Characterizing Symbolic Execution Behavior on Evasive Malware

Rachel Soubier, Shahid Ali, Ajay Kumara Makanahalli Annaiah
Department of Computer Science, University of North Carolina, Wilmington, NC, USA
rcs2002@uncw.edu, sal9310@uncw.edu, makanahalliannaiaha@uncw.edu

***Abstract*—Malware analysis is a rapidly growing component of cybersecurity, driven in part by the emergence of evasive malware. The complex methods utilized in modern malicious code, such as obfuscation, environmental checking, and runtime-dependent behavior, hinder static and dynamic analysis. Symbolic execution approaches can theoretically explore all execution paths to locate malicious behavior. However, evasive techniques used by malicious binaries stress the solver, leading to time-outs, incomplete exploration, and the missed detection of key structures. This work introduces SymExE, a symbolic-execution-based framework designed to provide insight into the execution behavior of evasive malware. We conduct the first empirical study of symbolic execution on real evasive malware samples, using key metrics such as code coverage, constraint complexity, and state growth across five sample size groups. Results show no strong relationship between binary size and solver strain, indicating internal code structure and evasive logic are the dominant causes of symbolic execution difficulty. Our findings provide new insight into the limitations of symbolic execution for evasive malware and highlight the importance of considering structural complexity. We further demonstrate that solver progress metrics, such as the number of states explored and the time spent, are unreliable indicators of analysis depth in evasive samples. Our approach establishes a new foundation for understanding symbolic execution of evasive malware, enabling more targeted improvements in solver design and malware analysis methodologies.**

***Index Terms*—malware analysis, symbolic execution, evasive malware, program analysis, binary analysis**

I. Introduction

Malware analysis is one of the most critical areas of cybersecurity, as it provides the key threat intelligence necessary for preventing cyberattacks. Advanced Persistent Threat (APT) groups and other threat actors have developed newer, more complex malware that can evade detection, making both static and dynamic analysis exceedingly difficult and time-consuming. Evasive malware employs techniques such as obfuscation, packing, environmental checking, and running solely in memory [1].

Static analysis, the examination of code without actively running it, is one of the most widely used strategies for understanding malware. Reverse engineering often provides key insight into the structure and functionality of malicious code, and this insight can aid in the prevention of future attacks. However, evasive samples conceal code logic, requiring experts with a deep understanding of systems and advanced reverse engineering experience [2]. In contrast, dynamic analysis enables analysts to observe code running in real-time. Yet, evasive malware that uses environmental checks to determine if it is running in a sandbox can terminate itself, preventing analysis. Further, some malware is fileless or runs entirely in memory, leaving minimal traces and making dynamic analysis efforts less effective [3].

Symbolic execution enables the exploration of multiple binary execution paths simultaneously by treating input variables as symbolic values. Applications of symbolic execution on code with hidden or conditional behaviors has shown moderate success [4]; however, analysis becomes difficult for the solver when executing malware that uses other evasive tactics. One of the biggest challenges of symbolic execution is path explosion, in which an overwhelming number of execution paths are created due to complex branching or obfuscation [5]. This can present several issues for analysis, including timeouts, large computational overhead, and incomplete results. Moreover, most existing symbolic execution frameworks lack the capability to efficiently interpret evasive characteristics to distinguish meaningful execution paths from redundant ones.

To address these challenges, we propose SymExE, a symbolic execution-based framework for systematically analyzing and exploring evasive binaries. Built upon the existing angr [6] platform, SymExE enumerates execution paths and extracts internal execution metrics such as state growth, execution time, and constraint-solving effort. Rather than applying symbolic execution to traditional malware classification or code optimization, we focus on empirically characterizing how evasive malware affects symbolic execution systems. By analyzing path diversity and solver behavior across size-grouped evasive samples, our study provides the first quantitative insights into symbolic execution performance on evasive binaries. The main contributions of this work are summarized as follows:

- The design and implementation of SymExE, a novel framework for systematic exploration of execution traces of various evasive binaries and collection of execution metrics.
- The introduction of solver-based symbolic execution metrics as indicators of evasive complexity.
- The first empirical study on the symbolic execution of solely real evasive malware samples reveals that execution difficulty is driven by structural complexity rather than binary size.

The remainder of this work is organized as follows. Section II discusses our key motivations for our research. Section III reviews recent related work on FL security threats and the

components of our proposed framework. Section IV presents our framework and experimental setup. Section V examines the results of our experiments. Section VI provides discussions and trade-offs of our approach. Finally, Section VII concludes the paper.

II. Motivation

The introduction of complex, evasive malware has made in-depth analysis an increasingly difficult task. Evasive malware is typically designed to conceal execution behavior and to avoid triggering detection. These programs frequently utilize conditional code that only executes under certain conditions or in response to user interaction, achieving this through environmental checks, obfuscation, or other anti-analysis tactics [2]. Traditional static or dynamic analysis techniques are often insufficient for understanding evasive logic, as they may miss alternate paths that do not execute due to their reliance on triggering specific runtime conditions. This limits the ability of analysts to gain a full understanding of code structure and behavior.

Though existing symbolic execution solvers may explore all possible paths, they struggle with the complex logic of evasive malware. Binaries that are large or employ evasion techniques such as obfuscation, packing, environmental checks, and opaque predicates may lead to path explosion. As a result, the system experiences timeouts and incomplete code coverage before reaching the branch of interest [5]. Moreover, systems using symbolic execution for malware detection may miss critical malicious functions or encounter false negatives.

Prior work on symbolic execution has primarily emphasized performance improvements and constraint-solver optimization [7], with techniques like path pruning [8] aimed at reducing redundant exploration; however, these efforts rarely examine the underlying causes of solver failures or how evasive binaries inherently stress symbolic execution engines. The relationship between constraint complexity, evasive behavior, and solver effectiveness remains largely unexplored. To address this gap, our work focuses specifically on evasive binaries and empirically characterizes how their behaviors influence symbolic execution by analyzing state growth, execution time, code coverage, and constraint complexity. This perspective shifts from optimizing execution to understanding how evasiveness shapes it, offering insights that can help malware analysts interpret symbolic execution metrics to assess complexity and evasive characteristics in malicious code.

III. Related Works

A. Evasive Malware

Evasive malware is generally defined as a malicious program that uses one or more strategies to evade detection and analysis [1]. Sandbox detection is the most researched form of evasion, and analysts generally find this technique easier to handle than obfuscation or anti-disassembly tactics [9]. Obfuscation is the process of packing or changing code to make analysis more challenging while preserving its functionality. Moser et al. (2007) proved how simple it is to bypass static analysis techniques with the introduction of their own binary obfuscation scheme [10]. By replacing code blocks with opaque constants, code size increased by an average of 237% on Linux; popular virus scanners McAfee and AntiVir were then unable to identify several malicious programs.

Experts have experimented with new detection strategies. Kirat et al., [11] proposed BareCloud, a system that conducts bare metal analysis to execute malicious software directly on a machine's physical hardware. Their model successfully identified 5,835 evasive samples that would not have been detected in virtualized and sandbox environments, effectively neutralizing sandbox evasion tactics. However, the lack of isolation and environmental control raised security concerns. Feng et al. deployed UBER, a user behavior emulator that creates an abstract profile simulating computer usage patterns, generates artifacts based upon these patterns, and then clones the system into a sandbox [12]. The authors found that their model generated a similar number of artifacts to those of a real machine in 17 categories, reducing sandbox detection.

Argus utilized the Llama model to generate human-readable explained features, correctly identifying 86.66% of fileless malware samples. Their results indicated some success but highlighted the potential for false negatives and dependency on snapshot timing. Nasereddin (2022) addressed process-injecting malware through forensic analysis of live memory, utilizing parallel computing to reduce load [13]. The detection accuracy of this approach was 99.3%, with an average detection time of .052 msec. While their experiment proved successful, it remains uncertain whether the method could continue to maintain high accuracy when faced with malware employing multiple evasion tactics.

B. Symbolic Execution

The goal of symbolic execution is to find constraints on inputs that lead to specific outputs, such as those indicative of malicious behaviors [4]. Longer programs often have an overwhelming amount of possible execution paths, resulting in the path explosion dilemma. Standard symbolic execution of malicious code often encounters path explosion and detection failures, as it must also examine all benign execution paths. Trabish et al. (2018) addressed this challenge by allowing users to specify which parts of the code to ignore, then executing it in fragments [8]. Their model, Chopper, outperformed popular symbolic execution engine KLEE in exploring all but three CVEs, spending no more than 19 minutes of analysis time per program and significantly reducing path explosion. However, this approach requires substantial user interaction, highlighting the potential for machine learning to automate the selection of ignored paths.

Current research on symbolic execution, specifically for evasive malware analysis, has identified several limitations. Obfuscation inherently causes path explosion by increasing branching complexity. Some obfuscators can manipulate or conceal conditional jumps in code, making it difficult to identify certain paths. Banescu et al. (2016) experimented with advanced obfuscation transformations, such as range dividers

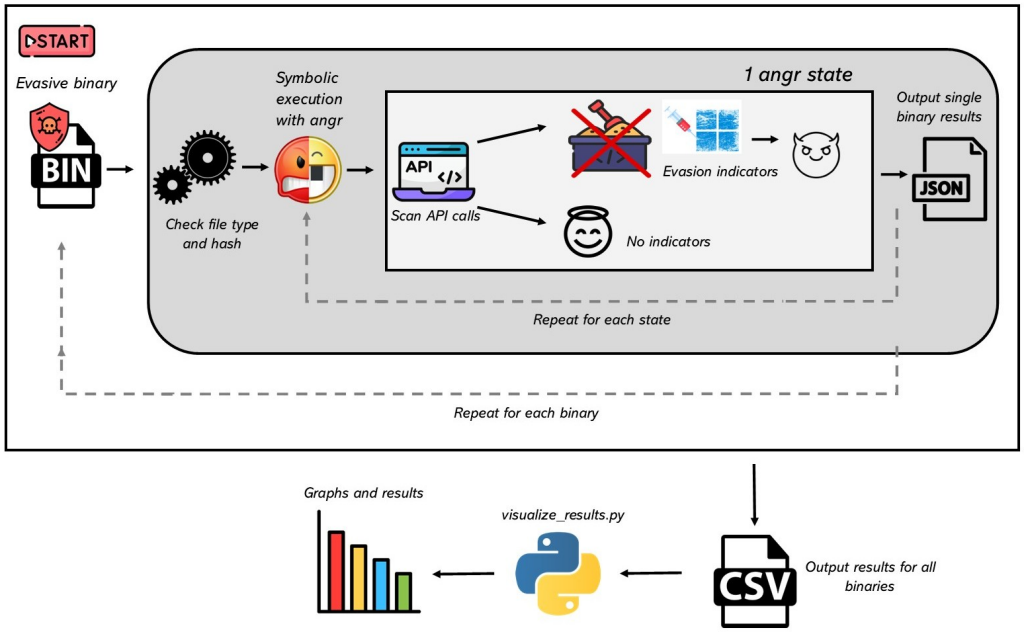

Fig. 1. The system performs symbolic execution on evasive binaries using the angr engine. Each binary is first processed to verify its type and hash, then executed symbolically. Within each state, API calls are analyzed to detect indicators of evasion such as process injection and sandbox checks. Results from each binary are exported as JSON and aggregated across all binaries into a CSV file for visualization. Comparative performance graphs and statistical summaries of execution behavior were generated.

and input invariants [14]. Symbolic execution of the code, incorporating their input invariant strategy, ran for one week before being stopped, showing that symbolic execution for de-obfuscating complex schemes is almost completely ineffective.

One proposed solution to the challenges of obfuscation is taint analysis, a process that marks bits or bytes as tainted if an input is untrusted. Static taint analysis is frequently used in conjunction with symbolic analysis on obfuscated code [15]. Yadegari and Debray (2015) utilized a combination of finer-grained bit-level taint analysis and architecture-aware constraint generation to improve symbolic and concolic execution for obfuscated code [16]. They found their system, ConcoLynx, effectively mitigated over-tainting and path explosion, while also tailoring to x86 architecture for increased accuracy. Their model correctly identified at least one possible input for seven programs obfuscated in four different ways, while tools Vine and S2E failed to do so. ConcoLynx also achieved faster results than existing symbolic execution tools, though the cost was far higher. Their results suggest symbolic and concolic execution require precision and system-specific constraints.

Vouvoutsis et al. (2024) experimented with a hybrid model with machine learning to enhance symbolic execution [3]. They compared execution traces with previously established behavior signatures, and their model achieved a 98% success rate in detecting evasive malware, compared to 78.3% in sandbox environments.

IV. Methodology

A. System Design

The objective of this study was to characterize the effect of evasive malware on symbolic execution tools using derived performance metrics. Fig. 1 illustrates the high-level architecture of the SymExe system. The system utilized angr to symbolically execute a loaded binary and collect metrics. After SymExe was initialized and the binary samples were loaded, each sample was symbolically executed until a timeout or the maximum number of states was reached, or until all states had been explored. Throughout execution, angr's state manager was used to track the active symbolic states and forks, while the path explorer enumerated execution paths. Each state encountered was checked for a series of evasion indicators. This consisted of mostly API calls, with additional checks for other indicators like high entropy. Each evasion indicator was logged, along with the constraints solved, execution time, memory usage, number of states and paths, and code coverage. Once the analysis of each sample was completed, the results were output to a JSON file. Each sample was run individually, and running the full 100 binaries took approximately 60 minutes. Combined results from all binaries were outputted to a CSV file.

B. Experimental Setup & Implementation

We demonstrated the capabilities of SymExE on a workstation with an Intel Core i9-14900 CPU and 48 GiB of RAM. All experimentation was conducted within an isolated Oracle VirtualBox 7.2.4 virtual machine, equipped with 22 CPU cores and 48 GB of virtual RAM, running Kali Linux 2025.3. All

scripting was done in Python 3.13.7. A snapshot of the virtual machine was taken prior to testing to ensure reproducibility.

The main component of SymExE contained 1,121 lines of code that incorporated Angr 9.2.181 [6], along with Claripy and the BackendZ3 [17] constraint solver, as well as the default Angr libraries. For this experiment, the execution of each binary occurred within a virtual environment, utilizing a maximum of 2500 states and a timeout limit of 600 seconds. These configurations can be changed through options in the command line.

C. Datasets

This experiment utilized a curated set of 100 evasive malware samples, collected from VXUnderground [18]. Each sample belonged to a well-known virus family, including Emotet, TrickBot, Zeus, AgentTesla, Qakbot, Dridex, LokiBot, AZORult, ZLoader, and IcedID. Table 1 illustrates the evasion tactics most commonly utilized by these malware families. The samples were all PE32 Windows executables and were grouped into five size-based categories for normalizing visualization: 0-100 KB, 100-200 KB, 200-300 KB, 300-400 KB, and 400-500 KB. Initial trials found that a size limit of 500 KB per sample was necessary to reduce runtime.

TABLE I
EVASION TECHNIQUES IN MALWARE FAMILIES

Family	Evasion Techniques
Emotet [19]	Polymorphism and modular DLLs; environmental checking; obfuscated payloads.
TrickBot [20]	Process hollowing and process injection; obfuscated and encrypted files; anti-debugging.
Zeus [21]	Encrypted C2 traffic; obfuscation and packing
AgentTesla [22]	Obfuscated and packed .NET assemblies; process hollowing.
Qakbot [23]	Polymorphic code; process injection; obfuscation.
Dridex [24]	Encrypted traffic; obfuscation and packing; anti-debugging.
LokiBot [25]	Obfuscation and packing; Process injection and hollowing.
AZORult [26]	XOR encryption/decryption; process hollowing; packing.
ZLoader [27]	Derived from Zeus; includes polymorphic payloads.
IcedID [28]	Masquerading; sandbox evasion; obfuscation; process injection, and process hollowing.

V. RESULTS & EVALUATION

We review the performance of symbolic execution on evasive binaries with SymExe, utilizing a variety of collected metrics. Samples are grouped by size strictly for visualization; our goal is to observe trends influenced by evasive behavior rather than size.

A. Constraints Solved

The sharpest contrast between sample size groups was shown in the number of constraints solved (Fig. 2). While the 200-300 KB size group had an average of 1.8 constraints solved per sample, the 100-200 KB group had 1002.5 per sample. The extreme spike in constraints for small to mid-sized groups indicates the presence of denser constraint chains due to evasive tactics. Additionally, while benign or simplistic malicious samples often show a greater number of constraints solved as size increases, we observed no correlation between size and evasion in these samples. This supports the hypothesis that evasive tactics are the greatest influence on the effectiveness of constraint solving and symbolic complexity.

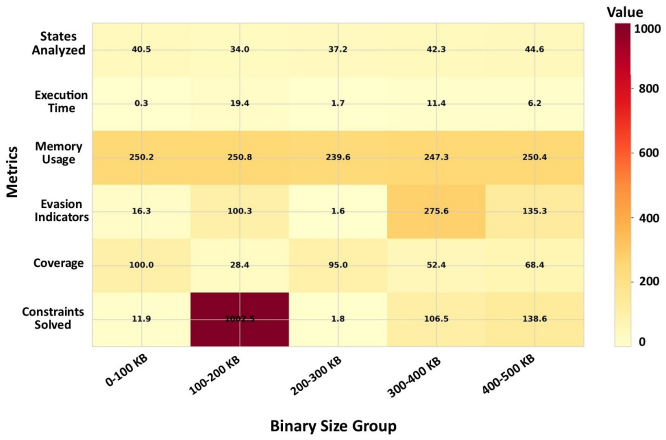

Fig. 2. Heat map of symbolic execution metrics across evasive binaries, grouped by size. Color intensity reflects the difficulty of execution and resource consumption, and illustrates the absence of correlation between size and symbolic complexity.

B. States Explored and Code Coverage

The number of states analyzed by the solver remained relatively stable across size groups; however, the percentage of code covered fluctuated greatly per sample. Higher state exploration did not correlate with higher code coverage. The solver analyzed a group-high average of 44.6 states per sample in the 400-500 KB size group, but was only able to achieve an average of 68.4% code coverage (Fig. 2). These results indicate the solver struggled with exploring meaningful execution paths despite progressing through a high number of states. This behavior is consistent with evasive techniques that mislead the solver and precludes it from easily locating sections of code containing key information on the malware's functionality.

C. Execution Time

Execution time varied dramatically between sample groups, as demonstrated in Fig.2 and Fig.3. Smaller binaries, less than 100 KB, had an average execution time of 0.3 seconds, while the 100-200 KB size group had the highest execution time at 19.4 seconds. Additionally, higher execution time did not correlate with higher code coverage; the solver only covered 28.4% of code in the 100-200 KB group. This suggests the solver spent more time exploring misleading or redundant execution branches introduced by evasive logic, preventing it from reaching key execution regions of the code. The code complexity of evasive behaviors frequently contributes to wasted symbolic effort by the solver.

D. Evasion Indicators

The average number of evasion indicators found across size groups showed no obvious correlation with any other metrics

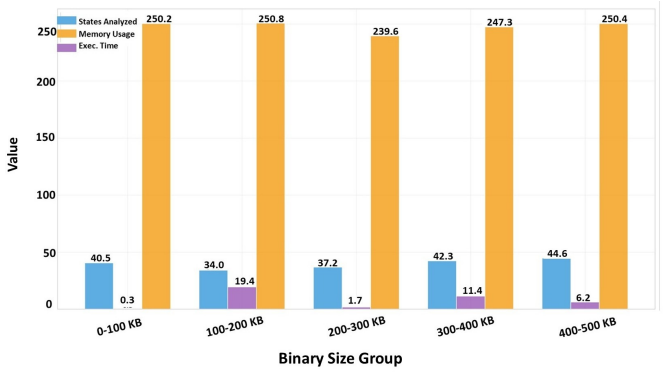

Fig. 3. Key performance metrics provide a comparative view of solver performance (states analyzed, execution time, and memory usage) grouped by binary size. Results highlight trends based on code structure rather than size.

(Fig. 2). A higher percentage of code covered did not indicate the solver was more effective at detecting these indicators; the evasive logic of the malware likely caused it to miss paths of interest during symbolic execution. For instance, the 200-300 KB size group had an average code coverage of 95% but only 1.6 evasion indicators found per sample. These results underscore the importance of not assuming safety based on the presence or absence of reported indicators.

E. Memory Usage

The average memory consumption per sample was consistently high across all size groups, ranging from 239.6 to 250.8 MB (Fig. 3). These findings reinforce the impact of code structure and the complexity of evasive binaries as the primary influences on memory usage during symbolic execution.

VI. Discussion

Overall, these results underscore that the structural complexity of evasive binaries has the greatest impact on the performance of symbolic execution, over factors such as binary size. SymExE struggled to achieve full code coverage, solve constraints, and exhibit high memory usage across all samples. Additionally, our findings did not suggest any correlation between binary size and the effectiveness of the solver. Tactics such as obfuscation, environmental checks, and complex branching drive symbolic execution toward redundant or meaningless paths, further highlighting the importance of understanding code internals in evasive samples. These findings provide the first empirical evidence that symbolic execution strain for evasive malware is a function of evasive behaviors and control-flow complexity.

A. Comparsion

There have been few studies conducted on the symbolic execution of evasive malware, as opposed to the many conducted on simple or benign programs. Moreover, these studies focused on malware classification or solver optimization rather than behavioral analysis. Our study exclusively utilizes real evasive samples, framing these metrics as a quantifiable measure of evasive complexity. Table 2 provides a comparison of previous works to our contribution.

TABLE II
Comparison of Symbolic Execution Studies

Reference	Purpose	Dataset	Focus Metric	Contribution
Trabish et al. (2018)	Path pruning for optimization	Generic, benign code	Path coverage and time	Reduced path explosion
Yadegari & Debray (2015)	Taint-aided concolic exec.	Obfuscated code	Path accuracy	Combined taint + symbolic exec. for precision
Banescu et al. (2016)	SE resistance eval.	Obfuscated binaries	Feasibility rate	Quantified symbolic limitations on obfuscated code
Vouvoutsis et al. (2025)	ML-assisted malware detection	Evasive samples	Classification and detection	Linked symbolic traces to ML classifiers
This work (2025)	**Execution performance on evasive malware**	**100 evasive binaries**	**Constraint, state, path, coverage**	**Solver focused; only evasive samples**

B. Limitations

This work provides novel insight into the symbolic execution of solely evasive binaries, but we encountered several limitations. The study was conducted on a fixed-size dataset of only 100 Windows binaries with a set maximum execution time of 600 seconds. More complex binaries with deeper evasion tactics may not have been fully analyzed as a result. These limitations are a function of the underlying setup, as discussed in section 4.B. Access to a GPU may have allowed for a larger dataset and an increase in allotted execution time. Additionally, SymExE was built upon angr, but other engines that use different search strategies may produce variable results, although evasive behavior patterns are expected to persist regardless. While samples were grouped by size for visualization, different malware families may have different levels of sophistication. Samples of differing sizes from each analyzed family were used, but structural differences in the code may influence symbolic complexity.

C. Future Scope

Future work could involve grouping binaries by the specific evasion tactics they employ. Understanding how symbolic execution performance is affected by the behavior of evasive malware would provide key insights for analysts and may help prevent future cyberattacks. This could be achieved with machine-learning techniques, specifically the use of Large Language Models (LLMs). Integrating LLMs into symbolic execution systems for identifying evasive strategies is one such unexplored avenue. Additionally, the use of LLMs to guide path pruning and prioritization could optimize solver performance, allowing for the complete execution of evasive binaries. By quantifying the impact of complex evasion tactics on solver ability, this work provides a solid baseline for adaptive symbolic execution engines.

VII. Conclusion

This work demonstrates that symbolic execution of evasive malware is entirely dependent on the internal code complexity and evasion tactics enforced by the malware, rather than factors such as binary size. Utilizing a custom-built symbolic execution tool to collect key execution metrics, we provide the first practical study of solver performance and path diversity metrics on strictly evasive binaries. Quantifiable results show evasive malware places a high strain on symbolic execution engines, regardless of size. The solver frequently failed to discover meaningful paths or evasion indicators, despite achieving higher code coverage and exploring more states. To successfully unpack modern evasive malware, analysts must employ targeted symbolic execution strategies to effectively navigate its high path diversity. The tool developed in this work, SymExE [29], has been released as an open-source implementation to support reproducibility and future research.

References

[1] A. Ruggia, D. Nisi, S. Dambra, A. Merlo, D. Balzarotti, and S. Aonzo, "Unmasking the veiled: A comprehensive analysis of android evasive malware," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 383–398. [Online]. Available: https://doi.org/10.1145/3634737.3637658

[2] Q. Le, O. Boydell, B. Mac Namee, and M. Scanlon, "Deep learning at the shallow end: Malware classification for non-domain experts," *Digital Investigation*, vol. 26, pp. S118–S126, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287618302032

[3] V. Vouvoutsis, F. Casino, and C. Patsakis, "Beyond the sandbox: Leveraging symbolic execution for evasive malware classification," *Computers Security*, vol. 149, p. 104193, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740482400498X

[4] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[5] J. Bailey and C. Nicholas, "Symbolic execution in practice: A survey of applications in vulnerability, malware, firmware, and protocol analysis," 2025. [Online]. Available: https://arxiv.org/abs/2508.06643

[6] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 138–157.

[7] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[8] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar, "Chopped symbolic execution," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 350–360. [Online]. Available: https://doi.org/10.1145/3180155.3180251

[9] M. Y. Wong, M. Landen, F. Li, F. Monrose, and M. Ahamad, "Comparing malware evasion theory with practice: Results from interviews with expert analysts," in *Twentieth Symposium on Usable Privacy and Security (SOUPS 2024)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 61–80. [Online]. Available: https://www.usenix.org/conference/soups2024/presentation/yong-wong

[10] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 421–430.

[11] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal analysis-based evasive malware detection," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 287–301. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat

[12] P. Feng, J. Sun, S. Liu, and K. Sun, "Uber: Combating sandbox evasion via user behavior emulators," in *Information and Communications Security*, J. Zhou, X. Luo, Q. Shen, and Z. Xu, Eds. Cham: Springer International Publishing, 2020, pp. 34–50.

[13] M. S. G. Nasereddin, "A new approach for detecting process injection attacks using memory analysis," Ph.D. dissertation, 2022, copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-03-08. [Online]. Available: https://www.proquest.com/dissertations-theses/new-approach-detecting-process-injection-attacks/docview/2771404101/se-2

[14] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 189–200. [Online]. Available: https://doi.org/10.1145/2991079.2991114

[15] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined symbolic taint analysis," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 65–80. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ming

[16] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 732–744. [Online]. Available: https://doi.org/10.1145/2810103.2813663

[17] L. de Moura and N. Bjørner, "Z3: an efficient smt solver," vol. 4963, 04 2008, pp. 337–340.

[18] V.-U. Team, "Vx-underground," https://www.vx-underground.org, 2025.

[19] Cybersecurity and Infrastructure Security Agency. (2018, July) Emotet malware. Alert (TA18-201A). [Online]. Available: https://www.cisa.gov/news-events/alerts/2018/07/20/emotet-malware

[20] ——. (2021, May) Cybersecurity advisory: Aa21-076a. Advisory (AA21-076A). [Online]. Available: https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-076a

[21] A. Hutchings and R. Clayton, "Configuring zeus: A case study of online crime target selection and knowledge transmission," in *2017 APWG Symposium on Electronic Crime Research (eCrime)*, 2017, pp. 33–40.

[22] N. A. for Electronic Certification and C. S. R. of Albania, "Agenttesla malware, technical analysis — version 1.0," Directorate of Cyber Security Analysis, National Authority for Electronic Certification and Cyber Security, Str. "Papa Gjon Pali II" no.3, Tirana, Albania, Tech. Rep., Apr 2024. [Online]. Available: https://aksk.gov.al/wp-content/uploads/2024/05/AgentTesla-Malware-Technical-Analysis-_v1.0.pdf

[23] Cybersecurity and I. S. Agency, "Qbot / qakbot malware," Online slide deck, Oct 2020. [Online]. Available: https://www.cisa.gov/resources-tools/resources/qbotqakbot-malware

[24] M. Corporation, "Dridex (software id s0384)," MITRE ATT&CK Enterprise Software Database, Apr 2025, last modified: 16 April 2025. [Online]. Available: https://attack.mitre.org/software/S0384/

[25] Cybersecurity and Infrastructure Security Agency. (2020, October) Cybersecurity advisory: Aa20-266a. Advisory (AA20-266A). [Online]. Available: https://www.cisa.gov/news-events/cybersecurity-advisories/aa20-266a

[26] M. Corporation, "Azorult (software id s0344)," MITRE ATT&CK Enterprise Software Database, Apr 2025, last modified: 16 April 2025. [Online]. Available: https://attack.mitre.org/software/S0344/

[27] M. T. Intelligence, "Dismantling zloader: How malicious ads led to disabled security tools and ransomware," Microsoft Security Blog, Apr 2022. [Online]. Available: https://www.microsoft.com/en-us/security/blog/2022/04/13/dismantling-zloader-how-malicious-ads-led-to-disabled-security\protect\discretionary{\char\hyphenchar\font}{}{}tools-and-ransomware/

[28] M. Corporation, "Icedid (software id s0483)," MITRE ATT&CK Enterprise Software Database, July 2020, last modified: 22 April 2025. [Online]. Available: https://attack.mitre.org/software/S0483/

[29] R. Soubier, S. Ali, and A. K. Makanahalli Annaiah, "SymExE: Characterizing symbolic execution behavior on evasive malware," https://github.com/cs2researchlab/SymExE, 2025, gitHub repository.