STAMAD: a STAtic MAlware Detector.

Khanh Huu The Dam, Tayssir Touili

STAMAD - a STAtic MAlware Detector

Khanh Huu The Dam
LIPN and University Paris 13
France

Tayssir Touili
CNRS, LIPN and University Paris 13
France

ABSTRACT

One of the main challenges in malware detection is the discovery of malicious behaviors. This task requires a huge amount of engineering and manual study of the code. To avoid this tedious manual task, we propose in this paper a tool, called STAMAD, that, given a training set of known malwares and benign programs, (1) either *automatically* extracts malicious behaviors using Information Retrieval techniques, or (2) applies machine learning techniques to *automatically* learn malwares. Then, in both cases, STAMAD can classify a new given unseen program as malicious or benign.

KEYWORDS

Malware detection, Malicious behavior extraction, Static analysis.

ACM Reference Format:
Khanh Huu The Dam and Tayssir Touili. 2019. STAMAD - a STAtic MAlware Detector. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES 2019) (ARES '19), August 26–29, 2019, Canterbury, United Kingdom.* 6 pages. https://doi.org/10.1145/3339252.3339274

1 INTRODUCTION

The number of malwares is increasing very fast, thus, malware detection is a big challenge. The well-known techniques for malware detection are based on signature (sequence of bytes) matching. However, this technique can easily be evaded by standard obfuscation techniques. Another technique for malware detection is code emulation where the behaviors of the malware are dynamically observed while running the program on an emulated environment. This technique is not robust since it is hard to get the malicious behaviors in a short period, as they may require a delay or only show up after user interaction. To sidestep the limitations of these techniques, we need to use static analysis approaches that check the *behaviors* (not the syntax) of a program *without executing it*. To this aim, we propose in this work to use extended API call graphs to represent malicious behaviors, where an extended API call graph is a directed graph whose nodes are API functions, and whose nodes and edges are annotated. An edge (f, f') expresses that there is a call to the API function f followed by a call to the API function f'. The annotation $\bar{i} \rightsquigarrow \bar{j}$ on the edge (f, f') means that the i-th parameter of function f and the j-th parameter of the function f' have a data dependence relation. It means that, in the program, either these two parameters depend on the same value or one parameter depends on the other. A node f with the annotation $\bar{i} = \{c\}$ means that the i-th parameter of function f gets the value c. This graph specifies the execution order of the API function calls as well as the links between the API functions' parameters. Indeed, API (which stands for Application Programming Interface) is a collection of functions supported by the operating system that allow users to interact with the system. These API functions are mediators between programs and their running environment (user data, network access...) that are mostly used to access or modify the system by malware authors. According to a statistic study [2], over 5TB of different samples of malwares, there are 527,992 samples that did import at least one API, compared to 21,043 samples with no import. Thus, API functions and their usages in the program are crucial to specify malicious behaviors. Figure 1(c) shows the extended API call graph of the assembly code in Figure 1(a). It expresses a self replication behavior in which the malware infects the system by copying itself to a new location. This is achieved by first calling the API function GetModuleFileName with 0 as first parameter and m as second parameter (Parameters to a function in assembly are passed by pushing them onto the stack before a call to the function is made. The code in the called function later retrieves these parameters from the stack.). This will store the file path into the memory address m. Then, CopyFile is called with m as first parameter. This allows to infect other files. To represent this behavior, [12, 16, 18, 23] use the API call graph in Figure 1(a) to express that calling GetModuleFileName is followed by a call to CopyFile. However, a program that contains this behavior is malicious only if the API function CopyFile copies the file returned by GetModuleFileName. If CopyFile copies another file, the behavior is not malicious. Thus, the above representation may lead to false alarms. To avoid this, we need to make the representation more precise and add the information that the returned parameter of GetModuleFileName should be the input argument of CopyFile. Therefore, we propose to use the Extended API call graph in Figure 1(c), where the edge labeled by $\bar{2} \rightsquigarrow \bar{1}$ means that the second parameter of GetModuleFileName (which is its output) is given as first argument of CopyFile. We also need to ensure that GetModuleFileName is called with 0 as first parameter. Thus, we label the node GetModuleFileName with $\bar{1} = \{0\}$ to express that the first parameter of this call should be 0. Thus, we propose in this work to use *Extended API call graphs* to represent malicious behaviors.
On the other hand, one of the most challenging tasks in malware detection is the discovery of malicious behaviors. This task requires a huge amount of engineering and manual study of the code. To avoid this manual step,we propose in this work to use learning and information retrieval: Given a training set of known malwares and benign programs, we apply a training phase, and we use the knowledge got from this training to analyse a new program and decide whether it is malicious or benign. We propose two approaches: our first approach allows to automatically extract the malicious

ARES '19, August 26–29, 2019, Canterbury, United Kingdom

ACM ISBN 978-1-4503-7164-3/19/08…$15.00
https://doi.org/10.1145/3339252.3339274

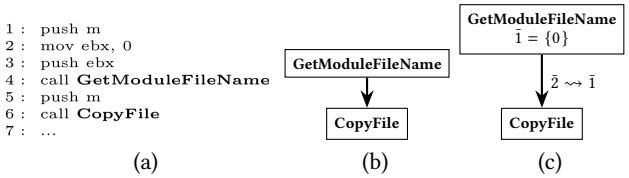


Figure 1: A piece of assembly code (a) of the behavior self replication, its API call graph (b) and its data dependence graph (c).

behaviors from the training set by applying Information Retrieval techniques [12]. These automatically extracted malicious behaviors are then used for malware detection. The second approach that we propose applies machine learning techniques to detect malwares without extracting the malicious behaviors [13]. We use extended API call graphs to represent programs and malicious behaviors in our both approaches. We implement our techniques in a tool for PC malware detection called STAMAD. In this paper, we present this tool.

Related Work BitScope [6] and Panorama [15], are two malware detectors. These tools are not robust since they are based on dynamic analysis, and thus, they may not trigger the malicious behaviors. In PyEA [7], the authors implement a malware detection technique based on control flow graph matching. This technique can easily be evaded by standard obfuscation techniques since one can easily change the control flow graph of a malware without changing its malicious behaviors. PoMMaDe [20] is a model-checking based tool for malware detection. However, currently, PoMMaDe can only detect 7 malicious behaviors (due to the fact that malicious behaviors are extracted by hand and given as input to PoMMaDe). Unlike PoMMaDe, our tool derives the malicious behaviors *automatically*, and does not need the help of the user for this.

2 BACKGROUND

Given a training set of known malwares and benign programs, STAMAD can (1) either *automatically* extract malicious behaviors using Information Retrieval techniques, or (2) apply machine learning techniques to *automatically* learn malwares. Then, in both cases, STAMAD can classify a new given unseen program as malicious or benign. We give in this section the main ideas behind these two approaches that are implemented in STAMAD.

2.1 Extraction of Malicious Behaviors

STAMAD implements the techniques described in [11, 12]: Given a set of extended API call graphs that correspond to malwares and a set of extended API call graphs corresponding to benign programs, we want to extract in a completely automatic way a malicious extended API graph that corresponds to the malicious behaviors of the malwares. This malicious extended API graph should represent the parts of the extended API call graphs of the malwares that correspond to the malicious behaviors. The best subgraphs that should be extracted are those able to distinguish the malicious extended API call graphs from the benign ones. Thus, our goal is to isolate the few relevant subgraphs from the nonrelevant ones. This problem can be seen as an Information Retrieval problem, where the goal is to retrieve relevant items and reject nonrelevant ones. The Information Retrieval community has been working on this problem for a long time. Over the past three decades, it has accumulated a large amount of experience on how to efficiently retrieve information. Thus, it would be interesting to adapt the knowledge and experience of the Information Retrieval (IR) community to our malicious behavior extraction problem. One of the most popular techniques that was shown to be very efficient in the IR community is the TFIDF scheme that computes the relevance of each item in the collection using the TFIDF weight that is computed from the occurrences of terms in a document and their appearances in other documents. We showed in [11, 12] how to adapt this technique that was mainly applied for text and image retrieval for malicious extended API graph extraction. For that, we associate to each node and each edge in the extended API call graphs of the programs of the collection a weight. Higher weight implies higher relevance. Then, we compute the malicious extended API graphs by taking edges and nodes that have the highest weights. Then, we use our automatically extracted malicious behavior specification for malware detection using a kind of product between graphs.

More precisely, let $\mathcal{G}$ be a set of extended API call graphs. Let i be a term (either a node, i.e., an API function and its parameter evaluation, or an edge) and j be an extended API call graph. Then, the weight of i in j is given by:

$$w_g(i,j) = F\big(\mathbf{tf}(i,j)\big) \times \mathbf{idf}(i) \tag{1}$$

here $\mathbf{tf}(i,j)$ is the number of occurrences of the term i in the graph j, i.e., term frequency, and $\mathbf{idf}(i) = \log(\frac{N}{\mathbf{df}(i)})$, is the inverse document frequency, where N is the total number of graphs and $\mathbf{df}(i)$ is the number of graphs containing the term i. The **idf** factor ensures that a common term (an API function node or an edge) which appears in a lot of graphs is not relevant (for example the API functions strlen, strcpy, strcat, etc. appear frequently in all the extended API call graphs but are not concerned with malicious behaviors).

F is a function that applies the optimizations that are made in the IR community in order to normalize graph sizes and to ensure that high **tf** for a relevant term in an extended API call graph does not place that graph ahead of other graphs which have multiple relevant terms but with lower **tf** values.

Then, the relevance of a term i in $\mathcal{G}$ is measured by its relevance in each graph in this set:

$$W(i,\mathcal{G}) = \frac{1}{K}\sum_{j=1}^{|\mathcal{G}|} w_g(i,j) \tag{2}$$

where K is a normalising coefficient.
Then, the malicious extended API call graph is computed by considering the nodes and edges which are highly relevant (have a high relevance) for the set $\mathcal{M}$ of malwares and not highly relevant (have a low relevance) for the set $\mathcal{B}$ of benwares . More details about our approach can be found in [11, 12].

2.2 Learning Malicious behaviors

In our second approach, we apply machine learning techniques on extended API call graphs to learn malicious behaviors, and detect malwares. Support Vector Machine (SVM) [8, 19] is one of the most successful techniques in machine learning. It has been applied to

several fields in pattern recognition including text analysis and bioinformatics. In STAMAD, we apply Support Vector Machine based learning techniques for malware detection. The choice of Support Vector Machine is motivated by the fact that they are very suitable for nonvectorial data (graphs in our setting), whereas the other well-known learning techniques like artificial neural network, k-nearest neighbor, decision trees, etc. can only be applied to vectorial data. This SVM method is highly dependent on the choice of kernels. A kernel is a function which returns similarity between data [22]. In STAMAD, we use a variant of the random walk graph kernel that measures graph similarity as the number of common paths of increasing lengths [21]. More details about our learning approach can be found in [10, 13].

3 STAMAD DESCRIPTION

STAMAD is a static PC malware detector. It takes as input a set of malwares and a set of benwares and can either (1) extract a malicious API graph representing the malicious behaviors of the malwares in the set (using the techniques described in Section 2.1); or (2) learn to classify malwares without extracting the malicious behaviors (using the techniques described in Section 2.2). These phases are called the training phases. Then, given a new program, STAMAD checks whether it is malicious or not. STAMAD consists of the following modules:

Module 1: Extended API Call Graph Computation

This module extracts an extended API call graph from a binary code. It relies on Jakstab [17] and IDA Pro [14]: Jakstab performs static analysis of the binary program and provides its corresponding assembly program and control flow graph. However, it does not allow to extract information of API functions and indirect calls to API functions in the program. Hence, IDA Pro is used to get these informations of API functions with an assembly code.

First, this module takes as input a binary code and uses PEfile[1] to check whether the binary code is packed or not. If so, this code is unpacked by the corresponding unpacker, e.g., UPX[2]. Then, it is fed to Jakstab [17] and IDA Pro [14]. Otherwise, this binary code is directly passed to Jakstab [17] and IDA Pro [14]. To be able to compute the edge annotations that describe the relations between the API function arguments, we compute a pushdown system (PDS) from the output of Jakstab and IDA Pro. Indeed, PDSs allow to keep track of the program's stack, and thus, of API function arguments since in assembly, parameter passing is done via the stack. The algorithm that we use to derive a PDS from a binary code is described in [11]. Then, we apply on this PDS the static analysis saturation procedure of [11] that allows to derive the relations between the API function's parameters, and hence, to compute the extended API call graph as described in [11]. We use Moped [3] to compute the reachable configurations of PDSs. We specify the parameters for each API function using the MSDN library [4] which gives the description of API functions. The structure of this module is described in Figure 2.

[1]https://github.com/erocarrera/pefile
[2]https://upx.github.io/

Module 2 : Extraction of Malicious Behaviors.

This module consists of two phases: the extraction of malicious behaviors phase and the malicious behavior detection phase. In the extraction phase, it takes as input a set of malwares and a set of benwares. After applying the Extended API Call Graph Module, to extract their corresponding extended API call graphs, these graphs are fed to the Malicious Graph Computation component to compute the malicious extended API graph. This component implements the TFIDF weighting term scheme to compute the malicious behaviors as described in [11, 12]. It outputs a malicious API graph representing the malicious behaviors. This phase will be called "training phase".
In the detection phase, this module takes as input a new binary program and applies the Extended API Call Graph Module to extract its corresponding extended API call graph. Then, it checks whether this graph contains any malicious behavior from the malicious graph (the output of the "training phase") or not by applying a kind of product between the two graphs. If this graph contains any malicious behavior, the output is "Malicious!". Otherwise, the output is "Benign!". The structure of this module is described in Figure 3 .

Module 3: Learning Malicious Behaviors.

This module implements the learning techniques as described in Section 2.2. It consists of two phases: the learning phase and the detection phase. In the learning phase, it takes as input a set of malwares and a set of benwares. It first applies the Extended API Call Graph Module to compute their corresponding extended API call graphs. Then, these extended API call graphs are fed to the SVM training component, i.e., LIBSVM [9], to compute a SVM training model.
In the detection phase, this module takes as input a binary code, applies the Extended API Call Graph Module to compute its corresponding extended API call graph. Then, it uses the LIBSVM classifier with the training model (the output of the first phase) to classify the program either "Malicious!" or "Benign!". The structure of this module is described in Figure 4.

4 EXPERIMENTS

To evaluate our tool, we use a dataset of 2249 benign programs and 4035 malwares collected from Vx Heaven [1] and from VirusShare [5].
Extraction of malicious behaviors. To evaluate the performance of Module 2, we first applied our tool to automatically extract an extended malicious API graph from a set of 2124 malwares and 1009 benign programs. The obtained extended malicious API graph is then used for malware detection on a test set of 1911 malwares and 1240 benign programs. We obtained encouraging results: a detection rate of 95.66% with 0 false alarms.
Learning malware. To evaluate the performance of module 3, we randomly split the dataset into two partitions, a training and a testing partition. For the training partition, the quantity of malwares and benign programs is balanced with 1009 samples for each, this will allow us to compute the SVM classifier. The test set consisting of 3026 malwares and 1240 benign programs, is used to evaluate the classifier. Using the training set, we compute the training model.

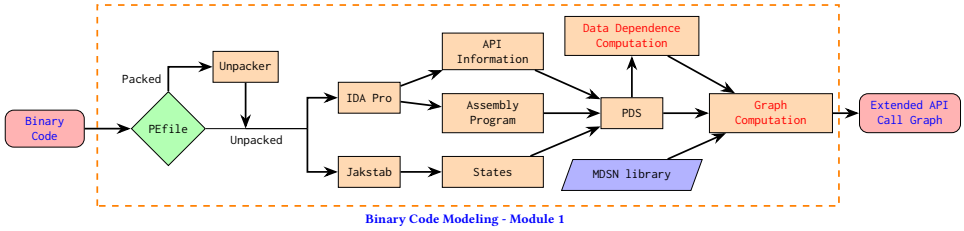


Figure 2: Extended API Call Graph Computation - Module 1.

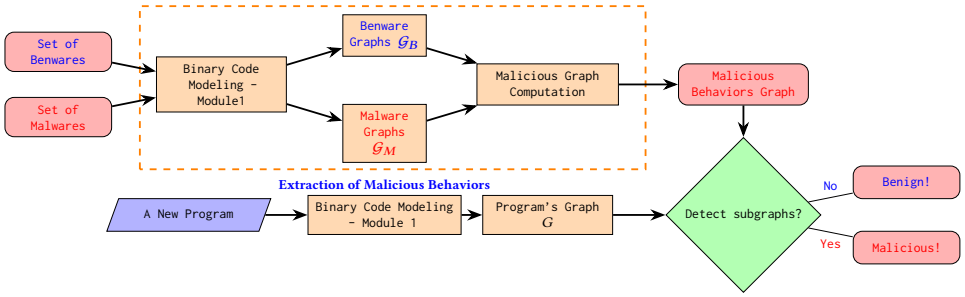


Figure 3: Extraction of Malicious Behaviors - Module 2.

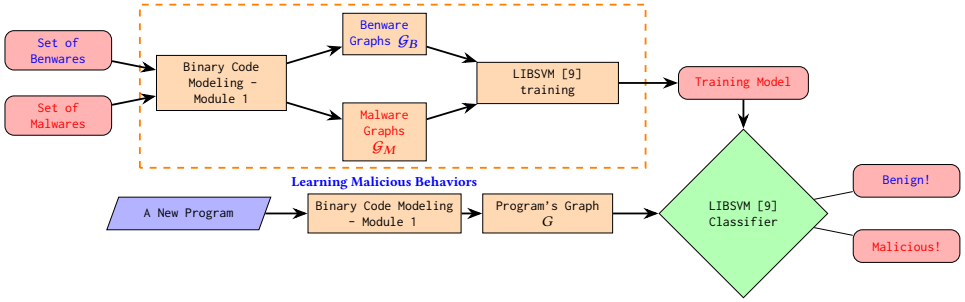


Figure 4: Learning Malicious Behaviors - Module 3.

Then, we apply this training model to classify malwares on the test set and obtain a detection rate of 96.73%, with 0.73% of false alarms. **Comparison with other antiviruses.** We compare the performance of our tool with different existing antiviruses including Avira, Kaspersky, Avast, Qihoo-360, McAfee, AVG, BitDefender, ESET-NOD32, F-Secure, Symantec and Panda. Since known antiviruses update their signature database as soon as a new malware is known, in order to have a fair comparision with these antiviruses, we need to consider new malwares. For this, we use three generators to create new malwares: NGVCK, RCWG and VCL32. The latter are able to create sophisticated malwares with morphing code and other features to avoid being detected by antiviruses. In total, we generate 180 new malwares by RCWG, VCL32 and NGVCK generators. As described in Table 1, using our tool, we are able to detect 100% of these new malwares while none of the well-known antiviruses can detect all of them [12, 13] .

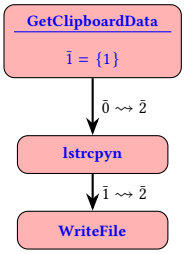
Antivirus	Detection Rates	Antivirus	Detection Rates
Module 2	**100%**		
Module 3	**100%**	Panda	19%
Avira	16%	Kaspersky	81%
Avast	87%	Qihoo-360	96%
McAfee	96%	AVG	82%
BitDefender	87%	ESET-NOD32	87%
F-Secure	87%	Symantec	14%

Table 1: This table shows a comparison of our method against well-known antiviruses. Our tool achieves a detection rate of 100%.

5 EXAMPLES OF MALICIOUS BEHAVIORS

In this section, we present some the malicious behaviors that are extracted by STAMAD.

Stealing clipboard data. This malicious behavior gets the text data from the current clipboard and stores them into a file. It is shown in the graph of Figure 5. The graph describes the following

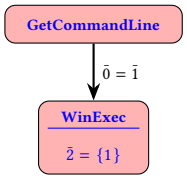


n_1 : *push* 1
n_2 : *call* **GetClipboardData**
n_3 : *mov ebp, eax*
n_4 : *push ebp*
n_5 : *push lpString*
n_6 : *call* **lstrcpyn**
....
n_7 : *push lpString*
....
n_8 : *call* **WriteFile**

Figure 5: The automatically extracted malicious graph of Stealing clipboard data malicious behavior, and one of its implementation in assembly code.

malicious behavior: The malicious program first calls the API function GetClipboardData with 1 as first parameter ($\bar{1}$) to retrieve the text data from the clipboard. Then, it copies this data to a string by calling the API function lstrcpyn with the output parameter ($\bar{0}$) of the function GetClipboardData as second parameter ($\bar{2}$). Finally, it writes this string into a file by calling the API function WriteFile with the first parameter ($\bar{1}$) of the previous function (lstrcpyn) as second parameter ($\bar{2}$). This behavior is implemented by the assembly code on the right-hand side. In this code, the first parameter of GetClipboardData is pushed into the stack at n_1. The output of GetClipboardData is stored in *eax*, then it is pushed into the stack at n_4 as the second parameter of the function lstrcpyn. The output of this function is stored in its first parameter (*lpString*) which is pushed into the stack at n_5. Finally, this output is pushed into the stack at n_7 as the second parameter of WriteFile.

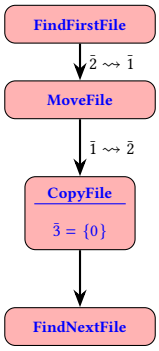
Executing the program and keeping it alive. This malicious behavior executes the current program whenever it is terminated. It is shown in the graph of Figure 6. The graph describes the following malicious behavior: The malicious program first calls the API function GetCommandLine to get the executing command of the current process. Then, it calls the API function WinExec with the output of the previous function call as first parameter and 1 as second parameter to run this command in a new process. This behavior is implemented by the assembly code on the right-hand side.



n_1 : *call* **GetCommandLine**
n_2 : *push* 1
n_3 : *push eax*
n_4 : *call* **WinExec**

Figure 6: Executing the program and keeping it alive. The automatically extracted malicious extended API call graph on the left-hand side and its implementation on the right-hand side.

Infecting files in the system. This malicious behavior searches for executable files in the system and replaces them by the malicious program. It is shown in the graph of Figure 7. The graph describes the following malicious behavior: The malicious program first calls the pair of API functions FindFirstFile at the start node and FindNextFile at the final node to search executables, i. e., ".exe", in a specific folder. If it finds one executable in this folder, it changes the name of this executable by calling the API function MoveFile with the second parameter of FindFirstFile as first parameter and replaces this executable with itself by calling the API function CopyFile with the first parameter of MoveFile as second parameter and 0 as third parameter. This behavior is implemented by the assembly code on the right-hand side.



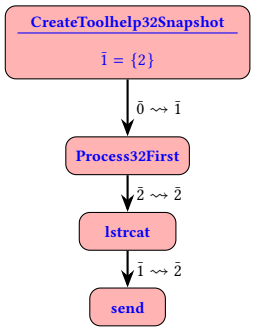
n_1 : *push FindFileData*
n_2 : *push ".exe"*
n_3 : *call* **FindFirstFile**
n_4 : *push eax*
n_5 : *mov edi, FindFileData*
n_6 : *lea eax, [edi + 2Ch]*
n_7 : *push NewFileName*
n_8 : *push eax*
n_9 : *call* **MoveFile**
n_{10} : *push* 0
n_{11} : *lea eax, [edi + 2Ch]*
n_{12} : *push eax*
n_{13} : *mov eax, ExistingFile*
n_{14} : *push eax*
n_{15} : *call* **CopyFile**
n_{16} : *pop eax*
n_{17} : *push edi*
n_{18} : *push eax*
n_{19} : *call* **FindNextFile**

Figure 7: Infecting files in the system. The automatically extracted malicious extended API call graph on the right-hand side and its implementation on the left-hand side.

Getting the information of the running processes. This behavior captures information of the running processes in the system and sends these data via the network. It is shown in the graph of Figure 8. The graph describes the following malicious behavior: The malicious program first calls the API function CreateToolhelp32Snapshot with 2 as first parameter to capture all processes running in the system. Then, the API function Process32First with the output of the previous call as first parameter is called to retrieve information of these processes. It captures the location of the executable of each process in a text buffer by calling the API function lstrcat with the second parameter of Process32First as second parameter. Finally, it sends this buffer via the network by calling the API function send with the first parameter of lstrcat as second parameter. Its implementation is shown on the right-hand side.

REFERENCES

[1] 2014. Vx Heaven Computer Virus Collection. http://vxheaven.org. Accessed: 2014-11-25.
[2] 2017. bnxnet. http://www.bnxnet.com/. Accessed: 2017-02-25.
[3] 2017. Moped. http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/. Accessed: 2017-08-25.
[4] 2017. MSDN Library. https://msdn.microsoft.com/en-us/library/. Accessed: 2017-08-25.



n_1 : *push* 0
n_2 : *push* 2
n_3 : *call* **CreateToolhelp32Snapshot**
n_4 : *mov snapshot*, *eax*
n_5 : *push pe*
n_6 : *push snapshot*
n_7 : *call* **Process32First**
n_8 : *push pe.szExeFile*
n_9 : *push lpString*
n_{10} : *call* **lstrcat**
...
n_{11} : *push lpString*
...
n_{12} : *call* **send**

Figure 8: Getting the information of the running processes. The automatically extracted malicious extended API call graph on the left-hand side and its implementation on the right-hand side.

[5] 2017. VirusShare CryptoRansom 20160715. https://virusshare.com. Accessed: 2017-2-25.

[6] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Detection: Countering the Largest Security Threat*, Wenke Lee, Cliff Wang, and David Dagon (Eds.). Springer US, Boston, MA, 65–88. https://doi.org/10.1007/978-0-387-68768-1_4

[7] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment*, Roland Büschkes and Pavel Laskov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 129–143.

[8] Christopher J.C. Burges. 1998. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery* 2, 2 (01 Jun 1998), 121–167. https://doi.org/10.1023/A:1009715923555

[9] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011). Issue 3. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[10] Khanh-Huu-The Dam and Tayssir Touili. 2018. Learning Malware Using Generalized Graph Kernels. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*. 28:1–28:6. https://doi.org/10.1145/3230833.3230840

[11] Khanh-Huu-The Dam and Tayssir Touili. 2018. Precise Extraction of Malicious Behaviors. In *2018 IEEE 42nd Annual Computer Software and Applications Conference, COMPSAC 2018, Tokyo, Japan, 23-27 July 2018, Volume 1*. 229–234. https://doi.org/10.1109/COMPSAC.2018.00036

[12] Khanh Huu The Dam and Tayssir Touili. 2016. Automatic extraction of malicious behaviors. In *Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 1–10.

[13] Khanh Huu The Dam and Tayssir Touili. 2017. Malware Detection based on Graph Classification. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - Volume 1: ICISSP,*. INSTICC, ScitePress, 455–463. https://doi.org/10.5220/0006209504550463

[14] Chris Eagle. 2011. *The IDA Pro Book* (2nd ed.). No Starch Press.

[15] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic Spyware Analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, USA, Article 18, 14 pages. http://dl.acm.org/citation.cfm?id=1364385.1364403

[16] Joris Kinable and Orestis Kostakis. 2011. Malware classification based on call graph clustering. *Journal in Computer Virology* 7, 4 (01 Nov 2011), 233–245. https://doi.org/10.1007/s11416-011-0151-y

[17] Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Vol. 5123.

[18] Deguang Kong and Guanhua Yan. 2013. Discriminant Malware Distance Learning on Structural Information for Automated Malware Classification. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '13)*. ACM, New York, NY, USA, 1357–1365. https://doi.org/10.1145/2487575.2488219

[19] Hichem Sahbi and Xi Li. 2011. Context-Based Support Vector Machines for Interconnected Image Annotation. In *Computer Vision – ACCV 2010*, Ron Kimmel, Reinhard Klette, and Akihiro Sugimoto (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 214–227.

[20] Fu Song and Tayssir Touili. 2013. PoMMaDe: pushdown model-checking for malware detection. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. ACM, 607–610.

[21] L. Wang and H. Sahbi. 2013. Directed Acyclic Graph Kernels for Action Recognition. In *2013 IEEE International Conference on Computer Vision*. 3168–3175. https://doi.org/10.1109/ICCV.2013.393

[22] L. Wang and H. Sahbi. 2014. Bags-of-daglets for action recognition. In *2014 IEEE International Conference on Image Processing (ICIP)*. 1550–1554. https://doi.org/10.1109/ICIP.2014.7025310

[23] Ming Xu, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren, and Ning Zheng. 2013. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques* 9, 1 (2013), 35–47. https://doi.org/10.1007/s11416-012-0175-y