

Evaluation of the effectiveness of static and dynamic methods in malware analysis

Dominik Tracz*, Daniel Sawicki, Konrad Gromaszek

Department of Electronics and Information Technology, Lublin University of Technology, Nadbystrzycka 38A, 20-618 Lublin, Poland

Abstract

This study provides a comparative evaluation of static and dynamic analysis techniques applied to different malware families targeting Windows operating systems. Real-world samples were obtained from the MalwareBazaar portal and included Jigsaw ransomware, the StealC infostealer, and Remcos RAT. The results indicate that static analysis is effective for rapid initial triage of a sample. In-depth static analysis provides a high level of certainty about a sample's capabilities. In certain scenarios, static analysis was clearly insufficient, for example, when the code was protected by a packer, when the configuration was encrypted, or when the relevant stage existed only in memory. In those cases, dynamic analysis techniques, such as memory inspection and memory dumping using a debugger, proved essential for obtaining indicators of compromise. Moreover, dynamic analysis is required to confirm runtime behavior. The reported effectiveness values apply only to the analyzed samples and to the scoring system adopted in this study. The findings demonstrate that hybrid analysis provides the most comprehensive and reliable interpretation of malware behavior within this experimental scope.

Keywords: malware analysis; reverse engineering; static analysis; dynamic analysis; ransomware; infostealer

*Corresponding author

Email address: dominiktr11@gmail.com (D. Tracz)

Published under Creative Common License (CC BY 4.0 Int.)

1. Introduction

Modern malware is rarely just a single executable file whose execution can be described solely by the code visible in a decompiler. Packers, obfuscation, dynamic API resolution, and configuration encryption are widely used. This makes a single analysis method insufficient, as it can lead to incomplete results or overinterpretation.

Static analysis is a technique for examining a file without executing it. It enables a quick overview of the PE format, architecture, compiler, imports, resources, strings, entropy of sections, and the presence of YARA signatures. Its advantages are safety and repeatability, but its effectiveness decreases when the malicious code is packed, encrypted, or exposed only in process memory.

Dynamic analysis is a technique based on running a sample in an isolated environment and observing its behavior. It consists of monitoring files, processes, the registry, and network traffic. It can also include debugging and memory inspection. Its advantage is the ability to confirm the actual behavior of the sample in a given environment. Its main limitation is its dependence on environmental conditions, network configuration, observation time, and the process execution path.

The main research assumption is that the most effective approach is a hybrid one, in which static reconnaissance identifies footholds, dynamic analysis confirms behavior and provides memory artifacts, and subsequent static stages describe the obtained data.

Three research hypotheses were formulated:

- H1. Static analysis is sufficient for a quick overview of the sample and for generating hypotheses, but it does not provide a full description of behavior in

cases of packing, configuration encryption, or a hidden second stage.

- H2. Dynamic analysis provides more confirmed behavioral findings than static analysis, but its results must be interpreted in the context of the execution environment.
- H3. A hybrid approach provides the most complete and reliable results because it connects the description of code structure with the confirmation of runtime behavior.

2. Literature review

Sikorski and Honig in "Practical Malware Analysis" [1] presented a systematic methodology for malicious software analysis, categorizing it into basic and advanced static and dynamic techniques. Building upon this foundation, the analytical process in this study is consolidated into three main stages: basic analysis, advanced static analysis, and advanced dynamic analysis.

Eilam in "Reversing: Secrets of Reverse Engineering" [2] explains in detail techniques of reverse engineering, including disassembling, decompiling, and debugging. The author emphasizes the rising complexity of defense mechanisms used by malware creators such as anti-reversing techniques and debugger detection.

In "A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection" [3], static, dynamic, and hybrid approaches to malware detection based on features obtained from different stages of analysis were presented. The authors indicated that each approach has its advantages and that the results depend on what is being analyzed: code features, runtime behavior, or a combination of both. This supports the assumption

adopted in this study that the effectiveness of the methods should be assessed comparatively.

Monnappa in "Learning Malware Analysis" [4] extends this methodological context by discussing techniques relevant to contemporary malware, such as detecting virtual environments, multi-stage code unpacking, and dynamic API resolution.

In the context of malware analysis automation, the YARA project [5] enables the creation of signature rules based on binary patterns and strings, which links static analysis to detection systems. Tools such as Detect It Easy (DIE) and Exeinfo PE allow rapid identification of the compilation technology and potentially used packers [6].

Research on infostealers indicates the rising popularity of the StealC family, which has been distributed in the Malware-as-a-Service (MaaS) model since 2023 [7,8]. StealC samples are characterized by multi-stage loading and unpacking, configuration encryption, and dynamic API resolution.

In July 2024 Google introduced the App-Bound Encryption (ABE) mechanism in Chrome v127 on Windows, aimed at protecting cookie files and saved passwords from infostealers [9]. This mechanism provides relevant context for the analysis of modern infostealers, but this study does not rely on ABE bypass claims as primary evidence.

Recent research further supports treating malware analysis results as dependent on the tested sample, tooling and execution environment. Afianian et al. [10] survey dynamic-analysis evasion techniques and show that sandbox fingerprinting, stalling, anti-debugging and environment checks can directly affect observed behavior. Maffia et al. [11] provide a large-scale longitudinal study of evasive techniques in Windows malware and report that anti-analysis logic is common across malware families. Alrawi et al. [12] systematize malware sandbox use and show that sandbox selection and configuration can substantially influence observable behavior and reproducibility. Recent studies on adversarial Windows PE malware [13] and malware benchmarks with packer labels [14] also confirm that packing, evasion and behavior-level labels remain important dimensions when interpreting static, dynamic and hybrid results.

3. Materials

Analysis was performed on three samples obtained from the MalwareBazaar malware sample exchange portal [15]. The selected samples represented three different classes of threats: ransomware, a remote access tool (RAT), and an infostealer. The samples were chosen so that each of them represented a different methodological problem: packing, file encryption, encrypted malware configuration, and a multi-stage loader with the second stage present only in memory.

The research environment included an isolated virtual machine with the Windows 10 operating system. Active protection mechanisms that could influence sample execution were disabled, and the network con-

nection was controlled by FakeNet-NG. The system state before and after sample execution was compared, especially the registry and files.

The tools used in static analysis included, among others, Detect It Easy, Exeinfo PE, Strings, YARA, and Ghidra [16]. The tools used in dynamic analysis included x32dbg/x64dbg, dnSpy, Process Monitor, Regshot, System Informer, Wireshark, and FakeNet-NG, while ANY.RUN [17] reports were used as an independent baseline. In the case of StealC, an additional script for decoding strings from the memory dump was prepared, and in the case of Jigsaw, a program for decrypting files encrypted by the ransomware was developed to verify the recovered AES parameters.

To improve reproducibility, the sample hashes, selected indicators of compromise, decoded configuration or string outputs, scoring matrices and offline helper scripts were prepared as supplementary material. The supplementary package does not include executable malware samples, unpacked payloads, full memory dumps or full packet captures.

Table 1 presents the characteristics of each investigated sample.

Table 1: Characteristics of analyzed malware samples

Sample	Type	SHA256	Main analytical barrier
Jigsaw	Ransomware	df049efbfa7ac0b76c8daff5d792c550c7a7a24f6e9e887d01a01013c9caa763	.NET Reactor, lack of access to the actual logic in static analysis
Remcos	RAT	892d9d082a6f846c77d611596da478efb11545709432211a837dce0119d127ca	Encrypted "SET-TINGS" resource, need to distinguish capabilities from executed behavior
StealC	Stealer	fb176719a4819edb677357bdefdc2dd270330395b040933344b50ead500a6a7b	Multi-stage loader, code and configuration hidden behind a memory dump

4. Methodological limitations

Because this study uses a controlled case-study design, its main methodological limitations are stated before the results. The research involved three samples, so the effectiveness values reported later should not be generalized to all malware families or interpreted as general detection rates. The reported percentages are valid only for the analyzed samples, the 16 findings selected for each case study and the adopted P/H/O scoring system.

Other samples, tool versions, sandbox settings, network simulation choices or scoring criteria could produce different values.

An important limitation is the dependence of dynamic analysis on the environment. Conditions such as system language, availability of the C2 server, debugger presence, network isolation, and observation time may influence the execution path. For this reason, sandbox reports, such as ANY.RUN, were used for validation, rather than as a full description of behavior.

Another limitation involves modules not used in the observed execution path. In Remcos and StealC, multiple capabilities were visible, but not all of them were confirmed at runtime. In this research, they were not considered executed behavior because of the lack of evidence from the debugger, system monitoring, network traffic, or an independent sandbox. This distinction is important for interpreting the results in the following sections.

5. Analysis results

Within the limitations described above, the methods of malware analysis were compared using the following simple scale:

- P = 1.0 - proven result
- H = 0.5 - hypothesis or premise
- 0 = 0.0 - no useful result.

Before calculating the percentages, a fixed set of evaluated findings was defined separately for each malware sample. A finding was included when it represented a distinct analytical question relevant to identification, protection or unpacking, configuration recovery, cryptographic parameters, persistence, network behavior, file-system effects, memory-only artifacts or practical indicators of compromise. Duplicate observations were merged, and observations that could not be attributed to the analyzed sample were excluded. For comparability, each case study used 16 evaluated findings.

Each finding was then scored independently for static, dynamic and hybrid analysis. A score of P = 1.0 meant that the method fully confirmed the finding. A score of H = 0.5 meant that the method produced a partial result, hypothesis or indirect indication that still required confirmation. A score of 0 meant that the method did not provide usable evidence for that finding. Hybrid analysis was scored according to the evidence obtained by combining static and dynamic results or by statically examining runtime-derived artifacts such as memory dumps.

Effectiveness was calculated as the sum of points obtained by a method divided by the number of evaluated findings for the given sample and multiplied by 100%. These values are not universal measures of method effectiveness for an entire malware class. They apply only to the analyzed samples, the selected findings and the adopted P/H/0 scoring system, and they serve only to compare methods within this controlled case-study design.

Table 2: Example rows from the scoring matrix

Sample	Finding	Static	Dynamic	Hybrid
Jigsaw	AES-CBC parameters recovered	0	P	P
Remcos	C2 domains and port recovered from the SETTINGS resource	P	P	P
StealC	String key and Base64 to RC4 decoder recovered from the second-stage dump	0	0	P

The interpretation of the example rows is as follows. In the Jigsaw case, static analysis did not recover the AES parameters, while dynamic memory inspection did, so the dynamic and hybrid scores are P. For Remcos, the encrypted SETTINGS resource was decoded statically and then confirmed during execution, therefore all three methods receive P for this finding. For StealC, the string key and decoder were available only after obtaining and statically examining the runtime memory dump, which makes the result hybrid rather than purely static or dynamic. P denotes a fully confirmed finding, H a partially confirmed finding or hypothesis, and 0 no usable evidence.

5.1. Jigsaw ransomware

Static analysis of Jigsaw showed a 32-bit PE (Portable Executable) file with elevated entropy. Exeinfo PE indicated .NET technology and .NET Reactor protection. An attempt at automatic unpacking using de4dot was unsuccessful. Strings and imports revealed: CorBindToRuntimeEx, VirtualAlloc, VirtualFree, IsDebuggerPresent, CreateToolhelp32Snapshot, Module32First, and Module32Next. YARA rules indicated packing, anti-debugging, SEH, and traces of API resolution using CRC32.

The static analysis results were useful at the triage stage but did not provide access to the actual ransomware algorithm logic. It did not allow the encryption algorithm to be reproduced, nor did it allow the key and IV to be obtained. This is an example of a situation where static analysis indicates the right direction for further work but stops before the most important data.

Dynamic analysis confirmed that after execution, the sample displays a false compatibility error message, runs the drpbx.exe process, encrypts user files, adds the .zemblax extension to them, and then displays the ransom note window (as shown in Figure 1). Process Monitor and Regshot confirmed the creation of copies in %userprofile%\AppData\Roaming\Frfr\firefox.exe and %userprofile%\AppData\Local\Drpbx\drpbx.exe, as well as an autostart entry in HKCU\Software\Microsoft\Windows\CurrentVersion\Run.

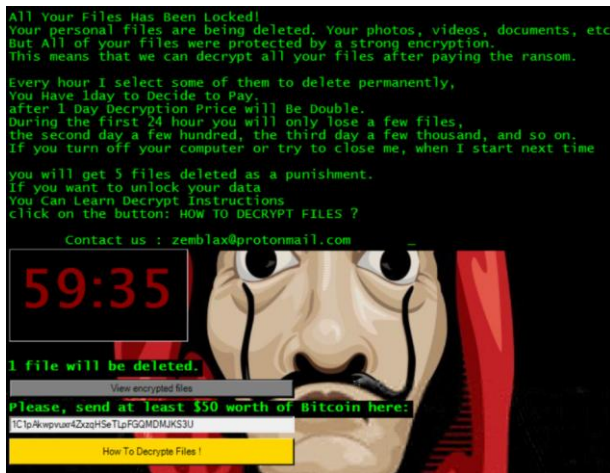


Figure 1: Ransom window

Local Wireshark and FakeNet-NG did not reveal any network traffic originating from the sample. However, the ANY.RUN report flagged the sample with a tag suggesting possible use of Tor. Because no active network communication was confirmed locally, the conclusion about Tor should be treated as a premise, not confirmed behavior.

The most important dynamic finding concerned cryptographic parameters. In dnSpy, empty or almost empty methods were visible, which suggested that parts of the protected code were only available during execution. Setting breakpoints on cryptographic classes of the mscorlib.dll library enabled the capture of encryption parameters from memory. The obtained parameters are presented in Table 3.

Table 3: Cryptographic parameters recovered for Jigsaw

Parameter	Value
Algorithm	AES-CBC
Padding	PKCS7
AES key	3A 82 2C 03 0C 05 DF 67 08 09 0A 0B 0C 0D 0E 0D
IV	00 01 00 03 05 03 00 01 00 00 02 00 06 07 06 00

Based on these data, a decryption program was created that recovered encrypted files with the .zemblax extension. In this case, the practical result, namely a successful decryptor, was possible only because of dynamic analysis and its correlation with prior static findings.

5.2. Remcos RAT

For Remcos, static analysis was more effective than for Jigsaw, because the actual configuration was stored inside the PE file. Detect It Easy confirmed a PE32 GUI file, and imports and strings analysis demonstrated dynamic API resolution, process operations, possible network communication, a keylogger, registry operations, and possible process injection techniques.

The most important artifact was the "SETTINGS" resource of type RT_RCDATA. Analysis of the structure of this resource showed the following format: [key_len][rc4_key][encrypted_config].

The first byte specifies the length of the RC4 key, the following bytes contain the key, and the rest is the encrypted configuration. Reconstruction of the RC4 algorithm and the use of a custom decoder allowed for recovery of the C2 domain, the network communication port 29810, the TLS flag, the campaign identifier "F-35 JET", the name of the file to which keylogger data is saved "koehkss.dat", which is presented in Figure 2, and the identifier "kohuiests-W6QL9U". Dynamic analysis confirmed that the values from the configuration were not merely unused configuration artifacts. A breakpoint on getaddrinfo showed the transfer of the domain and port to WinSock, and FakeNet-NG registered DNS requests and TCP connections to a controlled address on port 29810. The payload was readable in plaintext, which was consistent with the TLS flag being set to 0.

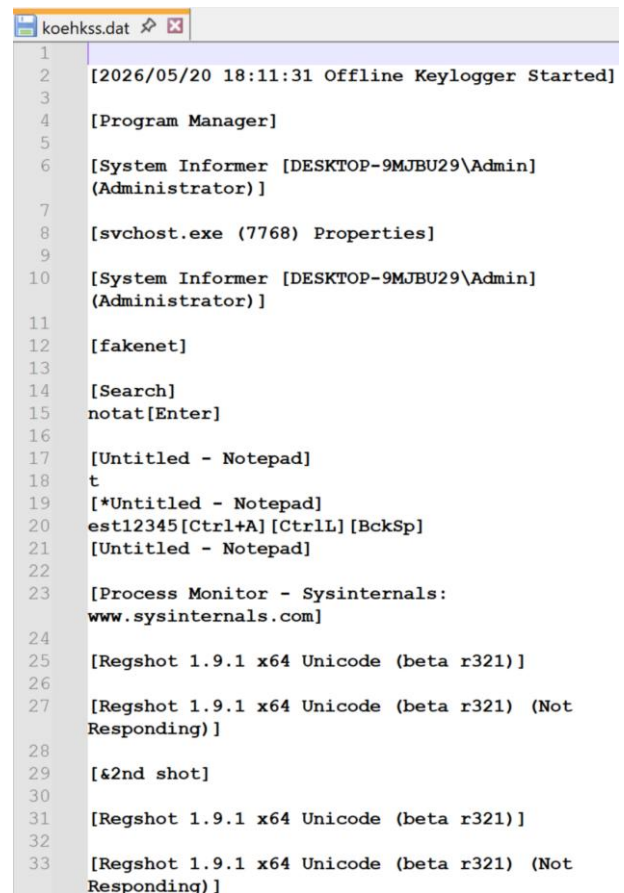


Figure 2: Contents of koehkss.dat file

Another important dynamic result was the confirmation of the keylogger module. In x32dbg, SetWindowHookExA was observed with the WH_KEYBOARD_LL value, followed by CreateFileW and WriteFile for the file C:\Users\Admin\AppData\Roaming\koehkss.dat. Regshot confirmed the creation of this file and the following registry key: HKCU\Software\kohuiests-W6QL9U.

At the same time, dynamic analysis narrowed down some hypotheses raised by static analysis. Imports and strings suggested process injection, UAC bypass, and classic autorun persistence. In the examined run, Procmon (Process Monitor) and Regshot did not con-

firm any Run or RunOnce entries, services, scheduled tasks, or any unambiguous injection into svchost.exe. The conclusion is methodologically important: the presence of a capability in code or strings does not mean execution of that function in a given environment.

5.3. StealC infostealer

StealC turned out to be the most complex case in terms of the analysis flow. Basic static analysis indicated a PE64 GUI file compiled with MSVC/Visual Studio 2022, metadata suggesting impersonation of Realtek/NVIDIA drivers, and a large low-entropy overlay resembling file pumping. RT_RCDATA resources had an unusual format and constituted the main foothold for further analysis.

Advanced analysis in Ghidra showed that WinMain creates a hidden window, sets a timer, and transfers execution to the loader logic only after a WM_TIMER message. Then, the sample removes separators from resources, performs Base64 and custom Base85 decoding, applies an AES-CTR-like transformation based on CNG/ECB and 32-byte XOR, decompresses the result, and maps the second stage as a PE image in memory. Figure 3 shows the contents of the WinMain function.

```

1
2 undefined8 main(HINSTANCE param_1)
3
4 {
5     int iVar1;
6     HWND hWnd;
7     tagMSG local_88;
8     WNDCLASSEX local_58;
9
10    FUN_140002fe0();
11    local_58.cbSize = 0x50;
12    local_58.style = 0;
13    local_58.cbClsExtra = 0;
14    local_58.cbWndExtra = 0;
15    local_58.lpfnWndProc = FUN_140003010;
16    local_58.hIconSm = (HICON)0x0;
17    local_58.lpszClassName = "cls_8a15337f";
18    local_58.hIcon = (HICON)0x0;
19    local_58.hCursor = (HCURSOR)0x0;
20    local_58.hbrBackground = (HBRUSH)0x0;
21    local_58.lpszMenuName = (LPCSTR)0x0;
22    local_58.hInstance = param_1;
23    RegisterClassExA(&local_58);
24    hWnd = CreateWindowExA(0, "cls_8a15337f", "", 0, 0, 0, 1, 1,
25                          (HWND)0x0, (HMENU)0x0, param_1,
26                          (LPVOID)0x0);
27    if (hWnd != (HWND)0x0) {
28        SetTimer(hWnd, 0x70a, 0xe2, (TIMERPROC)0x0);
29        iVar1 = GetMessageA(&local_88, (HWND)0x0, 0, 0);
30        while (0 < iVar1) {
31            TranslateMessage(&local_88);
32            DispatchMessageA(&local_88);
33            iVar1 = GetMessageA(&local_88, (HWND)0x0, 0, 0);
34        }
35        return 0;
36    }
37    FUN_1400017c0();
38    return 0;
39}

```

Figure 3: Contents of the WinMain function.

Dynamic analysis performed in x64dbg confirmed resource usage, memory allocation, buffer processing, the appearance of the MZ header in a memory section,

the setting of section protections with VirtualProtect, flushing of instruction cache, and the call to RtlAddFunctionTable. Ultimately, it was possible to dump a valid second-stage PE64 payload. This dump became a separate analytical artifact: it was not available during the basic static analysis of the original file, so further analysis of this artifact is classified as a hybrid analysis result.

The second stage contained a much richer set of strings and code, including WinINet libraries, JSON, Chromium/DPAPI artifacts, Firefox/Gecko, Outlook, Steam, WinSCP, and Foxmail. A crucial step was the reconstruction of the dedicated string-hiding mechanism in the second stage. In Ghidra, the plaintext key "TPSBPAQGEWvRVuTfBU" was identified, and the decoding functions corresponded to the Base64 -> RC4 scheme. A custom decoding script allowed for the recovery of the C2 and the list of modules, as shown in Figure 4.

1	offset	score	encoded	decoded
2	0x000074ff	1.000	WAVAWH	EU2T
3	0x00007e9f	1.000	UAVAWH	MU2T
4	0x0001434b	1.000	UAUAWH	MURt
5	0x00055b55	1.000	fd94Pu	ao
6	2			
7	0x00055df9	1.000	fd94Ku	ao
8	&			
9	0x000656d1	1.000	USVAWH	Lu2X
10	0x0006a58e	1.000	fd94Ju	ao
11	*			
12	0x0006a707	1.000	fb94Ou	ao
13	6			
14	0x00081220	1.000	K2BGNbDcgXb/ohkmzfE+SCa78+Q=	
15	604938ee81277e917dea			
16	0x00081260	1.000	djUAYuaIlyHp90d9	kernel32.dll
17	0x00081278	1.000	fdQEbFONlyHp90d9	advapi32.dll
18	0x00081290	1.000	WjUGXPGHh1Kj9110ieC=	
19	GetProcAddress			
20	0x000812a8	1.000	UT8TaM+NhmGm4VJQ	LoadLibraryA
21	0x000812c0	1.000	WCgbeNOWi3Ci4Fg=	ExitProcess
22	0x000812d8	1.000	Xjwdf+ashX2j/04=	CloseHandle
23	0x000812f0	1.000	WjUGWfCB1le19UpklubLGh+438E=	
24	GetUserDefaultLangID			
25	0x00081310	1.000	UiAXYsaSgX2zxXa==	OpenEventW
26	0x00081328	1.000	XiIXbfeBoWwi/V9G	CreateEventW
27	0x00081340	1.000	TjwXafM=	Sleep
28	0x00081350	1.000	WjUGT+yJlGaz911fm/liLg==	
29	GetComputerNameW			
30	0x00081370	1.000	WjUGWfCB112m/k5G	GetUserNameW
31	0x00081388	1.000	dsQGfLnLyyryvRmk1KY0QT/r	
32	http://95.85.238.4			
33	0x000813a8	1.000	MjVLNeDcOCT39xh0mPEWtyjp87Fj0AUuw==	
34	/e99c8470d3ebe7696e4a.php			

Figure 4: Results of custom script execution - decoded IoCs.

The most important StealC artifacts are summarized in Table 4.

Table 4: Selected StealC artifacts

Category	Value
String key	TPSBPAQGEWvRVuTfBU
String algorithm	Base64 -> RC4
Campaign identifier	604938ee81277e917dea
C2	hxxp://95[.]85[.]238[.]4
Endpoint	/e99c8470d3ebe7696e4a.php
HTTP method	POST
Targets	Chromium, cookies, logins, history, webdata, Steam, Outlook, WinSCP, Foxmail

The local virtual machine did not provide evidence of full data exfiltration or permanent persistence. The ANY.RUN report for the same sample confirmed classification as StealC and an HTTP POST request to 95[.]85[.]238[.]4:80 with HTTP response code 200. The difference between the local run and ANY.RUN probably results from environmental conditions and from the fact that the local virtual machine used FakeNet-NG, which means it did not have real contact with the actual C2.

This case illustrates the iterative nature of hybrid analysis: static analysis demonstrated the loader and resources, dynamic analysis provided a dump of the second stage, and only then did the second static analysis of the dump allow for the recovery of IoCs.

6. Comparison of method effectiveness

Table 5 shows a synthetic comparison of the relative effectiveness of the methods in the three case studies.

Table 5: Synthetic effectiveness of methods in the examined samples

Sample	Static	Dynamic	Hybrid
Jigsaw	31.3%	75.0%	100.0%
Remcos	71.9%	75.0%	90.6%
StealC	43.8%	68.8%	96.9%
Average	49.0%	72.9%	95.8%

Table 6 presents the number of evaluated findings and the total point scores from which the percentage values were derived. The detailed matrices are available as supplementary material, while Table 2 provides representative rows showing how individual findings were scored. Consequently, the percentages in Table 5 should be read as case-study scores derived from the selected findings, not as general detection rates or method benchmarks for the malware families.

Table 6: Basis for calculating method effectiveness

Sample	Number of evaluated findings	Static score	Dynamic score	Hybrid score
Jigsaw	16	5.0	12.0	16.0
Remcos	16	11.5	12.0	14.5
StealC	16	7.0	11.0	15.5

The results confirm hypothesis H1. Static analysis was useful in every case, but its scope depended on the sample's architecture. For Jigsaw, it stopped at the .NET Reactor protection layer. For Remcos, it was comparatively effective because the configuration was stored inside a resource that could be reconstructed. For StealC, it described the loader well but did not allow the recovery of the full second-stage configuration from the original file.

Hypothesis H2 was partly confirmed. Dynamic analysis provided more proven results than static analysis, especially in terms of processes, files, the registry, network communication, and data from memory. At the same time, StealC showed the limitations of dynamic analysis: the absence of observed network traffic did not mean the absence of networking functionality, because

ANY.RUN registered an HTTP POST in a different environment.

However, this does not mean simply combining every observation. The value of the hybrid approach lies in verification: static findings indicate what should be checked at runtime, while dynamic traces show which static hypotheses are actually relevant.

7. Research conclusions

Based on the conducted analysis, the following conclusions were formulated:

1. Static analysis is the best method to start an investigation, but it should not be treated as proof of a particular sample's behavior. In Remcos, it allowed for configuration retrieval, but for StealC and Jigsaw, it did not allow the most important data to be obtained without dynamic analysis.
2. Dynamic analysis is necessary to confirm how malware works in the system and what its effects are. This kind of analysis revealed file encryption and persistence in Jigsaw, the writing of keylogger data to a file in Remcos, and the preparation of a PE image in the second stage of StealC.
3. A negative result in dynamic analysis does not always mean the absence of a function. The lack of StealC network traffic in the local VM was an effect of the particular environment and did not rule out C2 communication, which was proven by the independent sandbox.
4. Correlation of findings provides the most reliable results. Static retrieval of Remcos C2 was validated by getaddrinfo, FakeNet, Wireshark, and the plaintext beacon. The StealC dump obtained using x64dbg revealed the RC4 key and full configuration only after static analysis in Ghidra.
5. The results should distinguish between proven behavior, capability in code, and analytical hypothesis. Without this distinction, it is easy to incorrectly treat strings or imports as actual module execution.
6. The hybrid approach requires the greatest amount of work, but it provides the greatest research and practical value. In this research, it allowed not only sample classification, but also the restoration of cryptographic parameters, preparation of a decryptor, C2 confirmation, and the construction of reliable IoCs.

8. Summary

The conducted research confirmed that malware analysis should be iterative and hybrid. Static analysis provides quick reconnaissance and indicates footholds. Dynamic analysis confirms the actual effects of execution, allows data retrieval from process memory and verifies hypotheses. Only the combination of these methods provided a sufficiently complete picture of the samples' behavior.

The main practical conclusion is that the effectiveness of analysis depends not on choosing the better method, but on appropriately combining methods and clearly indicating the source of each finding. In the analyzed cases, the average effectiveness of hybrid

analysis was 95.8%, while dynamic analysis reached 72.9% and static analysis 49.0%.

References

- [1] M. Sikorski, A. Honig, Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software, No Starch Press, San Francisco, 2012.
- [2] E. Eilam, Reversing: Secrets of Reverse Engineering, Wiley, Indianapolis, 2005.
- [3] A. Damodaran, F. Di Troia, V. A. Corrado, T. H. Austin, M. Stamp, A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection, Journal of Computer Virology and Hacking Techniques 13 (2017) 1-12, <https://doi.org/10.1007/s11416-015-0261-z>.
- [4] K. Monnappa, Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware, Packt Publishing, Birmingham, 2018.
- [5] V. M. Alvarez, YARA: The pattern matching swiss knife, <https://virustotal.github.io/yara/>, [12.05.2026].
- [6] horsicq, Detect It Easy: packer and compiler detection tool, <https://github.com/horsicq/Detect-It-Easy>, [18.05.2026].
- [7] Sekoia TDR, StealC: a copycat of Vidar and Raccoon info stealers gaining in popularity - Part 1, <https://blog.sekoia.io/stealc-a-copycat-of-vidar-and-raccoon-info-stealers-gaining-in-popularity-part-1/>, [02.06.2026].
- [8] Sekoia TDR, StealC: a copycat of Vidar and Raccoon info stealers gaining in popularity - Part 2, <https://blog.sekoia.io/stealc-a-copycat-of-vidar-and-raccoon-info-stealers-gaining-in-popularity-part-2/>, [02.06.2026].
- [9] W. Harris, Improving the security of Chrome cookies on Windows, Google Security Blog, <https://security.googleblog.com/2024/07/improving-security-of-chrome-cookies-on.html>, [03.06.2026].
- [10] A. Afianian, S. Niksefat, B. Sadeghiyan, D. Baptiste, Malware Dynamic Analysis Evasion Techniques: A Survey, ACM Computing Surveys 52(6) (2019) 1-28, <https://doi.org/10.1145/3365001>.
- [11] L. Maffia, D. Nisi, P. Kotzias, G. Lagorio, S. Aonzo, D. Balzarotti, Longitudinal Study of the Prevalence of Malware Evasive Techniques, arXiv preprint arXiv:2112.11289 (2021), <https://doi.org/10.48550/arXiv.2112.11289>.
- [12] O. Alrawi, M. Y. Wong, A. Avgetidis, K. Valakuzhy, B. V. Adjibi, K. Karakatsanis, M. Ahamad, D. Blough, F. Monrose, M. Antonakakis, SoK: An Essential Guide For Using Malware Sandboxes In Security Applications: Challenges, Pitfalls, and Lessons Learned, arXiv preprint arXiv:2403.16304 (2024), <https://doi.org/10.48550/arXiv.2403.16304>.
- [13] K. Aryal, M. Gupta, M. Abdelsalam, M. Saleh, Intra-section code cave injection for adversarial evasion attacks on windows PE malware file, Computers & Security 159 (2025) 104690, <https://doi.org/10.1016/j.cose.2025.104690>.
- [14] R. J. Joyce, E. Raff, C. Nicholas, J. Holt, MalDICT: Benchmark Datasets on Malware Behaviors, Platforms, Exploitation, and Packers, arXiv preprint arXiv:2310.11706 (2023), <https://doi.org/10.48550/arXiv.2310.11706>.
- [15] abuse.ch, MalwareBazaar: malware sample exchange portal, <https://bazaar.abuse.ch/>, [29.05.2026].
- [16] National Security Agency, Ghidra Software Reverse Engineering Framework, <https://ghidra-sre.org/>, [05.06.2026].
- [17] ANY.RUN, Interactive Online Malware Sandbox, <https://any.run/>, [29.05.2026].