Towards Logical Specification and Checking of Evasive Malware

Andrei Mogage[1,2], Dorel Lucanu[1,*]

[1]Alexandru Ioan Cuza University, Iași, Romania
[2]Bitdefender, Iași, Romania

Abstract

The thesis proposes a new approach of combining formal methods and malware analysis for quickly determining if an application has specific malicious capabilities. The proposed solution is a Formal Tainting-Based Framework that uses a combination of binary instrumentation, taint analysis, and temporal logic in order to selectively extract behavioral properties of a malware. These are then formalized in order to check if the application expresses certain capabilities. The findings are accompanied by a concrete implementation, which proved effective and efficient against real-life malware, as highlighted by an evaluation. Furthermore, the framework has been used in actual cyber forensics investigations, reducing the time and efforts of security researchers. In this paper we also investigate how the framework can be applied in the context of evasive malware, allowing us to bypass anti-analysis techniques in order to reach the malicious core of the malware. This feature has not been previously covered by our other papers.

Keywords

temporal logic, taint analysis, malware analysis, formal methods

1. Introduction

1.1. Context and Challenges

The purpose of this thesis is to combine knowledge from academia and expertise from the cyber-security industry to enhance a critical task: malware analysis. This can be a complex process considering that highly complex and evasive malware have been on an increasing trend for the last few years [1, 2, 3].

However, a rapid verification of a potential capability is more desired, in situations where quick triage is necessary. For instance, a more useful and meaningful solution is to automatically get a verdict/hint of what the program is capable of, e.g.: *Does it steal data? Is it a ransomware?* Moreover, it is very helpful *to know the arguments that led to that verdict.* Knowing which semantic correlations have been made in order to reach a verdict might allow us to determine the correctness of the assessment. For instance, we want to know which chain of related events has led to the verdict that the program has the capability of stealing data. This is in contrast with the AI-based approaches, e.g. machine learning, where such correlations are merely statistical.

There are some problems that might interfere with this process:

- *Obfuscation* - This limits the usability of static analysis or even some dynamic analysis tools.
- *Anti-analysis techniques* - This limits or prevents the analysis, regardless of the analysis type, depending on the techniques implemented.
- *Traceability* - While an analysis might capture the called APIs, such as a sandbox, there is no direct correlation between passing the output of an API to the input of another. This happens because the individual instructions are usually not traced, due to performance reasons (i.e., minimizing overhead).

PhD Symposium of the 19th International Conference on Integrated Formal Methods (iFM) at the University of Manchester, UK, 12 November 2024.
*PhD Supervisor
andrei.mogage@gmail.com (A. Mogage)
0000-0002-3533-7573 (A. Mogage); 0000-0001-8097-040X (D. Lucanu)

The issue has also been the subject of extended research, both through academic [4, 5] and industry contexts [6].

1.2. Contribution

Our contribution consists in a Formal Tainting-Based Framework (FTBF), which uses binary instrumentation to control an application, taint analysis to extract relevant behavioral properties into a trace, a new temporal logic to analyze the tainted trace, Tainting-Based Logic (TBL), and a formally described behavior (capability) that the user wishes to check. While taint analysis is more commonly used to detect vulnerabilities or input reaching certain *sinks*, we use it to extract complex behavioral properties, that are later formally checked against malicious capabilities. Even though we only include the highlights in this paper, the entire framework is completely defined in [7] (submitted and accepted at iFM 2024). To our best knowledge, this is the first time when dynamic instrumentation, taint analysis, and temporal-logic-based checking are integrated into a formal framework for malware analysis. The result is a sound confirmation of whether the application has a specific capability, ensured through the formal checking process. The formalization component ensures that the results are sound, providing accurate details, rather than *educated guesses* (e.g., based on naive static heuristics from sandbox reports). Nonetheless, the system is built in such a fashion that it requires no knowledge of formal methods for a user to be able to use it, but rather a basic understanding of how some capabilities can be implemented at low-level.

In order to provide a concrete example of an ideal solution, we may refer to the task of an analyst whose purpose is to determine whether the analyzed application possesses certain traits or exerts some interesting behavior:

- *Is it a ransomware?* - i.e., encrypts files and asks for a ransom in return;
- *Does it exfiltrate data?* - i.e., transfers sensitive data without authorization;
- *Does it deploy techniques to deter security solutions?* - i.e., attempts to disrupt the activity of a security solution in order to ensure the success of other malicious activities;
- *Is it an Active Persistent Threat (APT)?* - i.e., ensures the persistence over system reboot (or even reinstall), keeps a low profile (hard to detect its resources - files, registries, processes), contacts suspicious servers for further commands.

Assuming that the analysis tool can deter the anti-analysis techniques, thus managing to avoid execution refusal, our extension will firstly taint sensitive data (API output, registers, memory) and monitor its propagation towards the end of the execution. The tainting events(i.e., tainting and tracking) will be formalized, by introducing predicates that express behavioral properties. Finally, we may devise certain formulas that make use of temporal operators in order to check whether the analyzed application, through its formalized output, has certain patterns of behavioral capabilities. This, of course, implies a minimal knowledge of how certain attacks can be implemented, but the entire flow is effortless and extendable. Furthermore, the entire process of formally defining rules can be simplified by using artificial intelligence for obtaining hints regarding capabilities. The system is also accompanied by a concrete implementation (for the Windows OS) and an evaluation process that proves its benefits. The implementation has actually been integrated into two Dynamic Binary Instrumentation (DBI) tools, Intel Pin [8] and COBAI [9]. The framework is sustained by experiments using actual malware families, evaluating its efficiency in capturing various capabilities (e.g. code injection, encryption, deobfuscation, privilege escalation). Moreover, the system has been used in real-life scenarios, during cyber forensics, proving useful for speeding up the process of analysis and filtering data. Another important aspect is providing the capability extraction features while handling anti-analysis techniques implemented by evasive malware. This aspect has not been included in [7], but is discussed in Section 3.3, where we combine FTBF with COBAI to achieve this.

In the following, we intuitively describe the formal framework in Section 2, along with necessary elements for applying it towards malware analysis. We summarize our evaluation results against malware families and discuss the transparency features in Section 3, and we conclude in Section 4.

2. A Formal Tainting-Based Framework for Malware Analysis

The Formal Tainting-Based Framework (FTBF) uses Dynamic Binary Instrumentation (DBI) to instrument the target application, while informing the taint analysis component of various events that occur during execution (instrumented instructions, memory updates, etc.). According to a taint policy provided as input, some data will be tainted and monitored throughout the execution. All tainting-related events are then formalized using Tainting-Based Logic (TBL) and placed into a formal tainted trace. Lastly, this trace is checked against *rules that specify capabilities* (specified in TBL), which ultimately provides a verdict whether the application has the capability described by the rule.

The syntax of TBL consists of three main categories of sentences:

1. *behavioral patterns*, whose instances describe behavioral properties of executions;
2. *potential/capability patterns*, whose instances describe capabilities of the executions;
3. *rules*, which relates behavioral properties and capabilities of an execution.

An example of a behavioral pattern is:

$$(\mathit{TaintedAPI}(\texttt{CreateFile}) \wedge \mathsf{X}(\mathit{Tainted}(T))) \\ \text{andthen } \mathit{PropToAPI}(\texttt{ReadFile}, T)$$

which, intuitively, describes that in the current state a file is created or accessed the first time, and it is read at some later state. The parameter T is the tainting tag used to track the file handle. Despite its name, `CreateFile` is also used to open existing files. andthen is part of TBL and newly introduced, and describes the fact that a property ϕ_2 occurs at a certain time in the future after another one ϕ_1. X is borrowed from Temporal Logics and refers to the next state.

An example of a capability pattern is:

$$\texttt{FileRead(T)}$$

where the predicate describes the capability of a program to read the contents of a file, where the T variable will be instantiated by a behavioral pattern (see the definition for capability rules below).

A simple example of rule is

$$(\mathit{TaintedAPI}(\texttt{CreateFile}) \wedge \mathsf{X}(\mathit{Tainted}(T))) \\ \text{andthen } \mathit{PropToAPI}(\texttt{ReadFile}, T) \\ \vdash \\ \texttt{FileRead}(T)$$

which helps us to deduce that our program has the capability of reading the contents of a file. The behavioral pattern is the same introduced earlier, where we expect a propagation of the taint symbol, T (and its associated tainted data), from the `CreateFile` API to `ReadFile`. $\vdash$ is an entailment relation, i.e., $A \vdash B$ means *if A, then B*.

FTBF has been introduced as a general framework for checking program capabilities. Our main goal, however, is to channel its usage towards malware analysis. Specifically, to apply FTBF in those scenarios where a specific malware capability (such as keylogging, encryption, or data ex-filtration) needs to be quickly detected.

The requirements for doing so are related to taint policies and capability rules. We need to define policies that capture factual behaviors related to malicious activities, along with rules that define a behavioral pattern encoding a malicious capability. In doing so, we use the following sentences:

$$\begin{aligned} \texttt{Behavioral Facts} : \{&\mathit{Tainted}, \mathit{Untainted}, \mathit{TaintedAPI}, \mathit{TaintedAPICond}, \mathit{PropToAPI}, \\ &\mathit{PropToAPICond}, \mathit{PropToReg}, \mathit{UntaintedReg}, \mathit{PropToMem}, \\ &\mathit{UntaintedMem}, \mathit{TaintedMemAccess}, \mathit{TaintedCodeExecuted}\}. \\ \texttt{Capabilities} : \{&\mathit{DisableWindowsEvent}, \mathit{C2Communication}, \mathit{RansomwareBehavior}, \\ &\mathit{DebuggerDetection}, \mathit{DisableUAC}, \mathit{PayloadDeobfuscation}, \end{aligned}$$

CodeInjection, *PrivilegeEscalation*, *SGXSupplyChainAttack*}

`Constants` : {`CreateFile, ReadFile, VirtualAlloc, …`}∪

{ `eax, rbx, esi,…` }

The first category of constants refers to Windows APIs, whereas the latter describes CPU registers.

3. Evaluation

In this section, we present the potential of the analysis framework by testing it against a suite of malicious applications.

3.1. Method

We have selected eight malware families and created one rule for each family, such that we will capture different capabilities, depending on the target application. The selection process has been random, but with a focus on selecting malware families that are still relevant (i.e., seen in recent attacks or that represented a strong inspiration for subsequent cyber threats).

The number of samples for each family varies between 6 and 37, depending on their public availability. All applications have been collected from two publicly available sources: VirusTotal [10] and Malpedia [11, 12]. Each malware family is briefly described below:

- *Al-Khaser* [13]: An application which deploys a myriad of malicious techniques, borrowed from complex malware, in order for analysts to test analysis solutions. While not an actual malware (the only one in this situation in our test suite), it is relevant for capturing anti-analysis capabilities;
- *Avaddon* [14]: A ransomware family initially seen in 2020, with a strong RaaS (Ransomware-as-a-Service) model, which affected a high number of victims spanning multiple industries and countries. Even if the RaaS has been shutdown in 2021, the malware is still being distributed worldwide and several new ransomware families emerged later on and share common practices and even source code [15];
- *RokRat* [16]: A Remote Administration Tool (RAT), first reported in 2017, RokRat is a malware distributed in multiple malicious campaigns and makes use of numerous attack vectors and techniques in order to infiltrate a victim and then start an entire chain of receiving and executing commands. While it has not suffered significant changes at its core lately, it has been combined with new techniques to ensure its success in affecting victims [17];
- *Darkside* [18]: A ransomware family renowned for high-impact attacks targeting large corporations and governments, culminating with the attack on the Colonial Pipeline in the United States;
- *CobaltStrike Beacon* [19]: A beacon is a payload from Cobalt Strike [20] which mimics attacker activities seen in the wild. However, beacons have been used directly in actual attacks as an initial step for deploying other malware tools on the systems of affected victims;
- *Phant0m* [21]: A tool capable of tampering with the Event Logging Service integrated into the Windows OS, resulting in the operating system not logging critical events that occur. This leads to attacks that are more stealthy and harder to investigate;
- *HermeticWiper* [22]: A malware involved in cyber-attacks targeting Ukraine, emerged during the recent war, which disrupts the functionality of systems;
- *Makop* [23]: An infamous ransomware family and group, which has been active for at least the last 4 years.

For each malware category, we have verified one malicious capability:

- *Al-Khaser* - Debugger detection;

- *Avaddon* - Disable User Account Control (UAC);
- *RokRat* - Code Injection;
- *Darkside* - C2 Communication;
- *CobaltStrike Beacon* - Deobfuscation;
- *Phant0m* - Disable Windows Event Logging;
- *HermeticWiper* - Privilege escalation;
- *Makop* - Ransomware Behavior.

All experiments have been conducted inside a virtual machine (using VMWare Workstation 16), with a Windows 10 as guest with 8 GB of RAM, 4 cores and virtualization enabled, while the host uses Windows 11 on an Intel i7-11800H CPU @2.30GHz and 32 GB of RAM.

3.2. Results

For simplicity, we have summarized the results for each test in Table 1. The tables contain the name of the malware family, along with the duration (in seconds) necessary to analyze and capture the capabilities described by the rules and the duration of a native execution. All experiments were successful (i.e. the capability has been correctly identified). Moreover, each analysis instance was achieved under 30 seconds, except for a sample for Makop ransomware, where some anti-analysis tricks were deployed, which slowed the analysis process.

Table 1
Summary of results

Malware Family	Avg. Analysis (s.)	Avg. Native execution (s.)
Al-Khaser	7.73	TIMEOUT
Avaddon	14.375	TIMEOUT
RokRat	22.69	19.72
DarkSide	15.87	14.68
CobaltStrike	4.13	TIMEOUT
Phant0m	6.22	1.22
HermeticWiper	9	TIMEOUT
Makop	20.42	TIMEOUT

These experiments bring forward the potential of such an analysis solution to quickly identify if a program expresses a specific capability or not. Naturally, the analysis process will bring an additional overhead compared with the program executing outside an execution environment. Because of this, we have also tested the duration under native execution. However, while the analysis time represents only duration until the capability is expressed, we could not verify at which exact moment the same capability is expressed natively without tampering with the applications. Therefore, we have chosen to estimate the total time necessary to complete the execution, with a provided timeout of **10 minutes**. This leads to three scenarios of comparison:

- **TIMEOUT** on native execution - this was caused by applications that either take longer than 10 minutes to complete their execution, or continue executing endlessly, e.g. waiting for network commands or scanning possibly new files. This category only applies to native execution, while all analysis instances ended successfully.
- Native duration is **longer** than analysis duration - this is directly related to the fact that the execution of the analyzed program is terminated as soon as the capability has been expressed, as an optimization technique.
- Native duration is **shorter** than analysis duration - this represents the overhead of the analysis solution.

Table 2
Evaluation scenarios per malware family

Malware Family	TIMEOUT	LONGER NATIVE	LONGER ANALYSIS
Al-Khaser	15	0	0
Avaddon	8	0	0
RokRat	1	1	24
DarkSide	0	4	12
CobaltStrike	37	0	0
Phant0m	0	22	0
HermeticWiper	6	0	0
Makop	19	0	0

We have summarized how many of samples per malware family fall under these three scenarios in Table 2.

Nonetheless, an actual real-life comparison is not with a native execution, but rather with the effort, especially regarding the time consumed, of an analyst. Another important remark relates to the generality of these rules, i.e. how well they can capture a capability. This is directly influenced by how well a rule is constructed and if it considers multiple scenarios, because there might be multiple ways of achieving an expecting results, e.g. by combining different APIs or instructions.

In this regard, a prerequisite for a user of this analysis solution is to have some general information on such combinations that might lead to an expected output. Nonetheless, the effort required to construct a rule, along with the analysis duration, are significantly lower than manually analyzing an application or by filtering and correlating a myriad of results provided by other automated solutions, such as a sandbox. Furthermore, the process of crafting rules for capturing capabilities can be simplified by using Large Language Models (LLMs), as presented in [24]. The user can query a LLM for an overall idea of how capabilities can be implemented, and then proceed with the formalization process.

3.3. Transparency

Although we have chosen Pin as the DBI component, we have also experimented with COBAI [9] as well, due to its main focus on transparency, highly needed for evasive applications. This aspect has not been previously discussed in [7], here is the first time we discuss the transparency of FTBF.

An analysis environment is considered ”transparent” if a malware is unable to detect its presence. Usually, these attacks result in execution refusal or code executing outside the analysis environment. Despite some reports stating that DBI might be unsuitable against evasive malware (e.g., [25]), our empiric evaluation against malware or personally developed applications that deploy anti-analysis techniques reveal that COBAI can be properly used against this category of malware. This also takes into consideration various aspects of implementation details, Windows OS architecture, and malware analysis in general. More details about this have been included in [9]. In order to ensure transparency for both itself and the rest of the environment, COBAI deploys a series of heuristics based on anti-analysis techniques seen in malware, but also in analysis-testing frameworks (such as Al-Khaser [13] or Pafish [26]), or in academic literature [27, 28, 29]. We exemplify some of these heuristics below, where we mention how a malware would try to detect the analysis process:

- *System Artifacts*: Querying the presence of certain sandbox or VM-specific artifacts, such as files (i.e., hypervisor services), registries (i.e., sandbox configuration), network resources (i.e., IP or MAC addresses), processes (i.e., those related with the virtual machine or sandbox), or hardware properties (i.e., CPU virtualization flags);
- *Debuggers*: Scanning for a list of consecrated debugger names based on the list of active processes, files, or the currently loaded modules;
- *DBIs*: Scanning the memory pages, threads, loaded modules, and other resources in order to detect any possible anomaly;

- *Time-based Techniques*: Using time for either measuring the time between certain commands (which would be increased during the analysis, depending on the overhead), or verifying if the time has been somehow accelerated (i.e., skipping `sleep` commands);
- *Human Interaction*: Verifying that certain events occur, that would be normally expected under a legitimate scenario, such as cursor movement, mouse clicks, or keystrokes;

In order to counter-attack these techniques, COBAI will monitor specific instructions or APIs and provide fake results when the program attempts at querying certain artifacts that would reveal the analysis framework or environment. During some experiments with evasive malware, the results reveal a good combination of COBAI and FTBF: COBAI will ensure transparency, in order for FTBF to be able to capture the malicious capabilities that follow.

4. Conclusion

The Formal Tainting-Based Framework can be a powerful formalism and tool for quick and safe verification of malware capabilities. Having minimal knowledge on how capabilities can be implemented, a user can use taint policies and capability rules to determine if a malware express a specific behavior.

The entire process uses binary instrumentation, taint analysis, and the proposed Tainting-Based Logic, efforts that combine industrial efforts and academic knowledge, leading to sound and precise results. Our claims are sustained by evaluation results using multiple malware families, with a rule for each distinct capability. The results hint at significantly lowering the burden of a cyber researcher. This fact is also highlighted through real-life use-cases, where we used the current implementation and rule set to determine which malware samples might have a particular capability. Furthermore, we also tackled the case of evasive malware, where one must first counter-attack the anti-analysis techniques in order to reach the actual payload of a malware. This case has been handled by using COBAI as the DBI component, which ensures transparency of the framework and the rest of the environment.

References

[1] J. Coker, Evasive malware threats on the rise despite decline in overall attacks, 2020. URL: https://www.infosecurity-magazine.com/news/evasive-malware-rise-decline/.

[2] W. T. Inc, New research: Fileless malware attacks surge by 900% and cryptominers make a comeback, while ransomware attacks decline, 2021. URL: https://www.globenewswire.com/en/news-release/2021/03/30/2201173/0/en/New-Research-Fileless-Malware-Attacks-Surge-by-900-and-Cryptominers-Make-a-Comeback-While-Ransomware-Attacks-Decline.html.

[3] H. N. Security, The hidden picture of malware attack trends, 2023. URL: https://www.helpnetsecurity.com/2023/04/06/malware-attack-trends-q4-2022/.

[4] A. Afianian, S. Niksefat, B. Sadeghiyan, D. Baptiste, Malware dynamic analysis evasion techniques: A survey, ACM Comput. Surv. 52 (2019). URL: https://doi.org/10.1145/3365001.

[5] F. A. Aboaoja, A. Zainal, A. M. Ali, F. A. Ghaleb, F. J. Alsolami, M. A. Rassam, Dynamic extraction of initial behavior for evasive malware detection, Mathematics 11 (2023). URL: https://www.mdpi.com/2227-7390/11/2/416. doi:10.3390/math11020416.

[6] Defeating evasive malware whitepaper, 2023. URL: https://www.vmray.com/resources/defeating-evasive-malware-whitepaper/.

[7] A. Mogage, D. Lucanu, A formal tainting-based framework for malware analysis, Springer Lecture Notes in Computer Science (2024). The 19th International Conference on Integrated Formal Methods (iFM).

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, Association for Computing Machinery, New York, NY, USA, 2005, pp. 190–200. URL: https://doi.org/10.1145/1065010.1065034.

[9] V. Craciun, A. Mogage, D. Lucanu, Full transparency in DBI frameworks, 2023. URL: https://doi.org/10.48550/arXiv.2306.13529. arXiv:2306.13529.
[10] Virustotal: How it works, 2023. URL: https://docs.virustotal.com/docs/how-it-works.
[11] D. Plohmann, M. Clauss, S. Enders, E. Padilla, Malpedia: A Collaborative Effort to Inventorize the Malware Landscape, The Journal on Cybercrime & Digital Investigations 3 (2018). URL: https://journal.cecyf.fr/ojs/index.php/cybin/article/view/17.
[12] Malpedia - usage, 2023. URL: https://malpedia.caad.fkie.fraunhofer.de/usage/references.
[13] Noteworthy, LordNoteworthy/al-khaser, 2024. URL: https://github.com/LordNoteworthy/al-khaser, original-date: 2015-11-12T18:35:16Z.
[14] J. Yuste, S. Pastrana, Avaddon ransomware: An in-depth analysis and decryption of infected systems, Computers & Security 109 (2021) 102388. URL: https://www.sciencedirect.com/science/article/pii/S0167404821002121.
[15] One source to rule them all: Chasing AVADDON ransomware | mandiant, 2023. URL: https://cloud.google.com/blog/topics/threat-intelligence/chasing-avaddon-ransomware.
[16] Introducing ROKRAT, 2017. URL: https://blog.talosintelligence.com/introducing-rokrat/.
[17] etal, Chain reaction: ROKRAT's missing link, 2023. URL: https://research.checkpoint.com/2023/chain-reaction-rokrats-missing-link/.
[18] DarkSide ransomware explained: How it works and who is behind it, 2023. URL: https://www.csoonline.com/article/570723/darkside-ransomware-explained-how-it-works-and-who-is-behind-it.html.
[19] MAR 10339794-1.v1 - cobalt strike beacon | CISA, 2021. URL: https://www.cisa.gov/news-events/analysis-reports/ar21-148a.
[20] Cobalt strike beacon | cobalt strike features, 2024. URL: https://www.cobaltstrike.com/product/features/beacon.
[21] H. Dalabasmaz, Phant0m (github), 2023. URL: https://github.com/hlldz/Phant0m.
[22] HermeticWiper: A detailed analysis of the destructive malware that targeted Ukraine, 2022. URL: https://www.threatdown.com/blog/hermeticwiper-a-detailed-analysis-of-the-destructive-malware-that-targeted-ukraine/.
[23] P. Paganini, Dissecting the malicious arsenal of the Makop ransomware gang, 2023. URL: https://securityaffairs.com/143452/malware/dissecting-makop-ransomware.html.
[24] A. Mogage, A.I. Assisted Malware Capabilities Capturing, Procedia Computer Science (2024). In the proceedings of the 28th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems (KES 2024).
[25] J. Kirsch, Z. Zhechev, B. Bierbaumer, T. Kittel, PwIN - pwning intel piN: Why DBI is unsuitable for security applications, in: J. Lopez, J. Zhou, M. Soriano (Eds.), Computer Security, Springer International Publishing, 2018, pp. 363–382. doi:10.1007/978-3-319-99073-6_18.
[26] A. Ortega, a0rtega/pafish, 2024. URL: https://github.com/a0rtega/pafish, original-date: 2012-07-01T11:06:40Z.
[27] A. S. Filho, R. J. Rodríguez, E. L. Feitosa, Evasion and countermeasures techniques to detect dynamic binary instrumentation frameworks 3 (2022) 11:1–11:28. URL: https://dl.acm.org/doi/10.1145/3480463. doi:10.1145/3480463.
[28] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, L. Cavallaro, Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed), in: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, 2019, pp. 15–27.
[29] D. C. D'Elia, E. Coppa, F. Palmaro, L. Cavallaro, On the dissection of evasive malware, IEEE Transactions on Information Forensics and Security 15 (2020) 2750–2765. doi:10.1109/TIFS.2020.2976559.