Understanding LLMs Ability to Aid Malware Analysts in Bypassing Evasion Techniques

Miuyin Yong Wong
Georgia Institute of Technology
Atlanta, Georgia, USA

Kevin Valakuzhy
Georgia Institute of Technology
Atlanta, Georgia, USA

Mustaque Ahamad
Georgia Institute of Technology
Atlanta, Georgia, USA

Doug Blough
Georgia Institute of Technology
Atlanta, Georgia, USA

Fabian Monrose
Georgia Institute of Technology
Atlanta, Georgia, USA

Abstract

Over the past few years, the threat of malware has become increasingly evident, posing a significant risk to cybersecurity worldwide and driving extensive research efforts to prevent and mitigate these attacks. Despite numerous efforts to automate malware analysis, these systems are constantly thwarted by evasive techniques developed by malware authors. As a result, the analysis of sophisticated evasive malware falls into the hands of human malware analysts, who must undertake the time-consuming process of overcoming each evasive technique to uncover malware's malicious behaviors. This highlights the need for approaches that aid malware analysts in this process. Although active measures, such as forced execution and symbolic analysis, can automatically circumvent some evasive checks, they suffer from limitations like path explosion and fail to provide useful insights that analysts can use in their workflow. To fill this gap, we investigate how large language models (LLMs) can address shortcomings of symbolic analysis through the first comparative analysis between the two in bypassing evasion techniques. Our study leads to three key findings: (i) we find that LLMs outperform symbolic analysis in bypassing evasive code, especially in the presence of common code patterns, such as loops, which have historically posed a challenge for symbolic analysis, (ii) we show that LLMs correctly identify methods of bypassing evasive techniques in real-world malware, and (iii) we highlight how even in LLMs failure modes, human malware analysts can benefit from the step-by-step reasoning provided by the model.

Keywords

Malware Analysis; Large Language Model; Symbolic Analysis

ACM Reference Format:
Miuyin Yong Wong, Kevin Valakuzhy, Mustaque Ahamad, Doug Blough, and Fabian Monrose. 2024. Understanding LLMs Ability to Aid Malware Analysts in Bypassing Evasion Techniques. In *INTERNATIONAL CONFERENCE ON MULTIMODAL INTERACTION (ICMI Companion '24), November 04–08, 2024, San Jose, Costa Rica.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3686215.3690147

ICMI Companion '24, November 04–08, 2024, San Jose, Costa Rica

ACM ISBN 979-8-4007-0463-5/24/11
https://doi.org/10.1145/3686215.3690147

1 Introduction

Malware, malicious software designed to disrupt, damage, or gain unauthorized access to computer systems, poses a significant threat to cybersecurity worldwide, impacting businesses, governments, and individuals alike. This threat is especially pronounced in Latin America, where a 2023 Ernst & Young report indicated that 91% of companies reported a cyber incident, and 62% experienced a data breach.[1] These breaches impose a significant burden on medium and small companies, with each data breach costing $4.45 million on average.[2] To combat these attacks, it is essential to understand the capabilities of malicious software through malware analysis. There are two main methods of malware analysis: static and dynamic analysis. Static analysis examines the malicious code without executing it. In contrast, dynamic analysis observes the behavior of malware as it runs in a controlled environment (i.e. sandbox).

Unfortunately, modern malware employ various evasive techniques, such as code obfuscation and sandbox detection, to conceal their malicious capabilities from analysis. A recent study highlighted that analyzing malware that contain evasion techniques remains one of the most challenging tasks in practice, even for experienced malware analysts [24]. More specifically, Yong et al. [24] found that analysts must resort to the time-consuming process of switching between dynamic and static analysis when automated methods fail to overcome evasive checks, allowing malware to hide their malicious behaviors. Additionally, this complex process demands highly skilled malware analysts, a critical limitation given the significant shortage of qualified cybersecurity professionals.[3]

Over the years, significant efforts have been made to address malware evasion challenges that impede automated analysis. One common approach focuses on decreasing the opportunities for malware to detect it is being analyzed, either by leveraging more transparent monitoring environments [2, 7, 10, 18, 22] or explicitly hiding artifacts that reveal the true nature of the execution environment [3, 6, 8]. Despite the effectiveness of these approaches, these approaches are often limited to handling known evasion techniques. To bypass novel evasion techniques, more active strategies were developed to force malware to reveal hidden behaviors. Two common methods are forced execution [5, 11, 25] and symbolic execution [9, 19]. However, both strategies face time constraints and struggle to handle diverse conditions, often failing due to what is commonly known as the path explosion problem. Moreover, when

[1] EY Report on Cyber Incidents in Latin American Companies
[2] IBM Cost of a Data Breach Report 2023
[3] CSonline Cybersecurity Workforce Shortage

these approaches fail, they do not provide useful insights that analysts can integrate into their workflow.

To help fill this gap, we investigate the potential of large language models (LLMs) in assisting human analysts bypass evasion techniques. To accomplish this, we task the GPT 4o LLM with identifying the necessary conditions to trigger 56 distinct logic bombs, which closely resemble evasive techniques found in malware. To evaluate the effectiveness of the LLM, we also conduct the first comparative analysis between symbolic analysis and LLMs in the context of malware analysis. Based on our evaluation, we not only find that LLMs can aid in the process of bypassing evasion techniques but also address some of the shortcomings of symbolic analysis. The main contributions of this paper are the following:

- We identified which weaknesses of symbolic execution can be mitigated by LLMs.
- We tested LLMs effectiveness at identifying and overcoming evasion techniques in real-world malware.
- We identified weaknesses of LLMs that would benefit from future work.

2 Related Work

2.1 Countering Evasive Malware

Over the years, several approaches have been proposed to address evasion techniques that hinder automated analysis. These approaches can be grouped into two categories; preventative methods and active methods. Preventative methods aim to mitigate evasion techniques before executing the malware. The first preventative approach focuses on improving the transparency of dynamic analysis environments to prevent malware from detecting the controlled analysis setting. This can be achieved through bare-metal-based analysis [12, 17, 26] or hypervisor-based analysis [2, 7, 10]. Bare-metal analysis runs directly on hardware to avoid detection by malware, whereas hypervisor-based analysis uses virtualization to create isolated environments. The second preventative approach leverages existing knowledge about evasion techniques by employing predefined rules to hide known artifacts of the analysis environment [3, 6, 8, 15]. However, these two approaches do not adapt to malware with new methods of detecting analysis environments.

In contrast, active methods take a more direct approach by interacting with the malware during execution. Forced execution, manipulates the execution flow of the malware to bypass conditional checks used to hide malicious behaviors [5, 11, 25]. Another active strategy is symbolic execution, which simulates running malware by representing program variables and execution paths with symbolic expressions [1, 9, 19, 21]. These symbolic expressions can be passed to a constraint solver to determine the specific conditions required to trigger different behaviors. Unfortunately, both forced execution and symbolic execution face the limitation of path explosion, where the number of possible execution paths grows exponentially with the complexity of the program, making it computationally infeasible to explore all paths. Our approach also applies an active strategy while addressing their limitations.

2.2 Application of LLMs in Malware

Research in Large Language Models is rapidly expanding, driven by its potential to transform various fields, including cybersecurity. Notably, recent studies have demonstrated LLMs' capability in assisting cybercriminals to develop malware attacks [4, 16, 23]. These studies underscore the alarming potential of LLMs to enhance the abilities of cybercriminals, making malware more complex and difficult to detect. Despite this concern, there has been limited research into exploring the LLMs' potential to assist in the defense against malware attacks, a critical area to strengthen cybersecurity defenses [13]. Our study addresses this gap by exploring how LLMs can support human malware analysts in their analysis workflow.

3 Methodology

To evaluate the potential of LLMs to improve on existing automated code analysis systems and assist human malware analysts in bypassing evasion techniques, we investigate LLMs' effectiveness in analyzing challenging scenarios, such as logic bombs. Logic bombs are code designed to execute specific functionality only when certain conditions are met. These conditions are analogous to the evasive techniques employed by malware to conceal malicious behavior from malware analysis systems. We specifically focus on logic bombs that exploit well-known weaknesses of symbolic execution–such as unbounded loops and network interactions–as they provide a starting point for understanding how LLMs can enhance a malware analyst's toolkit.

For our LLM approach, we begin by assuming an analyst has run evasive code, such as an evasive malware sample or a logic bomb, and is tasked with figuring out how to bypass the evasive code. As evasive techniques will often cause malware to stop executing before exhibiting malicious behavior, simply monitoring the code that has been executed can narrow down which code is responsible for the evasive technique. We use a disassembler to extract the assembly code for the evasive technique along with any functions it calls outside of standard libraries. We avoid including standard library functions, as it greatly increases the tokens sent to the model. Since the model already knows the behavior of these commonly used functions, including them adds little value.

Next, we feed the disassembled code to a decompiler, which converts assembly into C-like code while preserving approximately the same behavior. Decompilers aid human understanding by translating complex lower-level code into simpler high-level concepts. We find that decompilation also helps LLMs correctly reason about the consequences of the code. The decompiled code is then provided to an LLM that is prompted to identify the correct function argument required to obtain the desired return value of the logic bomb. The prompt we give explicitly asks the model to explain its procedure step-by-step so that we can understand cases where it fails to find the right answer. The model is additionally capable of writing and executing its own Python scripts, which we have found improves the model's ability to correctly validate results or identify correct solutions through trying a variety of inputs. Finally, we validate the response by running the test executable with the output specified by the LLM to check if it triggers the logic bomb.

Our results on these logic bombs are compared to the results collected by Xu et al. [20] for symbolic analysis systems. These systems ingest the whole logic bomb executable, processing more code than what we provide the LLM. However, the extra code for each executable is limited to code introduced by the compiler and

standard libraries, which are necessary for the symbolic analysis system to accurately model the entire executable's behavior.

3.1 Dataset and Tools

We utilize an existing dataset from Xu et al. [20] consisting of 66 C and C++ programs. Each program contains a logic bomb that exploits a known weakness in symbolic analysis, such as handling multi-threaded code, loops, or external functions. We select the subset of logic bombs that involve finding the correct command line input to pass to the program, discarding logic bombs that require modifying the host system, such as altering the system time. Additionally, we excluded test cases involving integer overflows, as compiler optimizations removed the code for the logic bomb, even at the lowest optimization level (O0). Our tests were run with the 56 remaining logic bombs. In cases where tests are non-deterministic, we consider the answer given by the model to be correct if it matches the answer provided by the authors.

We chose the GPT 4o model from OpenAI for our LLM agent and complemented it with the IDA Pro disassembler and decompiler to extract information from each executable. The cost of using the GPT 4o to run our tests on all 56 logic bombs was approximately 2 dollars. We compare our results with those from the best performing symbolic analysis tool, angr [14], on the logic bomb benchmark [20]. We reran the publicly available benchmark on a more recent version of angr (v9.2.102) than originally used, however, we found that these results were slightly worse than angr's performance in the benchmark paper, with 4 fewer logic bombs triggered. Replicating the version of angr used in their work and compiling their benchmark with compilers available at the time of their publishing did not resolve these discrepancies. To eliminate any chance that our reproduction diminished angr's performance, we compare against the results from the original benchmark run by Xu et al. [20], which represent the best results achieved by angr.

Table 1: Results of GPT 4o and angr in solving various logic bombs from Xu et al. [20]. Categories represent areas of difficulty for symbolic analysis.

Category	# Tests	GPT 4o	angr
Buffer Overflow	4	0	**2**
Contextual Symbolic Value	4	**2**	0
Covert Propagation	10	**7**	4
Crypto Functions	2	0	0
External Functions	8	**4**	3
Floating Point	5	2	2
Loop	5	**4**	0
Parallel Program	5	**1**	0
Symbolic Jump	4	2	2
Symbolic Memory	9	7	7
Total	**56**	**29**	**20**

Table 2: Reasons that GPT 4o failed to trigger 27 logic bombs.

Failure Reason	# Cases
Inaccurate or incomplete pseudocode	13
Incorrect assumption	6
Verified in Python, Not C	3
Answer required system modification	2
Returned intermediate result	1
Unsuccessful "guess and check"	1
Mathematical error	1

4 Results

Our results, shown in Table 1, show that GPT 4o performs better than angr, the best performing symbolic analysis system on the logic bomb benchmark from Xu et al. [20]. More importantly, in cases where GPT 4o did not yield a correct answer, we could use the step-by-step reasoning provided in the model's output to understand the type of failure that occurred. In fact, after understanding the failure, we found that with minimal interaction, such as providing explicitly requested information or questioning stated assumptions, we could enable the model to come to the correct conclusion.

4.1 Where the LLM succeeds

The category where LLMs show a clear advantage over symbolic execution is in the handling of loops. Loops create many possible execution states, often leading to state explosions that hinder symbolic execution. In each of these cases, rather than attempting to solve the problem by mathematically breaking down the code, GPT 4o writes and executes a Python script containing a re-implementation of the logic bomb and identifies the correct value through brute force.

Additionally, GPT 4o is also able to identify answers to certain logic bombs without executing code, leading to an improvement in handling covert propagation. This category contains logic bombs that hide data flow using code constructs that are challenging for symbolic analysis to model. For example, a logic bomb that writes a value to a file and later reopens the file to retrieve the same value causes symbolic analysis systems to lose track that the values are likely the same. In contrast, the GPT 4o model accurately tracks this connection and can predict the likely outcomes of interactions with the operating system. The model's assumptions about the likely behavior of code works to its advantage in multiple test cases, but assumptions made for other logic bombs can cause the model to find an incorrect result.

A related category to covert propagation is the contextual symbolic value category, where inputs rely on external knowledge to trigger the logic bomb. For example, the required input for "ping_csv" is an IP address that responds to an ICMP echo packet. The GPT 4o model identifies this criteria for the IP address, and correctly states that the default IP address for localhost (127.0.0.1) can be used to handle this logic bomb. Even when GPT 4o does not provide a correct answer, the model still manages to identify the correct criteria the answer must fulfill. For example, in the logic bomb "file_csv", the model correctly identifies that the name of an openable file is needed and responds that the specified filename must be created prior to running the logic bomb.

4.2 Where the LLM falls short

To determine ways to improve the LLM's performance in bypassing evasive code, we categorized the reasons it failed to trigger 27 of the logic bombs by analyzing the step-by-step reasoning present in the model's output. Our results, shown in Table 2, indicate that almost half of the failures are caused by inaccuracies in the generated pseudocode, where values critical to the solution are absent. In these cases, the model assigns arbitrary values to undeclared variables or simply ignores important code. Preliminary tests show that allowing the model to request additional details on missing code and data can address many of these cases. The model also made incorrect assumptions about the behavior of code in 6 cases. For example, while analyzing the logic bomb "2thread_pp_l1", the LLM's response acknowledges that multi-threaded code could change the order of operations, but dismisses this case as "unlikely" before returning an answer the model acknowledges is incorrect. Thankfully, these incorrect assumptions are explicitly acknowledged in the model's response, allowing a human analyst to ask the model to challenge these assumptions in subsequent prompts.

Another implicit assumption made by the model was that the results of Python validation scripts would carry over to the logic bombs written in C and C++. For example, the LLM identified a random number generator seed in Python that seemingly fulfilled the criteria for the logic bomb "rand_ef_l2", but the seed exhibited different behavior when fed to the random number generator in the C standard libraries. In this case, a human analyst can explicitly ask the model to validate using C code. Surprisingly, when we made this request, the model used a Python script to write, compile, and execute C code which found the correct seed.

Lastly, we found that simple logic errors, such as computing the modulus operator incorrectly or returning the result of an intermediate calculation, occurred relatively infrequently. Even so, *no* such errors were observed in the Python scripts the model generated. We plan to investigate whether pushing the model to express calculations through code can reduce the occurrence of these simple logic errors.

4.3 Investigating Real World Evasive Malware

In addition to comparing the GPT 4o model against symbolic execution tools on logic bombs, we also explored real-world instances where AI model can help identify how to bypass evasive checks in malware. To test this task, we use instances of real-world malware that only execute under specific conditions pulled from previous malware analyses we have conducted. The first evasive malware sample we tested[4] stops execution if the operating system is configured with either an English or Russian keyboard layout. After providing the GPT4o model with a function comprised of 1 KB of code containing the check for the keyboard layout, the model successfully identified the code responsible for the evasive tactic and provided the keyboard layouts to avoid. The second evasive malware sample[5] checks to see whether a certain filepath ("C:\\Diebold") exists, as the malware is designed to target Diebold ATMs. By simply passing in the function containing the check, GPT4o identified the file needed to progress.

[4]sha1: ac1eb847a456b851b900f6899a9fd13fd6fbec7d
[5]sha1: 0d484d7adc95caf1b375c30dc949a32bd8b932c1

In both of these examples, the output provided by GPT 4o goes beyond identifying evasive criteria, it also helps human analysts complete essential tasks. After identifying the evasive check, the GPT 4o model provides potential modifications that can be made to the malware or execution environment to bypass the evasive check. While the modifications required to bypass the evasion techniques shown above are seemingly obvious, more sophisticated evasion techniques may require modifying the malware sample or overriding operating system behavior using techniques such as API hooking. In such cases, the model can guide analysts toward the appropriate methods for bypassing the evasive check. Additionally, the model's detailed human-readable explanations are highly beneficial for analysts when writing threat reports, another important task performed by malware analysts.

Unfortunately, applying symbolic analysis in these cases is nontrivial. This is primarily due to the known limitation of symbolic analysis systems in correctly handling calls to external code, such as Windows APIs and system calls. Hence, effective use of symbolic analysis tools requires more than just providing the function containing the evasive tactic.

5 Limitations and Future Work

A major challenge in this study is ensuring that our results generalize beyond this test set, as the publicly available test samples and their answers may be part of the training data for the GPT 4o model. To enhance confidence in the generalizability of our findings, we plan to test the LLM on novel test cases that were not publicly available at the time of its training. Another limitation of our current approach arises when an analyst fails to locate the evasive tactic within the binary. To mitigate this limitation, we plan on measuring the models ability to explore executables and discover evasive techniques on its own. We also aim to understand the impact of different inputs to the LLM's ability to handle evasive code. More specifically, we plan on measuring the impact that these and other tools used by malware analysts, such as debuggers, can have on LLM's reasoning of code.

Moreover, we foresee conducting a user study involving human malware analysts to assess the practical utility and impact of integrating LLMs in their workflow. This is driven by our hypothesis that even when LLMs fail to provide the correct solution, their response containing step-by-step instructions on identifying and bypassing evasion checks can aid human analysts. Furthermore, we believe this can also highlight the capability of human analysts to improve the results of LLMs by pushing back on inappropriate assumptions that are explicitly declared by the model.

6 Conclusion

In this study, we found that LLM-based systems can surpass existing symbolic analysis tools in analyzing certain complex code structures. Moreover, we showcase that even when LLMs fail, the human-readable output of the LLM can still assist human analysts in accomplishing their daily tasks. Finally, we demonstrated cases in real-world malware where LLMs can locate evasive techniques and identify methods to bypass them. These preliminary results show encouraging potential for the use of generative AI to aid human analysts and reduce the impact of malicious software.

References

[1] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy*. IEEE, 633–651.

[2] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*. 51–62.

[3] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. 2022. A Systematical and longitudinal study of evasive behaviors in windows malware. *Computers & security* 113 (2022), 102550.

[4] Maanak Gupta, CharanKumar Akiri, Kshitiz Aryal, Eli Parker, and Lopamudra Praharaj. 2023. From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy. *IEEE Access* (2023).

[5] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. 2011. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*. 285–296.

[6] Kevin Leach, Chad Spensky, Westley Weimer, and Fengwei Zhang. 2016. Towards transparent introspection. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 248–259.

[7] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th annual computer security applications conference*. 386–395.

[8] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. 2021. Longitudinal study of the prevalence of malware evasive techniques. *arXiv preprint arXiv:2112.11289* (2021).

[9] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 757–768.

[10] Anh M Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T King, and Hai D Nguyen. 2009. Mavmm: Lightweight and purpose built vmm for malware analysis. In *2009 Annual Computer Security Applications Conference*. IEEE, 441–450.

[11] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. [n. d.]. X-force: Force-executing binary programs for security applications. In *USENIX Security symposium 2014*.

[12] Paul Royal. 2012. Entrapment: Tricking malware with transparent, scalable malware analysis. *talk at Black Hat* (2012).

[13] Venkata Ramana Saddi, Santhosh Kumar Gopal, Abdul Sajid Mohammed, S Dhanasekaran, and Mahaveer Singh Naruka. 2024. Examine the role of generative AI in enhancing threat intelligence and cyber security measures. In *2024 2nd International Conference on Disruptive Technologies (ICDT)*. IEEE, 537–542.

[14] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[15] Chad Spensky, Hongyi Hu, and Kevin Leach. 2016. LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis.. In *NDSS*.

[16] Fabian Teichmann. 2023. Ransomware attacks in the context of generative artificial intelligence—an experimental study. *International Cybersecurity Law Review* 4, 4 (2023), 399–414.

[17] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. 2006. Automated web patrol with strider honeymonkeys. In *Proceedings of the 2006 Network and Distributed System Security Symposium*. 35–49.

[18] Carsten Willems, Thorsten Holz, and Felix Freiling. 2007. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy* 5, 2 (2007), 32–39.

[19] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 442–458.

[20] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R Lyu. 2018. Benchmarking the capability of symbolic execution tools with logic bombs. *IEEE Transactions on Dependable and Secure Computing* (2018).

[21] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 674–691.

[22] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 227–238.

[23] Yagmur Yigit, William J Buchanan, Madjid G Tehrani, and Leandros Maglaras. 2024. Review of generative ai methods in cybersecurity. *arXiv preprint arXiv:2403.08701* (2024).

[24] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M Blough, Elissa M Redmiles, and Mustaque Ahamad. 2021. An inside look into the practice of malware analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3053–3069.

[25] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. Pmp: Cost-effective forced execution with probabilistic memory pre-planning. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1121–1138.

[26] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. 2013. Spectre: A dependable introspection framework via system management mode. In *2013 43rd Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 1–12.