

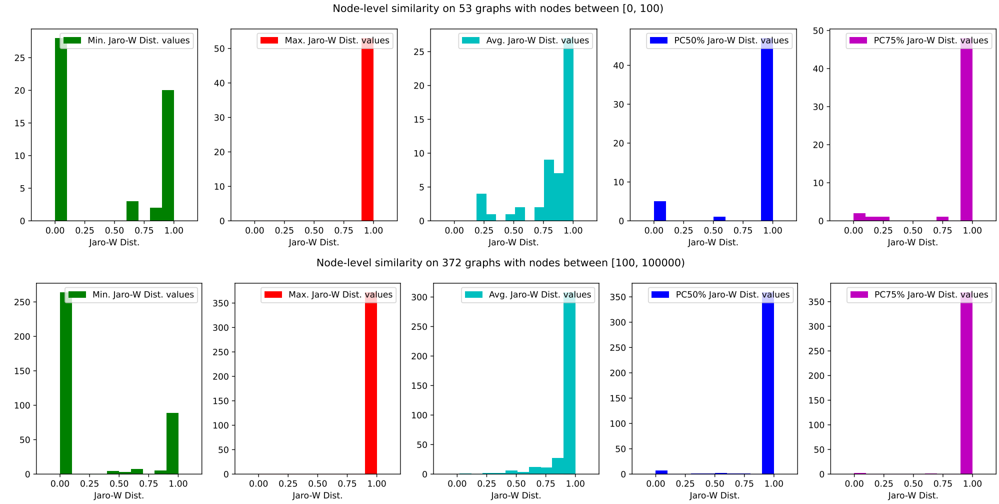

Figure 10. Histogram of Jaro-Winkler distances between Radare2 and IDA nodes.

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`