



HAL
open science

BAGUETTE: Hunting for Evidence of Malicious Behavior in Dynamic Analysis Reports

Vincent Raulin, Pierre-François Gimenez, Yufei Han, Valérie Viet Triem Tong

► **To cite this version:**

Vincent Raulin, Pierre-François Gimenez, Yufei Han, Valérie Viet Triem Tong. BAGUETTE: Hunting for Evidence of Malicious Behavior in Dynamic Analysis Reports. SECRIPT 2023 - 20th International conference on security and cryptography, Jul 2023, Rome, Italy. pp.1-8. hal-04102144

HAL Id: hal-04102144

<https://hal.science/hal-04102144v1>

Submitted on 16 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

BAGUETTE: Hunting for Evidence of Malicious Behavior in Dynamic Analysis Reports

Vincent Raulin¹, Pierre-François Gimenez², Yufei Han¹ and Valérie Viet Triem Tong²

¹*Inria, Univ. Rennes, IRISA, Rennes, France*

²*CentraleSupélec, Univ. Rennes, IRISA, Rennes, France*

{vincent.raulin, yufei.han}@inria.fr; {pierre-francois.gimenez, valerie.vietriemtong}@centralesupelec.fr

Keywords: Malware Analysis, Visualization, Dynamic Analysis

Abstract: Malware analysis consists of studying a sample of suspicious code to understand it and producing a representation or explanation of this code that can be used by a human expert or a clustering/classification/detection tool. The analysis can be static (only the code is studied) or dynamic (only the interaction between the code and its host during one or more executions is studied). The quality of the interpretation of a code and its later detection depends on the quality of the information contained in this representation. To date, many analyses produce voluminous reports that are difficult to handle quickly. In this article, we present BAGUETTE, a graph-based representation of the interactions of a sample and the resources offered by the host system during one execution. We explain how BAGUETTE helps automatically search for specific behaviors in a malware database and how it efficiently assists the expert in analyzing samples.

1 INTRODUCTION


Malware infection is a major cybersecurity threat that evolves persistently and rapidly. The perpetrators progressively refine their malware payloads and related extortion activities. More than 120 million new malware samples are uncovered every year. According to the report from Cybersecurity Ventures¹, Ransomware will cost its victims around 265 billion USD annually by 2031. Therefore, developing and updating malware detection techniques to mitigate malware-associated cyber-threats becomes necessary.


Security researchers have been working on malware detection for decades. The mainstream malware analysis solutions can be categorized as static analysis and dynamic analysis. Static analysis methods scan the raw bytes of the software to extract signatures, such as printable strings, n-grams, and instructions. However, the signature-based static analysis might be vulnerable to code obfuscation. In contrast, dynamic analysis executes each software in an isolated environment (e.g., a sandbox) to collect its run-time behaviour information. By using such be-


havioral information, dynamic analysis can produce a higher detection rate and is more robust to evading techniques such as packed and obfuscated codes. Our study focuses on representing and visualizing the behaviour information about malware samples extracted from Cuckoo report. **We aim to make this information easy to analyze and manipulate by the human expert and to produce easy-to-interpret results to identify suspicious payloads.**

In previous works, dynamic traces are usually organised as reports stacking all run-time information, including file access, network activities, registry operations, and process/thread information. As informative as they are, they can't help automatically identify the malicious payloads, i.e., facilitating machine learning-driven methodologies to detect the existence of malicious functions in malware samples. Furthermore, it is difficult for human experts to understand how malware samples compromise the target system by simply reading the malware analysis reports. Therefore, we propose a graph-based post-processing analysis report framework, named **Behavioral Analysis Graph Using Execution Traces Towards Explainability** (BAGUETTE). We describe how the BAGUETTE graph-based representation is designed and we demonstrate in the rest of the article how it can benefit the downstream analysis.

The main contributions of this article are:

^a <https://orcid.org/0000-0002-7125-9947>

^b <https://orcid.org/0000-0002-4238-4423>

^c <https://orcid.org/0000-0003-4838-2952>

¹<https://tinyurl.com/3ck4uyfr>

- We define the heterogeneous graph used by BAGUETTE that summarizes the interactions between the malware and the network, the file system activity, the registry keys and more;
- We demonstrate the usefulness of such graphs with the automatic search of behavioral subgraphs on Windows Malware;
- We detail a study of a Windows ransomware, LockBit.

2 RELATED WORKS

Most works on dynamic analysis (Sihwail et al., 2018) (Raff and Nicholas, 2020) collect system call or API call traces because they are the medium of interaction between a program and the host OS. Any malicious action the malware may take over the resources of the target system involves API calls in the captured traces (Kolosnjaji et al., 2016)(Wunderlich et al., 2020). While some work only makes use of the name of the API calls, their parameters can also be important to identify suspicious behaviours (Ding et al., 2018)(Park et al., 2010)(Xiao et al., 2019). Monitoring disk access, network traffic, and OS activity inside a virtual machine have also been proposed in (Alptekin et al., 2019)(Wang et al., 2019). Finally, physical properties like CPU usage (Piplai et al., 2020) or CPU power consumption (Bridges et al., 2018) can also be monitored.

To facilitate human analysts and Machine Learning-driven malware classification, execution traces produced by dynamic analysis needs to be re-organised into well-formatted feature representation of malware behaviors. We summarize previous works on defining malware feature representations from the following three perspectives.

2.1 Sequences of function calls to record malware behaviours

A classical representation directly uses time series of system calls without parameters (Kolosnjaji et al., 2016)(Wunderlich et al., 2020). N-grams (sequences of n system calls) or word embedding vectors (a machine learning technique) are then extracted to conduct malware analysis. A graph structure similar to Markov chains has also been proposed (Chen et al., 2017), where each vertex denotes a system call and each edge is labelled by the probability to chain two system calls in the observed executing traces. This representation was extended by (Grimmer et al., 2018) to consider n -tuples of system calls. The rep-

resentation has a main advantage: its size does not depend on the sample or the size of the execution traces, but only on the number of system calls being monitored. Another approach is proposed by (Qiang et al., 2022) that reconstructs the program’s control flow graph from execution traces.

The limit of the sequential representation is they cannot produce a semantic understanding of malware behaviours. The temporal sequence of the function calls cannot show much beyond the execution order of different API/system functions and it does not detail the system resources accessed, created, or modified by these functions. This prevents further interpretation of malware functions and makes malware analysis vulnerable to mimicry attacks, where adding/removing a few function calls evades detection.

Besides the name of the function calls, the parameters of these API/system calls allow human analysts to characterize the behavior of a malware. For example, it can be used to identify whether an "open file" system call and a "write into file" system call concern the same file. Several representations track objects, such as files, sockets, or threads manipulated by system calls from parameters or return values. For example, (Ding et al., 2018) defines a graph where each vertex is a system call, and a directed edge links every pair of consecutive system calls that manipulate the same object in the sequence. Two other works, (Xiao et al., 2019) and (Park et al., 2010), consist of multiple time series, one for each set of system calls manipulating the same object.

While these representations use API call parameters to track actions on objects, they do not allow easy analysis of the interactions between system objects. Our heterogeneous representation includes the effects of system calls with their parameters on the system, creating semantics links between the different objects.

2.2 Interactions with the hosting system

To better analyze malware behaviors, several works do not only rely on system calls but also on multiple other sources of information about the system, such as network traffic, contacted IPs, URLs, or domains, and file accesses. In (Alptekin et al., 2019), processes, threads, files and sockets are included in a graph-based representation. The manipulated objects are higher-level concepts: for example, file objects (which have persistence in the system) replace file descriptors (which are only valid for the current process). Sadly, there is little information on how they build this graph.

These works build representations for a single purpose: for example, (Piplai et al., 2020) is a data min-

ing tool and (Kolosnjaji et al., 2016) is meant to be used with machine learning algorithms. Our work aims to create a well-defined representation that can be used for multiple purposes, from machine learning to expert analysis. We demonstrate this ability with two experiments: automatic graph analysis and malware visualization.

2.3 Interactions between processes of different applications

The work from (Wang et al., 2019) proposes a graph-structured representation to monitor multiple processes on several machines. This is a heterogeneous graph with three node types: file, process and socket. There are three kinds of edges: from a process to a socket (if that process uses this socket to communicate), from a process to a file (if that process interacts with it) and from a process to another one (if the former creates the latter). This malware representation can identify a suspicious process by comparing its behavior with benign processes. It can also be used to analyze the behavior of a malware with respect to the files it handles, the processes it spawns and the IP addresses it contacts. For example, it could help identify a remote Command and Control (C&C) server. However, this method requires a more complex experimental setup to build than a classical host-scale monitoring. Our work focuses on the impact of the malware on the system and, in this regard, is more complete than (Wang et al., 2019)’s work. Indeed, on most systems, three types of entities is not enough to describe the behavior of a malware sample. In our representation, more types of entities are present, and we define more subtle interactions between them. Our representation is restricted to the direct perimeter of action of the malware : we track all the actions of the malware on the host system, as well as what goes in and out of the machine.

Finally, VirusTotal’s database exploration graph² is a representation that entails the interactions between a malware and other entities (file, IP, domains, etc.), both via internet and via the file system. It is a very high-level representation that does not contain, for example, the network data transmitted or the interactions between file access.

From the previous works, it appears that a higher-level, more detailed and semantically richer representation of malware behavior is necessary in the community of malware analysis. To answer this problem, we present here BAGUETTE a framework based on heterogeneous graphs that attempts to capture as

much as possible the effect of the malware on the system while staying readable by both ML-based approaches and human analysts.

3 BAGUETTE GRAPHS FOR AUTOMATIC EXPLORATION

We propose to use the BAGUETTE graph to represent the behavioral information extracted during a dynamic analysis conducted with Cuckoo over 2 minutes (limit of resources). A BAGUETTE graph shows the actions that malware samples took over system resources, such as files and the network traffics. It enables human analysts to flag and understand the malicious payloads of malware samples.

3.1 Definition of BAGUETTE graphs

A BAGUETTE graph is defined as a heterogeneous graph containing 14 types of nodes: *Host*, *Process*, *Thread*, *API Call*, *Import*, *File*, *Directory*, *Handle*, *Connection*, *Socket*, *Diff*, *Registry Key*, *Registry Entry* and *Registry Handle*. Each of these types of nodes has its own attributes. Edges and arrows are typed as well (e.g., "Has Child Process", "References File System", "Has Similar Content"... 59 types in total). The definition of the node attributes and edges types in a BAGUETTE graph is presented in Figure 1.

The root node of a BAGUETTE graph is a *Host* node. It represents a physical machine that starts processes, connects to other machines and owns hard drives to store the file system. In our study, one *Host* node will always denote the virtual machine (VM) executing dynamic analysis. Other hosts may appear in the graph if they are connected to the VM by network connections.

We introduce multiple categories of types which group vertices, edges and arrow types by the behavior they represent. For example, *Process*, *Thread* and *Call* nodes are related to the **execution** of programs. They will be organized to represent the execution tree (process and thread hierarchy, which each *Thread* node spanning the sequence of *API Call* nodes that it made).

File system activity is represented by *File*, *Directory* and *Handle* nodes. In a BAGUETTE graph, they will form the part of the file system tree that was used by the sample during the execution, with its root drives connected to the host machine.

Network activity is represented by *Socket*, *Connection* and *Host* nodes. Multiple *Host* nodes might be linked by *Connection* nodes, and their even-

²The Virus Total website: <https://www.virustotal.com/>

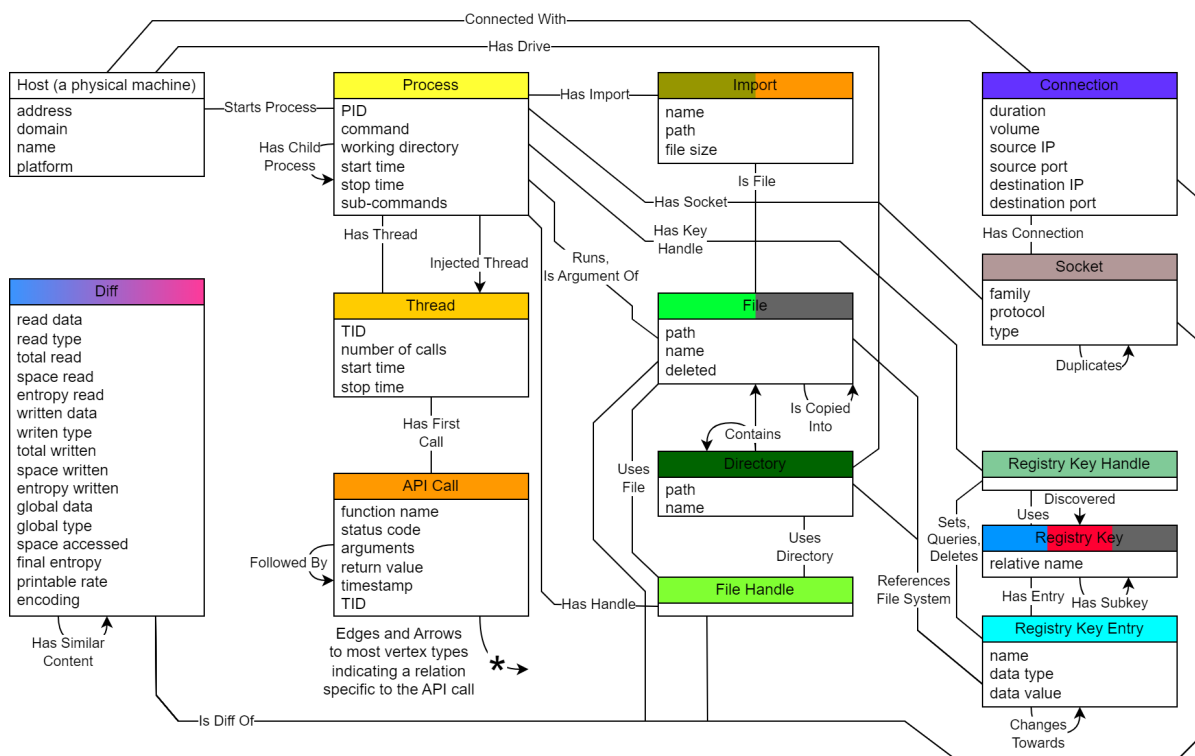


Figure 1: Structure of BAGUETTE graph

tual conversation might be accessible through *Socket* nodes.

Windows **registry** activity is represented by *Key*, *Key Entry* and *Key Handle* nodes. Similarly to the file system, the registry tree used by the sample will be present in the BAGUETTE graphs, featuring all operations made on the keys and entries.

Diff nodes trace the **data** flows across the execution. They represent data that was captured through any data-capable object and are compared to each other using a similarity metric.

Finally, *Import* nodes correspond to DLL *imports*.

All of these categories also have edges and arrow types, which relate the different types of vertices (potentially from different categories).

3.2 BAGUETTE graph construction

A BAGUETTE graph is built in 3 steps.

Step 1: Host, Process, Thread and API call nodes The host node of the VM is first created. Then, one node is created for each process, thread and API calls, and they are linked with the according edge and arrow types according to the BAGUETTE scheme defined in Figure 1.

Step 2: File, File Handle, Socket, Connection, Registry Key, Registry Key Handle, Registry Key

Entry, and Import nodes In this step, each API call is processed chronologically. For example, each time a file handle is used by an API call, we search the corresponding *Handle* nodes (it is created if it does not exist) and we connect the *API Call* nodes to the *File Handle* nodes. *Registry Key*, *Registry Key Entry*, *Registry Key Handle*, *Sockets* and *Imports* nodes are created and connected to *API call* with the adequate edge type 1 similarly. Besides multiple special case handling happen. For example, edges "Injected Thread" can be added between a process and a foreign thread if a `CreateRemoteThread` call (or similar) is encountered or when an API call references directly a file path without using a file handle (e.g., `CopyFile`), a *File* node is created.

Step 3: Diff and Directory nodes and other refining For each *File* node, *Directory* nodes are created in a tree structure to place the file in their relative positions in the file system. Besides, the *Diff* nodes are compared to each other using a similarity metric or might be merged if they have identical data. If the name of a *File* or a *Directory* is mentioned in a *Registry Key Entry*'s value, an edge labeled "References File System" is added.

BAGUETTE is available here³.

³<https://pypi.org/project/baguette-verse/0.9/>

4 IDENTIFYING BEHAVIORS WITH METAGRAPHS

A limitation of dynamic analysis is that malware samples may not execute their malicious payloads during the analysis period. There could be multiple causes, e.g. dead C&C server, incompatibility crash, waiting for a specific date, or evasion techniques (VM environment or Cuckoo agent detection, sleep for the duration of the analysis, etc.). For this reason, for a given malware family, we run many of its variants. As a result, our dynamic analysis produces a large set of malware execution traces. Only a fraction of them correspond to observable malicious payloads.

To facilitate tagging the potentially interesting traces, we propose to search for subgraphs of the BAGUETTE graphs that correspond to particular behaviors probably related to malicious actions. These graph patterns denoting specific behaviours and shared in different BAGUETTE subgraphs are typically referred to as *metagraphs* (Fang et al., 2021). A metagraph is a type graph that complies with the BAGUETTE scheme (cf. Fig. 1), which can also have partially defined attributes (i.e., associated with a range of values). We define a match of a metagraph pattern in a BAGUETTE graph as finding a subgraph with the same structure and compatible node attributes as the metagraph pattern. Subgraph matching is an NP-complete problem. We use some heuristic rules to narrow down the graph exploration. In particular, we prioritize the search by starting with nodes for which the types are the rarest. Note that we search for all possibly matching subgraphs in a BAGUETTE graph given one specific metagraph pattern.

4.1 Metagraph examples

We propose 5 metagraphs. Each one shows a particular behavior that could exist in our BAGUETTE graphs, which correspond to suspiciously malicious payloads.

The (a) **”High-Entropy Writing”** metagraph represents the writing of high entropy bytes to a file. It likely belongs to the payloads of data encryption. This behaviour is encoded as a *File* node connected with a *Diff* node, where the *Diff* node writes high-entropy contents into the corresponding file.

The (b) **”Changed File Type”** metagraph denotes the action of reading a file’s contents and changing the file type associated with the file content. This behavior pattern is encoded by a *File* node connected to a *Diff* node with the type of the read data different from the type of the written data. This behavior can be an

indicator to masquerading⁴.

The (c) **”Covert Execution”** metagraph represents the behavior of executing a file that is not supposed to be executable. This is encoded by searching for a *Process* node that executes a file with an extension beyond typical Windows executable file types, including *”.exe”* (PE Executable), *”.vbs”* (Visual Basic Script) and *”.ps1”* (Powershell script) . This technique can also be used for masquerading.

The (d) **”Extraction and Execution”** metagraph represents a process writing to a file first, and then executing it in a child process. This is encoded as two *Process* nodes. One process is the parent of the other. The parent process uses a handle on the executed file. There is also a *Diff* node connected to the *Handle* node. The *Diff* node indicates only writing operations are performed on the file via this *Process*’s *Handle*. Finally the same file is executed by the child process.

The (e) **”Auto-Run”** metagraph represents the behavior that a program changes/creates a key value in the run keys, in order to launch automatically the specified applications during startup of the system or user login (Auto-Run/Logon-Run). The added key value refers to a file that will be executed or used as a parameter in a new process created at the next system startup. To match the exact keys, the metagraph must contain the complete *Key* hierarchy (constrained by their *”name”* attributes). At the end, there is a modified *Key Entry* linked to the *Key Handle* used to set the *Key Entry* (the type of the link is *Set Entry*) and the file referred by the value of the entry.

4.2 Experiment

We created a dataset of BAGUETTE graphs defined using the Cuckoo reports of malware samples of three families collected from MalwareBazaar: 247 samples of GCleaner, 436 samples of SnakeKeyLogger and 7 samples of LockBit. GCleaner⁵ is a file dropper, SnakeKeyLogger⁶ is a key logger and spyware, and LockBit⁷ is a ransomware. Their BAGUETTE graphs are available in our repository⁸ and can be visualized using Gephi⁹. We searched for metagraphs inside these BAGUETTE graphs and we counted the number of matches for each metagraph. The results are presented in Table 1. *p* represents the fraction

⁴<https://attack.mitre.org/techniques/T1036/>

⁵<https://bazaar.abuse.ch/browse/signature/GCleaner/>

⁶<https://bazaar.abuse.ch/browse/signature/SnakeKeylogger/>

⁷<https://bazaar.abuse.ch/browse/signature/LockBit/>

⁸https://files.inria.fr/baguette/SECRYPT_2023.zip

⁹<https://gephi.org/>

of samples belonging to one family, which contain at least 1 match to the given metagraph pattern. n (resp. σ) is the average number (resp. standard deviation) of matches per sample with at least one match.

From the results, we can find that every metagraph pattern can be matched with at least one BAGUETTE graph. It implies that these patterns may indicate important payloads in these malware samples. Furthermore, the distribution of metagraph patterns varies significantly across malware families. This shows that different metagraph patterns can be considered as behavioral signatures of malware family.

In the columns of *GCleaner*, we can see that all metagraph patterns except "Auto-Run" can find the matched subgraphs. Furthermore, the very low variance of the number of matches per sample tell us that we identified characteristic behaviors of this family. For example, each of the *GCleaner* samples contain on average 3 matches of the "Extraction and Execution" metagraph pattern.

For *SnakeKeyLogger*, there are much fewer matches. Indeed, 326 out of the 436 *SnakeKeyLogger* samples are not executed properly and their BAGUETTE graphs are almost empty. These samples may deploy evading techniques to avoid sandbox analysis. However, our current metagraph patterns are not designed to capture these evasive behaviours.

In the case of *LockBit*, 2 of the 7 samples ran properly. The other 5 samples may have had evasive behaviors. These 2 samples contribute matches for the metagraphs "High Entropy Writing" and "Extraction and Execution". These numbers show that metagraphs can help the experts quickly identify traces with interesting behavior. Interestingly, we can find on average 2450 matches of the "High Entropy Writing" metagraph pattern in the 2 samples. This can be a strong indicator that these samples executed ransomware payloads encrypting many files. The low number of "High Entropy Writing" matches for the other families probably indicates false positives.

Let us point out that malware family may demonstrate multiple metagraph patterns simultaneously. This echoes the difficulty of accurately classifying malware families in practice, as the different families may adopt similar attack techniques. The samples of *GCleaner* and *LockBit* are easier to tag, as they demonstrate on average significantly more "Extraction and Execution" and "High Entropy Writing" patterns respectively than the other family.

Remark that, in some of the samples of *SnakeKeyLogger* and *LockBit*, we fail to find matches to some metagraph patterns. It is possible that these patterns are not enough to capture possible malicious payloads in these samples, for example capturing the

evading behaviours. The results imply the necessity of enriching the metagraph patterns while exploring BAGUETTE graphs.

The "Auto-Run" metagraph is the least matched pattern across these families. The reason is "Auto-Run" is more complex than the other metagraph patterns. It contains more nodes / links inside and therefore exactly matching "Auto-Run" in BAGUETTE graphs poses more constraints than the simpler metagraphs. On the other hand, "Auto-Run" metagraph can convey more accurate description of the behaviour than the other patterns. It raises a trade-off between the flexibility and complexity of metagraph patterns as more accurate (less flexible) descriptions of payloads require more complex structures of metagraphs but could raise less false positives.

5 USE CASE STUDY WITH BAGUETTE VISUALIZATION

We demonstrate how to use BAGUETTE graphs for visualization for investigating dynamic analysis reports.

5.1 Graph pruning for visualization

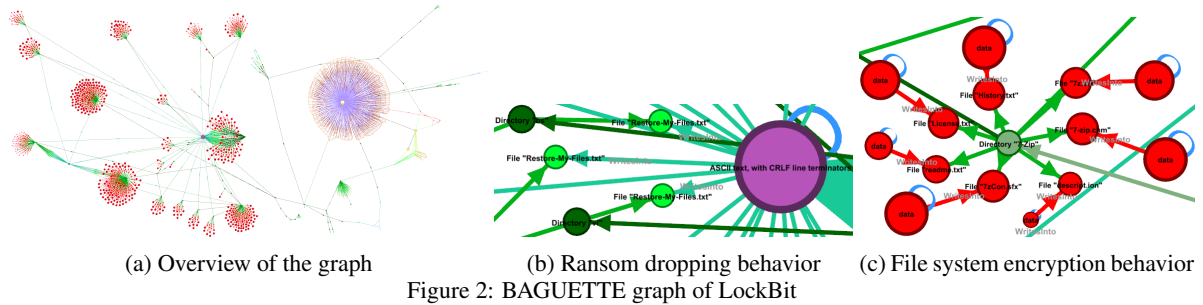
In the previous subsection, we used metagraphs to identify suspicious behaviours. However, BAGUETTE graphs can be very large (on average 10,000 nodes and 15,000 directed edges in our use case study), making manual graph exploration difficult. We thus introduce a pruning method of the raw BAGUETTE graphs for better visualizing the traces. We remove *Import* nodes not connected to a *File* node. We only keep the *File Handle* nodes, with the corresponding *File* nodes connected to a *Process* node via a "Runs" or a "Is Argument Of" edge (i.e. when the process's command line references that file). We only keep the *Thread* nodes that are started remotely (i.e., with an *Injected Thread* edge). Remotely started threads are typically used by malware for evasion or privilege escalation¹⁰. *Process* nodes that are only linked to their parent processes are removed. We keep the *Registry Keys* and *Registry Key Entries* that are modified, removed or created by a process. We keep *API Call* nodes that are linked to other types of nodes, which indicate they might be more significant.

Besides, we highlight important nodes with bigger sizes and use different color to denote different types of nodes. The color of each node type is given in Figure 1. The size and the color of a node can vary

¹⁰<https://attack.mitre.org/techniques/T1055/>

Metagraph	GCleaner			SnakeKeyLogger			LockBit		
	p	n	σ	p	n	σ	p	n	σ
(a) High-Entropy Writing	97.57%	1.53	0.59	13.53%	1.08	0.28	28.57%	2450.0	1878.0
(b) Changed File Type	97.57%	1.0	0.0	4.59%	1.05	0.22	14.29%	1.0	0.0
(c) Covert Execution	98.38%	1.0	0.0	0%	-	-	0%	-	-
(d) Extraction and Execution	98.38%	2.97	0.17	13.53%	1.0	0.0	0%	-	-
(e) Auto-Run	0%	-	-	0%	-	-	28.57%	1.0	0.0

Table 1: Statistics extracted for malware samples for three families of malware



according to its content. *Diff* nodes are bigger when they contain larger buffers. Also, *Diff* node color has a gradient from blue (small entropy of "global data") to purple (high entropy of "global data"). *File* nodes are green when the files still exist at the end of the analysis and grey when they have been deleted. *Registry Key* and *Registry Key Entry* nodes also turn to grey when they are deleted. Furthermore, *Registry Key* nodes change to red when their names contain UTF-16 values that are not mapped to any character or symbol. The width of the edges "Has Similar Content" depends on the similarity (narrower edges for smaller similarity, thicker edges for higher similarity).

Finally, when a match was found for a metagraph in a given BAGUETTE graph, we color the matching vertices, edges and arrows in red.

5.2 LockBit (ransomware) analysis

LockBit is a ransomware family that appeared in 2019. Its BAGUETTE graph can be seen in Figure 2a.

Its pruned graph is still larger than average, probably because it has much more interactions with the system. Four different types of nodes groups can be identified: first, a purple and yellow circle on the right, indicating network activity. Second, a purple node in the middle linked to many other nodes, indicating that the same data was found in many file I/O operations. Then, the group of dynamic dll import at the bottom right. Last, multiple green and red areas indicating high entropy contents written to many files. These subgraphs matched the (a) **High-Entropy Writing** behaviour.

Figure 2c zooms out one of these green and red

groups where 7 (a) metagraphs can be found. We can see that file contents of high entropy are written into many files inside the "7-Zip" directory. In fact, this behavior can be seen in many other directories ("Audacity", "Inkscape", etc.). Upon inspection, we find an average entropy of about 6.2 bits per byte. Producing file contents with such a high entropy implies of encryption or compression of files. Furthermore, the type of the written files is tagged as "data" by libmagic, which indicates that it may be not compressed / packed with any well-known off-the-shelf compression algorithms. An assumption is that the malware sample encrypted these files. Note that we cannot see the writing operation as the sample uses asynchronous reading¹¹, which is not handled by Cuckoo, and thus absent from the reports.

To confirm this hypothesis, Figure 2b shows a single *Diff* node was linked to many *File* nodes. By checking the *Diff* node, we find this node only contains writing operations and the bytes written into the connected *File* nodes named with "Restore-My-Files.txt" are exactly the same. Upon inspection, the *Diff* node shows the file content is ASCII text. The files "Restore-My-Files.txt" were created in many different directories, which confirms that this is a ransom message to the victim. More specifically, the writing buffer of the *Diff* node contains the text "LockBit 2.0 Ransomware. Your data are stolen and encrypted. The data will be published on TOR website [...]". We cannot observe data exfiltration, as it was done with low-

¹¹A call to "NtSetInformationFile" with the parameter "FileCompletionInformation" indicates undocumented AsyncIO operations

level methods, e.g. using special I/O API calls with undocumented flags. Instead, we can only observe sockets binding to all the opened interfaces without sending anything.

All of these examples of BAGUETTE graphs show that: 1) A global view of the dynamic traces in the BAGUETTE graph help human analysts to quickly focus on possibly interesting behaviours. 2) Zooming the graph and unveiling detailed information allows human analysts to conduct deeper investigation into specific payloads and help producing explanations for the malware sample's behaviour.

6 CONCLUSION

Malware analysis consists of understanding the objective of a malware, and its various attack, protection, and evasion techniques. BAGUETTE is a post-processing of the dynamic analysis report produced by Cuckoo sandbox relying on a heterogeneous graph. Based on this representation, it is easy to specify metagraphs that describe suspicious behaviours and to use them to filter a database of analysis reports to highlight malware displaying a given suspicious and precise behaviors. Finally, experts can use BAGUETTE graphs to manually analyze the interaction between the malware and the host and quickly discover and verify hypotheses. In the future, we will use metagraphs to identify clusters of malware exhibiting partially similar behaviors and qualify unknown malware by recognizing partially known behaviors through metagraphs. We will also try to create an algorithm to learn significant metagraphs to separate datasets of BAGUETTES.

REFERENCES

- Alptekin, H., Yildizli, C., Savas, E., and Levi, A. (2019). TRAPDROID: Bare-Metal Android Malware Behavior Analysis Framework. In *2019 21st International Conference on Advanced Communication Technology (ICACT)*, pages 664–671, PyeongChang Kwang-woon_Do, Korea (South). IEEE.
- Bridges, R., Hernández Jiménez, J., Nichols, J., Goseva-Popstojanova, K., and Prowell, S. (2018). Towards Malware Detection via CPU Power Consumption: Data Collection Design and Analytics. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1680–1684. ISSN: 2324-9013.
- Chen, Z.-G., Kang, H.-S., Yin, S.-N., and Kim, S.-R. (2017). Automatic Ransomware Detection and Analysis Based on Dynamic API Calls Flow Graph. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, pages 196–201, Krakow Poland. ACM.
- Ding, Y., Xia, X., Chen, S., and Li, Y. (2018). A malware detection method based on family behavior graph. *Computers & Security*, 73:73–86.
- Fang, Y., Lin, W., Zheng, V. W., Wu, M., Shi, J., Chang, K. C.-C., and Li, X.-L. (2021). Metagraph-Based Learning on Heterogeneous Graphs. *IEEE Transactions on Knowledge and Data Engineering*, 33(1):154–168.
- Grimmer, M., Röhling, M. M., Kricke, M., Franczyk, B., and Rahm, E. (2018). Intrusion detection on system call graphs. *Sicherheit in vernetzten Systemen*, pages G1–G18.
- Kolosnjaji, B., Zarras, A., Webster, G., and Eckert, C. (2016). Deep Learning for Classification of Malware System Call Sequences. In Kang, B. H. and Bai, Q., editors, *AI 2016: Advances in Artificial Intelligence*, volume 9992, pages 137–149. Springer International Publishing, Cham. Series Title: Lecture Notes in Computer Science.
- Park, Y., Reeves, D., Mulukutla, V., and Sundaravel, B. (2010). Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research - CSIIRW '10*, page 1, Oak Ridge, Tennessee. ACM Press.
- Piplai, A., Mittal, S., Abdelsalam, M., Gupta, M., Joshi, A., and Finin, T. (2020). Knowledge Enrichment by Fusing Representations for Malware Threat Intelligence and Behavior. In *2020 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 1–6.
- Qiang, W., Yang, L., and Jin, H. (2022). Efficient and Robust Malware Detection Based on Control Flow Traces Using Deep Neural Networks. *Computers & Security*, 122:102871.
- Raff, E. and Nicholas, C. (2020). A Survey of Machine Learning Methods and Challenges for Windows Malware Classification. *NeurIPS 2020 Workshop: ML Retrospectives, Surveys & Meta-Analyses (ML-RSA)*.
- Sihwail, R., Omar, K., and Zainol Ariffin, K. A. (2018). A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. *International Journal on Advanced Science, Engineering and Information Technology*, 8(4-2):1662.
- Wang, S., Chen, Z., Yu, X., Li, D., Ni, J., Tang, L.-A., Gui, J., Li, Z., Chen, H., and Yu, P. S. (2019). Heterogeneous Graph Matching Networks for Unknown Malware Detection. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 3762–3770, Macao, China. International Joint Conferences on Artificial Intelligence Organization.
- Wunderlich, S., Ring, M., Landes, D., and Hotho, A. (2020). Comparison of System Call Representations for Intrusion Detection. *Springer, Cham*, 951:14–24.
- Xiao, F., Lin, Z., Sun, Y., and Ma, Y. (2019). Malware Detection Based on Deep Learning of Behavior Graphs. *Mathematical Problems in Engineering*, 2019:1–10.