Journal of *Cybersecurity and Privacy*

MDPI

Article

Automating the Detection of Evasive Windows Malware: An Evaluated YARA Rule Library for Anti-VM and Anti-Sandbox Techniques

Sebastien Kanj *, Gorka Vila and Josep Pegueroles

Department of Telematics, Universitat Politècnica de Catalunya, Jordi Girona 31, 08034 Barcelona, Catalonia, Spain; gorka.vila@estudiantat.upc.edu (G.V.)
* Correspondence: sebastien.kanj@upc.edu

Abstract

Anti-analysis techniques, also known as evasive techniques, enable Windows malware to detect and evade dynamic inspection environments, undermining the effectiveness of virtual-machine and sandbox-based inspection. Despite extensive prior research, no unified classification has been paired with a large-scale empirical evaluation of static detection capabilities for these behaviors. This paper addresses this gap by presenting a comprehensive classification and detection framework. We consolidate 94 anti-analysis techniques from academic, community, and threat-intelligence sources into nine mechanistic categories and derive corresponding YARA rules for static identification. In total, 82 YARA signatures were authored or refined and evaluated on 459,508 malware and 92,508 goodware samples. After iterative refinement using precision thresholds, 42 rules achieved high accuracy ($\geq$75%), 16 showed moderate precision (50–75%), and 24 were discarded due to unreliability. The results indicate strong static detectability for firmware- and BIOS-based checks, but limited precision for timing-based evasions, which frequently overlap with benign behavior. Although YARA provides broad coverage of observable artifacts, its static nature limits detection under obfuscation or runtime mutation; our measurements therefore represent conservative estimates of technique prevalence. All validated rules are released in an open-source repository to support reproducibility, improve incident-response workflows, and strengthen prevention and mitigation against real-world threats. Future work will explore hybrid validation, container-evasion extensions, and forensic attribution based on signature co-occurrence patterns.

Keywords: malware; anti-analysis; YARA; sandbox; virtual machine; goodware

Academic Editors: Feng Wang and Yongning Tang

Received: 22 February 2026
Revised: 31 March 2026
Accepted: 3 April 2026
Published: 8 April 2026

1. Introduction

Virtual machines (VMs) and sandboxes constitute foundational components in contemporary malware analysis and threat research. Virtualization enables analysts to execute untrusted binaries in isolated environments, while sandboxes provide controlled execution contexts that often combine virtualization, API interception, and behavioral logging. These techniques allow analysts to capture runtime interactions such as file system access, registry manipulation, and network communication without compromising production systems [1]. Together, these techniques form the core of automated dynamic analysis infrastructures deployed in both academic research and enterprise security products [2].

However, modern Windows malware families increasingly incorporate anti-analysis mechanisms to evade such monitoring. Early evasion checks targeted recognizable artifacts

such as driver names or registry entries, but current samples employ complex detection strategies including CPU and timing probes, hardware fingerprinting, API and process hook enumeration, undocumented system-call interrogation, and even human-interaction or firmware-level side channels [3]. These behaviors can suppress or modify payload execution, effectively degrading dynamic analysis coverage and impeding threat-intelligence pipelines [4].

Static signature frameworks such as YARA [5] offer a complementary detection layer by enabling the identification of characteristic byte or string patterns in binaries without execution [6]. A YARA rule consists of metadata, pattern definitions, and Boolean conditions that collectively detect distinctive code fragments associated with specific behaviors or families [7]. Nevertheless, balancing rule coverage against false positives remains a non-trivial task [8], especially when targeting evasive behaviors that may vary across implementations.

To guide this study, we formulate the following research questions:

- RQ1: To what extent can anti-analysis techniques in Windows malware be reliably detected using static YARA signatures?
- RQ2: Which classes of evasion techniques are most amenable to static detection, and which remain difficult due to implementation variability or benign overlap?
- RQ3: How does large-scale empirical validation refine rule reliability and reveal limits of static detection?

In this work, we focus on automating the detection of evasive behaviors in Windows malware through a comprehensive, evaluated library of YARA rules targeting anti-VM and anti-sandbox techniques, as Windows remains the most widely targeted and virtualized operating system and constitutes the primary execution environment for sandbox-based malware analysis platforms. Our contributions are fourfold. First, we conducted an exhaustive review of the literature and Cyber Threat Intelligence (CTI) sources, spanning 2014–2025, to identify all publicly known anti-VM and anti-sandbox techniques employed in Windows malware. Second, for each technique, we developed or refined a corresponding YARA rule capable of statically detecting indicative code or string artifacts. Third, we systematically evaluated each rule across two large corpora—459,508 malware samples from the Bazaar repository [9] and 92,508 benign Windows executables from the Assemblage datasets [10]—iteratively refining the rules to minimize false positives and maximize precision. Finally, we release the resulting ruleset as an open-source library, providing the research community with an empirically validated, automation-ready resource for evasion detection. Beyond producing a rule library, our study also offers an empirical assessment of which classes of anti-analysis techniques are inherently compatible with static detection and which remain fundamentally constrained by implementation variability, thereby providing guidance for future hybrid detection research.

Our framework aims to bridge the gap between conceptual surveys of evasion mechanisms and their practical, rule-based detection. Beyond describing known techniques, our work emphasizes the reproducible creation, empirical validation, and open dissemination of YARA-based detectors that can be readily integrated into existing malware analysis pipelines and security products. To our knowledge, this study provides the first cross-technique, empirically validated YARA rule library specifically targeting Windows anti-analysis behaviors and evaluated at this scale. The full rule set is publicly available at https://gist.github.com/sebastienkanj-upc/d514bc23c6d10cb55c692759fcc5226e (accessed on 22 February 2026), enabling reproducibility and further extension by the research community. By integrating systematic technique identification, reproducible rule construction, and large-scale validation across both malware and goodware corpora, the framework supports automated anti-evasion modules in sandboxes and endpoint protection systems while strengthening the link between conceptual taxonomies and operational static detection.

The remainder of this paper is organized as follows. Section 2 reviews related studies on anti-analysis strategies and static detection techniques. Section 3 details our methodology for collecting evasion techniques, constructing the classification framework, and developing and evaluating YARA rules. Section 4 presents the identification and characterization of anti-VM and anti-sandbox techniques organized into nine mechanistic categories, defined as groups of techniques sharing similar underlying operational principles (e.g., timing manipulation, hardware fingerprinting, or artifact-based detection). Section 5 describes the development, refinement, and consolidation of the YARA rule library. Section 6 discusses the empirical evaluation outcomes, including detection metrics, category-level observations, and temporal trends. Finally, Section 7 summarizes our findings and outlines directions for future work, including hybrid detection approaches and expanded evasion coverage.

2. Related Work

Research on anti-analysis mechanisms in Windows malware can be broadly divided into two complementary domains: (1) studies that document, categorize, or empirically characterize evasion techniques, and (2) works that investigate static or dynamic detection methods, including the use of YARA or hybrid approaches to capture such behaviors.

2.1. Evasion and Anti-Analysis Studies

The study of malware evasive behavior has evolved from early surveys to large-scale empirical analyses and systematic taxonomies. Early foundational works, such as Jadhav et al. [11] and Biondi et al. [12], provided broad overviews of evasive malware traits and discussed how these mechanisms challenge traditional analysis pipelines. These surveys highlighted the arms race between malware authors and defenders but did not establish fine-grained classifications or derive actionable detection signatures.

A major milestone in systematizing evasion techniques is the survey by Afianian et al. [13], which presents a comprehensive classification of dynamic-analysis evasion strategies and distinguishes between detection-dependent techniques (e.g., fingerprinting and debugger detection) and detection-independent approaches such as stalling or environment-triggered execution. Their work also emphasizes how sandbox evasion remains the dominant focus of modern malware design and highlights the limitations of current defensive transparency mechanisms. While highly influential, the study remains primarily conceptual and does not provide empirically validated detection artifacts.

More structured empirical systematizations emerged in recent years. Galloro et al. [14] conducted one of the most comprehensive longitudinal analyses of evasive Windows malware, collecting and categorizing 92 distinct techniques and evaluating them across more than 45,000 samples observed between 2010 and 2019. Their study shows that while the overall proportion of evasive malware has remained relatively stable, the specific techniques used have shifted significantly over time, confirming the adaptive nature of attacker strategies. Although this work provides a strong empirical taxonomy and statistical characterization, it does not translate these observations into reusable rule-based detection mechanisms.

Additional works further deepen the technical understanding of evasive strategies. D'Elia et al. [15] and Filho et al. [16] document low-level anti-instrumentation and anti-DBI behaviors, while Gavrila and Zacharis [17] examine long-term trends and forecast the emergence of AI-assisted evasion. Mills and Legg [18] provide complementary insights from an experimental perspective, showing how automated sandbox reconfiguration can expose hidden execution triggers and revealing that modern malware frequently combines multiple context-aware checks, reverse Turing tests, and environment-dependent activation

logic. Despite these advances, such studies typically focus on behavioral understanding rather than producing portable detection signatures.

Empirical measurements of real-world prevalence have also been explored. Botacin and Uebel [19] analyzed more than 100,000 binaries and reported widespread use of environment-aware checks. Kim et al. [20] extended this to hundreds of thousands of Windows samples, confirming persistent adoption of timing-based and API-level evasion. Chen et al. [21] compared targeted and generic threats, showing that sophisticated families often deploy multiple layered evasion strategies simultaneously. Collectively, these works demonstrate the maturity of evasion techniques but also reveal a recurring limitation: most studies remain descriptive and do not yield reusable static detection rules or standardized evaluation frameworks.

2.2. Static Detection and YARA-Based Methods

Parallel research has explored static pattern matching and rule-driven detection frameworks. In this context, YARA has emerged as a widely adopted standard for expressing human-readable detection rules in malware analysis. YARA is a rule-based framework that allows analysts to define patterns based on textual strings, binary sequences, and structural conditions, enabling the identification and classification of files according to their content.

Prior studies have demonstrated both the effectiveness and limitations of this approach. Mahdi et al. [6] showed that carefully crafted YARA rules can provide efficient and accurate binary classification, while Naik et al. [7] highlighted that automatically generated rules often require substantial manual refinement to reduce false positives. However, most existing rule sets are primarily designed for malware family identification or indicator-of-compromise detection, rather than explicitly targeting anti-analysis logic.

Recent work has also explored how YARA can be integrated into operational cybersecurity workflows. Mirza et al. [22] illustrate the practical value of YARA-based scanning within real-world defensive pipelines by developing a visualization-driven system that processes rule matches and IOC detections from large-scale scans. Their work highlights the operational importance of interpretable rule outputs and reinforces the role of YARA as a bridge between detection engines and analyst decision-making. Nevertheless, the study focuses on workflow integration and visualization rather than systematic construction or validation of rule sets for specific evasion behaviors.

Hybrid detection pipelines have also been proposed. Dai Nguyen et al. [23] introduced a framework that forces execution past anti-emulation checks, enabling YARA matching on memory artifacts, while Bhattacharjee et al. [24] integrated rule-based filtering into a multi-stage classifier to improve overall detection performance. Behavioral approaches such as Fukushima et al. [25] further stress the importance of contextual constraints and structural reasoning to mitigate false positives, principles that are reflected in our rule construction methodology through section filtering, conditional clauses, and opcode-level matching.

Taken together, prior efforts provide valuable conceptual, empirical, and methodological foundations for understanding malware evasion. However, several gaps remain evident: (i) taxonomies and empirical studies rarely provide actionable reusable signatures; (ii) YARA-based research focuses primarily on family identification or IOC detection rather than explicitly modeling anti-analysis logic; (iii) no prior work offers a unified and empirically validated YARA library spanning the breadth of anti-VM and anti-sandbox techniques documented in the literature; and (iv) large-scale precision-driven evaluation against both malware and goodware datasets remains largely absent.

These gaps motivate the central contribution of this work: the construction, refinement, and validation of a comprehensive YARA rule set for detecting Windows anti-analysis

techniques, grounded in systematic classification and evaluated at scale to support reproducibility, operational integration, and improved defensive capabilities.

3. Methodology

Our methodology follows eight structured stages, from the systematic discovery of evasion techniques to the assembly and validation of a publicly available YARA rule library. Each step is described below, and Figure 1 provides a schematic overview of the full workflow. Step numbers in the figure correspond to the subsection numbering in this section.

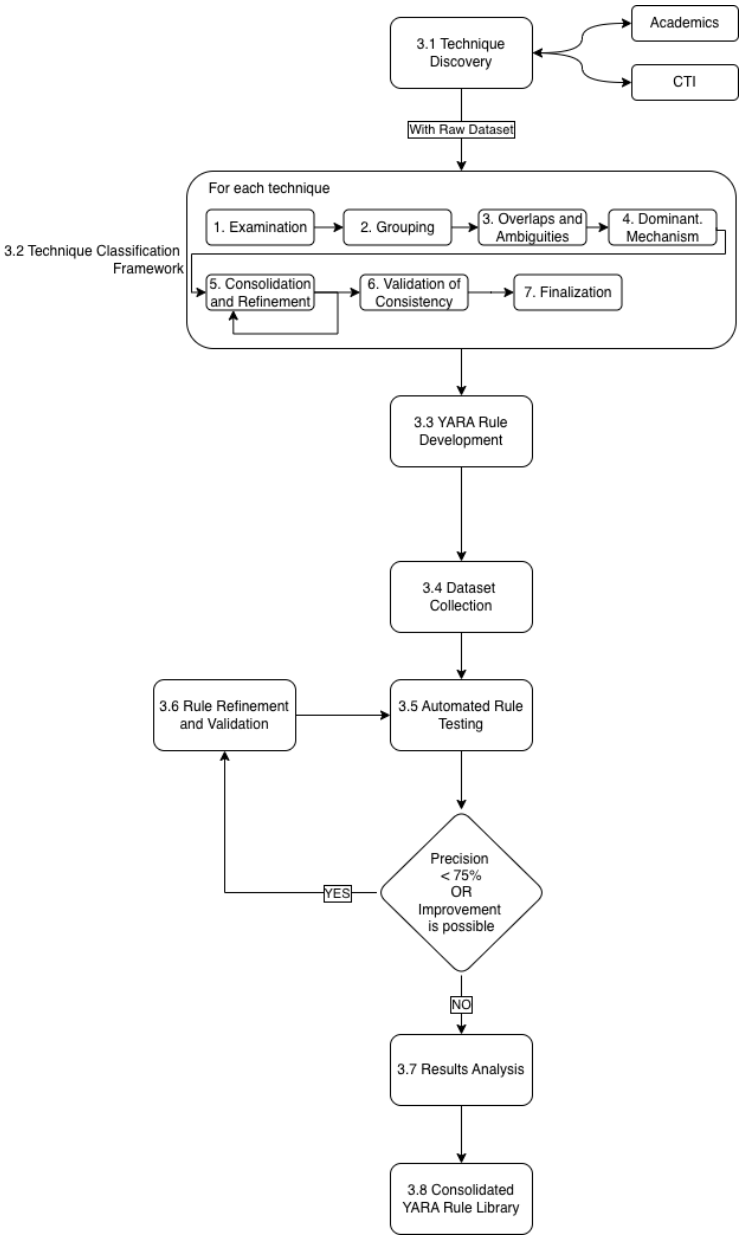

Figure 1. Workflow diagram of the methodology.

3.1. Technique Discovery

We conducted an exhaustive search for anti-analysis techniques, focusing on anti-VM and anti-sandbox behaviors observed in Windows malware between 2014 and 2025. Our data sources spanned both academic and operational domains:

- Academic literature: peer-reviewed papers from leading cybersecurity journals and conference proceedings, with particular emphasis on works that explicitly document evasion or anti-analysis mechanisms.
- Threat intelligence (CTI) sources: repositories and feeds from platforms such as VirusTotal, MISP, and open-source CTI aggregators.
- Vendor and community resources: whitepapers, GitHub projects, and technical blogs maintained by antivirus and EDR vendors.

For each documented method, we recorded the description, relevant source references, and, where available, code snippets, pseudocode, or disassembly fragments. This ensured that both conceptual and practical variants of each anti-analysis check were captured.

To avoid redundancy, we normalized the dataset by merging entries that represented variations of the same fundamental mechanism (e.g., different implementations of a CPUID hypervisor probe). This deduplication ensured that each entry corresponded to a unique anti-analysis technique before proceeding to classification and rule design.

3.2. Technique Classification Framework

After collecting all unique techniques, we organized them into a coherent classification framework based on their dominant detection mechanism. In this context, the dominant mechanism refers to the primary operational behavior through which a technique achieves evasion and that can be observed through static artifacts.

When a technique could reasonably be associated with multiple categories, classification was determined by identifying the mechanism that most directly drives the evasion logic and is most consistently reflected in detectable features, such as API usage, instruction patterns, or system artifacts. In cases of residual ambiguity, priority was given to the mechanism that provided the most reliable and distinctive indicators for static detection.

This process followed a transparent and iterative workflow designed to ensure consistency and reproducibility:

1. Detailed Examination of Each Technique: Each entry was analyzed in depth to determine its primary evasion mechanism (e.g., CPU instruction probe, timing measurement, file or registry artifact lookup, API hook inspection, hardware or BIOS query, network or WMI query, user-interface interaction).
2. Provisional Grouping: Techniques sharing similar indicators were grouped into provisional clusters. For example, instructions leveraging `CPUID` or `SIDT` were placed under CPU fingerprinting, whereas high-resolution timing checks were grouped under timing-based heuristics.
3. Resolution of Ambiguities: Ambiguous techniques combining multiple indicators (e.g., measuring time while reading BIOS data) were reviewed and assigned to the category corresponding to their most distinctive detection mechanism.
4. Iterative Consolidation and Validation: Clusters were merged, split, or renamed through multiple refinement rounds until all techniques mapped unambiguously to a single, stable category.

The final framework provided a structured set of categories that covered the full range of anti-VM and anti-sandbox mechanisms encountered in Windows malware. These categories guided the subsequent YARA rule creation process.

3.3. YARA Rule Development

For each unique anti-analysis technique identified, we authored a YARA rule designed to statically detect byte or string artifacts indicative of that behavior. When existing community rules were available, they were reviewed and refined to improve precision; otherwise, new rules were developed from scratch.

The construction of each rule followed a structured process combining literature analysis, reverse engineering insights, and empirical inspection of representative samples. For every technique, we first derived its expected implementation primitives based on documented behavior, including specific API calls, CPU instructions, registry keys, file-system artifacts, and characteristic string literals. When available, public proof-of-concept code, malware reports, and disassembly fragments were analyzed to identify concrete byte sequences or strings associated with the technique. In addition, a subset of malware samples known to implement particular evasion behaviors was manually inspected to extract recurring constants, opcode patterns, or structural features suitable for static detection.

Candidate patterns were then evaluated and generalized to ensure both representativeness and robustness. Patterns that were overly specific to individual samples were abstracted using wildcards, alternative string sets, or structural conditions, whereas patterns that were too generic (e.g., common API usage without contextual constraints) were either refined or discarded. Preference was given to artifacts intrinsically linked to the detection logic of the technique, such as hypervisor vendor strings, specific I/O ports, or known sandbox-related resources, rather than incidental implementation details.

To reduce false positives, each rule incorporated contextual constraints whenever possible. These included combinations of multiple indicators, section-based filtering, file-size bounds, and entropy conditions. The co-occurrence of independent artifacts was frequently used to increase specificity. Patterns that appeared frequently in benign software during preliminary testing were iteratively refined or removed.

Each rule contained the following components:

- Metadata: including descriptive and traceability information such as a human-readable description of the technique, author, creation date, rule version, an internal identifier, and references to external documentation. This metadata structure supports reproducibility, facilitates rule maintenance, and enables integration with automated analysis workflows by providing contextual information about the intent and origin of each rule.
- Strings and Patterns: ASCII/Unicode literals, hexadecimal sequences, or regular expressions characteristic of the technique's implementation.
- Conditional Clauses: filters such as file size bounds, section name constraints, or entropy thresholds to reduce false positives.

All provisional rules were stored in a structured working directory, labeled by category and technique name. Rule development was performed manually to ensure accuracy, while testing and refinement were partially automated.

Although rule development involved manual analysis, the process followed a consistent and reproducible methodology grounded in documented technique behavior, publicly available code artifacts, and systematic validation against large datasets. All resulting rules, along with their metadata and refinement history, are released publicly to facilitate independent verification and further extension by the research community.

To illustrate the structure of the developed rules, Figure 2 shows an example targeting user-interaction evasion through mouse hooks and click monitoring. This rule detects binaries that install low-level mouse hooks via `SetWindowsHookExA/W` or that asynchronously inspect mouse-button activity through `GetAsyncKeyState`, in combination with mouse-related event constants. The rule also includes metadata for traceability and requires the PE header signature to reduce irrelevant matches.

```
rule Detect_Mouse_Click_Hooks
{
    meta:
        description = "Detects malware using mouse hooks or tracking clicks to
        evade analysis"
        author = "Sebastien Kanj"
        date = "2025-03-15"
        version = "2.0"
        internal_code = "53"
        reference = "https://evasions.checkpoint.com/src/Evasions/techniques/\
        hooks.html#check-user-clicks-via-mouse-hooks"

    strings:
        // API for setting mouse hooks
        $set_mouse_hook = "SetWindowsHookExA" wide ascii nocase
        $set_mouse_hook_w = "SetWindowsHookExW" wide ascii~nocase

        // Hook type for detecting mouse input
        $mouse_hook = "WH_MOUSE_LL" wide ascii~nocase

        // Mouse events being monitored
        $mouse_event1 = "WM_MOUSEMOVE" wide ascii nocase
        $mouse_event2 = "WM_NCLBUTTONDOWN" wide ascii nocase
        $mouse_event3 = "WM_LBUTTONUP" wide ascii nocase
        $mouse_event4 = "VK_LBUTTON" wide ascii nocase
        $mouse_event5 = "VK_RBUTTON" wide ascii nocase
        $mouse_event6 = "VK_MBUTTON" wide ascii~nocase

        // API for tracking user input asynchronously
        $get_async_keystate = "GetAsyncKeyState" wide ascii~nocase

    condition:
        uint16(0) == 0x5A4D and
        (
            (
                ($set_mouse_hook or $set_mouse_hook_w) and
                $mouse_hook and
                any of ($mouse_event*)
            ) or
            (
                $get_async_keystate and
                any of ($mouse_event*)
            )
        )
}
```

Figure 2. Example YARA rule targeting mouse-hook-based user-interaction evasion.

This example reflects the general rule-construction strategy used throughout the library: combining semantically meaningful API names, behavior-specific constants, and lightweight structural constraints in order to capture anti-analysis logic while limiting overly generic matches. In the full repository, many rules additionally include further contextual restrictions introduced during refinement to improve precision against benign software.

3.4. Dataset Collection

To evaluate rule precision and false-positive behavior, we assembled two labeled corpora:

1. Malware Corpus (***M***): 459,508 Windows PE samples collected from the Bazaar repository [9] (February 2020–December 2024), spanning diverse families including Trojans, Backdoors, Ransomware, and Infostealers.
2. Goodware Corpus (***G***): 92,508 benign Windows executables from the Assemblage datasets [10], comprising:
    - 62,869 system and application binaries representing common Windows software.
    - 29,639 vcpkg packages representing third-party libraries widely used in Windows environments.

In addition to dataset size, several characteristics of the collected corpora were considered in order to contextualize the evaluation results. The malware dataset was obtained from MalwareBazaar over the period February 2020 to December 2024 and includes all available samples during this interval. As such, it reflects a broad and naturally occurring distribution of malware families, campaigns, and implementation strategies observed in the wild. While MalwareBazaar provides metadata such as malware type and potential family labels, these annotations were not used in the analysis due to the absence of a documented and consistent labeling methodology.

The temporal distribution of the malware corpus spans multiple years and exhibits variability in the number of samples per month, capturing fluctuations in real-world malware activity. This longitudinal coverage supports the temporal analysis presented in Section 6, although no explicit balancing across time periods or families was enforced.

No explicit deduplication was performed on the collected samples. However, the dataset does not contain identical binaries, as each sample is uniquely identified by its hash. Nevertheless, it may include closely related variants originating from the same malware family or campaign, potentially differing through minor modifications, repacking, or incremental evolution. While this may introduce some degree of redundancy at the behavioral level, it also reflects realistic conditions in large-scale malware repositories and preserves the natural distribution of samples encountered in operational environments.

All analyzed files were restricted to Windows Portable Executable (PE) format to ensure consistency with the scope of the study. No additional filtering or preprocessing was applied to remove packed or obfuscated binaries. As a result, both corpora may include samples employing packing, encryption, or other obfuscation techniques. These transformations can conceal or alter static artifacts, thereby affecting YARA rule matching and contributing to conservative estimates of technique prevalence.

The goodware dataset, derived from the Assemblage project, includes a wide range of system binaries, application software, and third-party libraries. This diversity provides broad coverage of benign behaviors, but may also include legitimate uses of APIs or patterns that overlap with those targeted by anti-analysis rules.

Overall, while the datasets provide large-scale and diverse coverage suitable for empirical evaluation, they should not be considered exhaustive representations of all malware or benign software. The reported results should therefore be interpreted in light of these practical constraints, which are inherent to large-scale malware analysis studies.

Both corpora were indexed on network-attached storage to enable automated querying during evaluation.

3.5. Automated Rule Testing

Each rule was automatically executed using YARA 4.5.2 against both corpora. For every rule r, we collected the following metrics:

- TP_r: number of malware samples matched.
- FP_r: number of benign binaries matched.
- $\mathrm{nTP}_r = \mathrm{TP}_r / M$: normalized true positives.

- $\text{nFP}_r = \text{FP}_r / G$: normalized false positives.
- $\text{Precision}_r = \frac{\text{nTP}_r}{\text{nTP}_r + \text{nFP}_r}$.

Because the available corpora are labeled only at the malware/benign level and not at the level of individual anti-analysis techniques, recall could not be measured directly for each YARA rule. In other words, while a rule match can be counted reliably, the total number of binaries that truly implement a given evasive technique is unknown for the full dataset. For this reason, we use precision as a rule-level reliability metric rather than as a complete measure of detector performance. This choice allows us to evaluate how often a matched artifact is associated with malware rather than benign software, while acknowledging that undetected implementations of the same technique may exist.

A rule was provisionally accepted if $\text{Precision}_r > 0.75$. The 75% threshold was selected as a research-oriented filtering criterion rather than an operational deployment target. Its purpose is to distinguish comparatively reliable rules from noisier ones while preserving sufficient coverage to study the static detectability of different anti-analysis categories. In production security pipelines, higher thresholds or additional validation stages would typically be required depending on the intended use case. Accordingly, rules retained under this criterion should be interpreted as empirically useful indicators for analysis and triage, not as standalone production signatures. Rules below this threshold were flagged for refinement and manual review.

In addition to rule-level precision, we compute aggregate sample-level coverage statistics to provide a broader view of detection capability across the dataset. These metrics, reported in Section 6.1, capture the proportion of samples matched by at least one rule and the average number of rule matches per sample, offering an approximate indication of the prevalence of statically observable anti-analysis artifacts.

3.6. Rule Refinement and Validation

Rules failing the precision criterion underwent two rounds of manual refinement, involving:

- Pattern tightening: refining string literals or hexadecimal sequences to focus on distinctive instruction patterns or constant values directly associated with the technique, while removing ambiguous substrings that may occur in benign contexts;
- Constraint adjustment: introducing or modifying contextual conditions such as file size limits, section names, entropy thresholds, or required co-occurrence of multiple indicators to reduce matches in benign binaries;
- Generic pattern elimination: removing or replacing patterns that rely on broadly used APIs or common programming constructs without sufficient contextual specificity, which were found to generate high false-positive rates;
- Rule consolidation: merging rules that targeted equivalent or closely related implementations of the same technique in order to reduce redundancy and improve maintainability of the rule set;
- Iterative validation feedback: analyzing false-positive and true-positive samples after each evaluation round to identify systematic sources of error, and incorporating these observations into subsequent refinements.

After each refinement, the updated rules were automatically re-executed on both datasets. We continued iterations until no further improvement was achievable without reducing detection coverage. In a subset of cases, known malware samples exhibiting specific evasion behaviors were manually verified in virtualized environments to confirm the correctness of detections.

Each rule was ultimately classified as:

- Accurate: $\text{Precision}_r \geq 0.75$.

- Marginal: $0.50 \leq \text{Precision}_r < 0.75$.
- Unreliable: $\text{Precision}_r < 0.50$ (excluded from the final library).

3.7. Result Analysis

With validated rules, we performed aggregate analyses across the malware corpus to identify:

- The most prevalent evasion techniques (nTP_r ranking);
- Correlations between categories and malware families;
- Temporal trends in evasion behaviors (2020–2024);
- Recurring false-positive motifs in benign binaries.

These analyses provided insights into the practical distribution and evolution of anti-analysis techniques in contemporary Windows malware.

3.8. Consolidated YARA Rule Library

All validated YARA signatures were consolidated into a single structured file, `rules.yara`. Each rule includes descriptive metadata, category tags, and the detection logic. This unified library facilitates direct deployment in sandboxes, antivirus scanners, and security orchestration pipelines, supporting automated detection of evasive Windows malware behaviors.

4. Identification and Characterization of Anti-Virtual-Machine and Anti-Sandbox Techniques

To assemble a comprehensive list of anti-analysis checks specific to Windows malware, we combined techniques curated in community and vendor resources, notably the Check Point "Evasion" repository [26] and the "Unprotect Project" [27]. After deduplication and normalization, we identified a total of 94 unique techniques that are used by Windows malware to detect virtual machines or sandboxed environments.

4.1. Classification Framework

To facilitate systematic analysis and subsequent rule development, we introduce a classification framework that groups anti-analysis checks by their dominant detection mechanism. The nine categories below classify each technique according to how it probes for virtualization or sandbox artifacts. While some techniques could plausibly span multiple mechanisms, we assign each to the category that best reflects its primary indicator, for clarity and reproducibility.

1. Instruction-Based Checks: Instruction-based checks exploit CPU opcodes or specialized instructions whose behavior differs under virtualization. For example, the CPUID instruction can reveal a hypervisor vendor string or set a "hypervisor present" bit when executed on a virtual platform. The "Red Pill" technique uses SIDT/LIDT to read the Interrupt Descriptor Table (IDT) base, which is relocated in many hypervisors. Additional checks include IN-port accesses to magic values (e.g., VMware's I/O port 0x5658), execution of undefined or privileged opcodes (which may fault differently in a VM), and inconsistencies when executing MMX/SSE instructions. These checks typically rely on a single, deterministic instruction or small sequence that directly exposes virtualization artifacts at the CPU level.
2. Timing-Based Evasions: Timing checks measure execution delays or clock skew introduced by virtualization or sandbox instrumentation. Common approaches include comparing high-resolution timers (RDTSC, `QueryPerformanceCounter`) against lower-resolution APIs (`GetTickCount`/`GetTickCount64`), detecting Sleep acceleration or bypassing, and scheduling deferred execution that exceeds automated analysis

lifetimes. Cross-referencing multiple timing sources or inserting repeated delays helps identify latency overhead imposed by hypervisors or monitors. This category also includes date/time gating to evade short-term analyses.

3. File and Registry Artifacts: Techniques that search for static indicators of virtualization or sandbox platforms in the file system or registry (e.g., hypervisor drivers like `VBoxGuest.sys`, sandbox DLLs such as `sbiedll.dll`, uninstall entries, product-specific paths, or analysis-tool executables). These checks rely on artifacts often unique to particular hypervisors or sandbox products.
4. Hardware Fingerprinting: Direct interrogation of hardware properties that differ between physical machines and virtual instances (e.g., disk geometry, virtual NIC OUIs, BIOS/SMBIOS vendor strings, RAM size, core counts, ACPI contents, GPU identifiers). Discrepancies versus expected physical values indicate virtualization.
5. API and Process Hooking: Detection of instrumentation introduced by dynamic analysis frameworks or sandbox hooks (e.g., IAT integrity for `ntdll.dll`, scanning function prologues for trampolines, comparing on-disk vs in-memory stubs, anomalous DLL load order, code caves, kernel/user callbacks). The aim is to uncover API interception or inline patching.
6. OS and System-Call Interrogation: Use of undocumented structures or direct system calls to reveal hypervisor/sandbox artifacts at kernel or system level (e.g., `NtQuerySystemInformation` for driver lists, SSDT or HAL inspection, executive callbacks, virtualization-specific exception codes). This category also includes VMware I/O port probing (e.g., port 0x5658 magic value), which we place here to reflect its reliance on system-level I/O semantics.
7. Network and WMI Queries: This category consolidates techniques that query management and network interfaces to uncover virtualization or sandbox characteristics. We intentionally group WMI and networking together, as both are frequently employed through overlapping code paths and share underlying system dependencies such as RPC/DCOM and CIM providers. In practice, WMI is used not only for local configuration queries but also for retrieving network-related data (e.g., adapter characteristics, IP configuration, BIOS metadata), making it analytically consistent to treat these sources jointly. Moreover, sandbox and hypervisor environments often instrument WMI and network APIs using common interception layers, which can introduce correlated anomalies detectable by malware. Additional checks rely on WMI to infer system uptime, process creation, or network reset times, which can reveal snapshot restores and other transient sandbox behaviors.
8. UI and Human-Interaction Checks: Evidence of real user activity or graphical rendering absent in automated sandboxes (e.g., `GetCursorPos/GetLastInputInfo` activity, invisible UI controls requiring clicks, GUI rendering latencies, window focus changes, scroll events, recent document history). These distinguish human-operated systems from fully automated analysis platforms.
9. Firmware and BIOS Checks: Interrogation of system-level tables for virtualization-specific entries (e.g., `GetSystemFirmwareTable` or `NtQuerySystemInformation` to read DMI/SMBIOS strings like “VMware” or “innotek GmbH”; UEFI variable namespaces; ACPI fields). Many virtual platforms emulate BIOS/UEFI data that can be recognized reliably.

The distribution of identified techniques across categories is summarized in Table 1, showing absolute counts and percentages of the total.

Table 1. Number and percentage of identified anti-analysis techniques per category.

Category	Count	Percentage
Instruction-Based Checks	9	9.6%
Timing-Based Evasions	16	17.0%
File and Registry Artifacts	14	14.9%
Hardware Fingerprinting	18	19.1%
API and Process Hooking	8	8.5%
OS and System-Call Interrogation	15	16.0%
Network and WMI Queries	7	7.4%
UI and Human-Interaction Checks	5	5.3%
Firmware and BIOS Checks	2	2.1%
Total	94	100%

Our classification framework is consistent with prior taxonomies of anti-analysis techniques while extending them in both scope and operational applicability. Previous works, such as Afianian et al. [13] and Galloro et al. [14], provide comprehensive conceptual categorizations of evasion behaviors, typically distinguishing between environment-aware checks, timing-based evasion, and anti-instrumentation mechanisms. However, these taxonomies are primarily descriptive and are not designed to support direct implementation of detection mechanisms.

In contrast, the framework proposed in this study is explicitly designed to bridge conceptual classification and practical detection. Each category is defined in terms of its dominant observable artifacts, enabling systematic mapping from technique descriptions to static YARA signatures. Furthermore, our taxonomy consolidates techniques from both academic literature and operational threat intelligence sources into a unified structure that is directly aligned with rule development and empirical validation. As a result, it complements existing taxonomies by providing an operational perspective focused on reproducible and automatable detection.

4.2. Identified Techniques

The following subsections provide concise definitions and illustrative examples for each anti-analysis check cataloged in our classification. Techniques were identified from the foundational studies in Section 2 and, primarily, from two community-driven repositories: the Check Point "Evasion" list [26] and the UnprotectProject [27]. Each entry is grouped by category to clarify its underlying detection mechanism and includes a brief description of the targeted artifact or behavior.

- **Instruction-Based Checks**:
  - *Hypervisor brand via CPUID instruction*: The CPUID instruction provides processor and feature information, including a hypervisor vendor string when run under virtualization.
  - *Hypervisor existence via CPUID instruction*: The CPUID instruction can detect a hypervisor by checking the "hypervisor present" flag in ECX.
  - *MMX support via CPUID instruction*: Malware may use MMX instructions as a test; failure or exception behavior can indicate an emulated environment lacking full MMX support.
  - *The Red Pill*: Uses SIDT/LIDT to read the Interrupt Descriptor Table base, which is relocated by many hypervisors.
  - *No Pill*: Reads the Local Descriptor Table Register (LDTR); a nonzero limit often indicates a VM.

  - *Execution of illegal instructions-VirtualPC*: Issues undefined or privileged opcodes whose fault behavior differs in VirtualPC versus physical hardware.
  - *VPCEXT Instruction Detection*: Invokes the VPCEXT instruction, which exists only under VirtualPC, to confirm virtualization.
  - *Bochs Emulator Detection via CPU Artifacts*: Examines CPU artifacts (e.g., CPUID leafs) that reveal the Bochs emulator.
- **Timing-Based Evasions**:
  - *Simple Delaying Operation*: Inserts a long Sleep or loop to exceed the maximum analysis time of a sandbox.
  - *Deferred Execution via Task Scheduler*: Schedules a task for delayed execution, evading sandboxes that only run samples briefly.
  - *Delayed Malicious Actions Until Reboot*: Uses Windows startup scripts or services so the payload runs only after a reboot, bypassing short-lived analysis.
  - *Date-Specific Execution*: Executes payload only on predefined dates, avoiding sandboxes that lack those date conditions.
  - *Parallel delays using different methods*: Uses multiple delay APIs (e.g., `SetWaitable Timer` and `Sleep`) to measure real elapsed time and detect sleep-skipping indicative of a sandbox.
  - *Multi-source clock comparison*: Cross-checks time readings from `RDTSC`, `GetTickCount`, and `QueryPerformanceCounter` to detect virtualization-induced drift.
  - *Sleep-skipping detection via timing discrepancies*: Cross-checks elapsed time using `QueryPerformanceCounter`, `GetTickCount64`, and `KUSER\SHARED\DATA` ticks to detect discrepancies caused by sandbox sleep-skipping.
  - *Multi-Method System Time Check*: Compares system time from `GetSystemTimeAs FileTime` and `NtQuerySystemInformation` to detect discrepancies from sandbox sleep-skipping.
  - *Delay value integrity check*: Verifies that Sleep is actually delayed for the intended duration rather than being fast-forwarded by a sandbox.
  - *Absolute Timeout Detection*: Uses `NtDelayExecution` with absolute timeouts to detect sandboxes that mishandle absolute delays, revealing sleep-skipping.
  - *Cross-Process Time Comparison*: Compares elapsed time between two processes; significant discrepancies reveal sandbox sleep-skipping.
  - *External Time Source Verification*: Retrieves current time from an external source (e.g., NTP or HTTP header) to detect sandbox clock manipulation and bypass internal VM time overrides.
  - *RDTSC and RDTSCP Timing Discrepancy Detection*: Measures cycle counts with RDTSC/RDTSCP (often bracketed by CPUID) around operations like `GetProcessHeap` or `CloseHandle`; significant timing variations reveal virtualization.
  - *Last Boot Time Discrepancy Detection*: Compares system last-boot time via WMI against BIOS counters; VM snapshot restores can create anomalies.
  - *Expiration Date-Based Evasion*: Embeds a future activation date in the binary so the sample remains dormant until after typical analysis periods.
  - *API Hammering*: Repeatedly invokes benign APIs (e.g., `printf`) to delay payload execution and outlast the sandbox's analysis time.
  - *Short System Uptime Detection*: Checks system uptime via `GetTickCount`, `Get TickCount64`, or `NtQuerySystemInformation`; unusually short uptime indicates a freshly initialized sandbox.
- **File and Registry Artifacts**:

- *Specific files' existence*: Checks for known sandbox or VM binary files (e.g., `VMMouse.sys`, `VBoxGuest.sys`) that reside only in virtual environments.
- *Specific directories existence*: Detects directories (e.g., `C:\Sandbox`) unique to sandbox implementations.
- *Executables launched from specific directories*: Verifies if the sample's parent process path matches known sandbox-injection directories.
- *Full path executables containing specific strings*: Scans the full executable path for substrings (e.g., "_SAND_") used by sandbox environments.
- *Executable files with specific names present in physical disk drives' root*: Looks for filenames that sandboxes commonly drop in the root for instrumentation.
- *Specific registry paths' existence*: Probes registry paths used by virtualization tools.
- *Specific registry keys containing particular values*: Reads registry values (e.g., "VMTools"); presence suggests virtualization.
- *VBAWarning enabled*: Checks Office VBA security settings in the registry; sandboxes often disable warnings to auto-run macros.
- *Recently Opened Documents Check*: Reads "RecentDocs" entries in the registry; lack of user history suggests a sandbox.
- *Browser History URL Count Check*: Counts URLs in browser history; a near-empty history suggests an analysis environment.
- *Installed Software Package Check*: Enumerates installed programs via registry; missing typical software (e.g., antivirus) can indicate a sandbox.
- *Any.Run Sandbox Detection*: Searches for artifacts (e.g., `ANYRUN\CAPTURE`) left by the Any.Run platform.
- *Goat File Detection in Defender AV Emulator*: Checks for the presence of known "goat files" used by Defender's emulator.
- *Default Wallpaper Detection*: Reads the current desktop wallpaper path; generic or missing wallpaper often occurs in VM templates.

- **Hardware Fingerprinting**:
  - *Common MAC addresses*: Enumerates adapters via `GetAdaptersAddresses`/`GetAdaptersInfo` and checks OUIs linked to VMware/VirtualBox.
  - *Specific network adapters name existence*: Identifies adapter names tied to VMs (e.g., "VirtualBox Host-Only Network").
  - *Low RAM Detection*: `GetMemoryStatusEx` shows unusually low RAM (e.g., $<$1 GB) indicative of sandbox/VM.
  - *Unusual Screen Resolution Detection*: `GetSystemMetrics`/`GetMonitorInfo` reveals atypical resolutions for physical hosts.
  - *Low Processor Count Detection*: `GetSystemInfo`, PEB, or `KUSER\SHARED\DATA` show very low core counts.
  - *Low Monitor Count Detection*: `EnumDisplayMonitors` or `SM_MONITORS` indicates single/absent monitors typical of VMs.
  - *Small Disk Size and Free Space Detection*: `DeviceIoControl`, `IOCTL_DISK_GET_LENGTH_INFO` and `GetDiskFreeSpaceExA/W` show unusually small volumes.
  - *Virtual Hard Disk Boot Detection*: `IsNativeVhdBoot` confirms OS booted from VHD.
  - *Specific Virtual Device Detection*: Attempts to open/query device names (e.g., `\.\vmmem`) via `NtCreateFile`.
  - *Specific HDD Name Detection*: `SetupDi*` APIs reveal HDD model strings (e.g., "VBOX HARDDISK").
  - *Specific HDD Vendor ID Detection*: `IOCTL_STORAGE_QUERY_PROPERTY` returns vendor IDs like "VMware"/"QEMU."

  - *Audio Device Absence Detection*: No enumerated audio devices suggests virtual/sandboxed hosts.
  - *CPU Temperature Availability Detection*: Temperature queries fail or return zero on VMs.
  - *Physical Display Adapter Detection*: `Direct3DCreate9/GetAdapterIdentifier` indicates generic/software adapter only.
  - *Printer Detection*: No installed printers via `EnumPrinters`.
  - *USB Drive Detection*: No mounted USB drives via `SetupDi` or `GetLogicalDrives`.
  - *CPU Core and Hyperthreading Detection*: Core/HT flags from PEB/`CPUID` suggest underprovisioned VM.
  - *Odd CPU Thread Count Detection*: Odd thread counts (e.g., 3) indicate VM manipulation.
- **API and Process Hooking**:
  - *Specific libraries loaded in the process address space*: `GetModuleHandle` checks for analysis-injector DLLs.
  - *Specific functions present in particular libraries*: `GetProcAddress/LdrGetProcedureAddress` locate sandbox-specific exports.
  - *Certain libraries safely loaded*: `LoadLibraryA/W` behavior on rare/fake DLLs reveals emulation quirks.
  - *Specific artifacts present in the process address space - Sandboxie*: `NtQueryVirtualMemory` scans for Sandboxie stubs.
  - *Unbalanced Stack Detection*: Precise stack layout detects corruption by hooks.
  - *System Function Hook Detection*: Prologue comparison for trampolines in critical `Nt*` APIs.
  - *Mouse Click Detection via Hooks*: `SetWindowsHookEx/GetAsyncKeyState` expect real user input.
  - *Incorrect Hook Detection*: Argument/validation anomalies in hooked `Nt*` paths expose sandboxes.
- **OS and System-Call Interrogation**:
  - *Specific processes and services running*: Presence of known monitoring services (e.g., "VBoxService").
  - *Specific User Name Existence*: `GetUserNameA/W` matches typical sandbox accounts.
  - *Specific Host Name Existence*: `GetComputerNameExA/W` matches typical sandbox hostnames.
  - *Specific Global Mutex Detection*: `CreateMutex/OpenMutex` for VM-created mutexes.
  - *Specific Global Pipe Detection*: `CreateFile` on named pipes typical of VMs.
  - *Specific Global Object Detection*: `NtOpenDirectoryObject/NtQueryDirectoryObject` list VM-specific global objects.
  - *Specific Window Class Name Detection*: `EnumWindows/FindWindow` detect classes like `VBoxTrayToolWndClass`.
  - *Low Top-Level Window Count Detection*: `EnumWindows` reveals unusually few windows.
  - *Debug Privilege Detection*: Attempt to enable debug privilege and perform privileged actions.
  - *Wine Detection*: Probe Wine-specific inconsistencies or exports.
  - *Invalid Argument Detection for Delay Functions*: `NtDelayExecution` with invalid/unaligned timeouts.
  - *VMware I/O Port Detection*: Query VMware-specific I/O ports (e.g., 0x5658) and check for `VMXh` magic value.
  - *Active Directory Domain Membership Check*: Compare `LogonServer/ComputerName`.
  - *BuildCommDCBAndTimeouts Device String Validation*: Failure semantics for bogus device strings.

- **Network and WMI Queries**:
  - *Specific provider's name for network shares*: Shares revealing sandbox domains.
  - *Networks belonging to security perimeters*: Isolated/corporate subnets typical of sandboxes.
  - *NetValidateName result based anti-emulation*: `NetValidateName` anomalies in sandboxes.
  - *WMI Process Creation Evasion*: `Win32_Process`.Create to bypass `CreateProcessInternalW` hooks.
  - *WMI Task Scheduler Evasion*: `Win32_ScheduledJob`.Create for deferred execution.
  - *Last Boot Time Anomaly Detection*: WMI-based last boot time inconsistencies.
  - *Network Adapter Last Reset Time Check*: WMI-reported adapter reset times across snapshots.
- **UI and Human-Interaction Checks**:
  - *Mouse Movement Detection*: Repeated `GetCursorPos` polling for real input.
  - *User Interaction Request*: `MessageBox` gating on human click.
  - *Invisible Button Evasion*: Transparent button requiring real click.
  - *Scroll-Triggered Payload Activation*: Requires scroll event prior to activation.
  - *User Activity Detection*: `GetLastInputInfo` inactivity threshold.
- **Firmware and BIOS Checks**:
  - *Firmware Table String Detection*: SMBIOS/DMI strings exposing vendors such as "SeaBIOS," "VMware," or "innotek GmbH."
  - *SMBIOS firmware table string detection*: `NtQuerySystemInformation` extraction of DMI/SMBIOS strings indicating virtual hardware.

5. YARA Detection of Anti-Virtual-Machine and Anti-Sandbox Techniques

For each of the 94 anti-analysis technique identified in Section 4, we sought to develop a corresponding YARA rule capturing its most stable and discriminative static artifact. Where a reliable static signature could not be defined, either because the implementation of the technique varies substantially across malware families, or because the detection logic is inherently too generic and prone to false positives, we deliberately refrained from creating a rule. Specifically, no YARA rule was produced for the following twelve techniques:

- API Hammering.
- Bochs Emulator Detection via CPU Artifacts.
- Cross-Process Time Comparison.
- Delay Value Integrity Check.
- Delayed Malicious Actions Until Reboot.
- Expiration Date-Based Evasion.
- External Time Source Verification.
- Incorrect Hook Detection.
- Invisible Button Evasion.
- Odd CPU Thread Count Detection.
- Scroll-Triggered Payload Activation.
- User Interaction Request.

These omitted techniques rely on dynamic or context-dependent behaviors that cannot be reliably captured through static pattern matching. In several cases (e.g., timing or interaction checks), the behavioral nature of the evasion would have required dynamic analysis or runtime instrumentation to avoid excessive false positives against benign software.

In total, we authored 72 new YARA rules and updated 10 existing community rules, yielding a consolidated library of 82 YARA signatures explicitly targeting anti-VM and anti-

sandbox behaviors. This collection represents a significant expansion over previous studies: Afianian et al. [13] considered six techniques, Botacin and Uebel [19] five, Galloro et al. [14] sixty-five, and Branco et al. [28] three. Each signature was developed using the systematic methodology described in Section 3 and underwent empirical validation against both malware and goodware corpora.

During evaluation, rules that generated false positives were refined iteratively by constraining string and hexadecimal patterns, introducing conditional clauses, and reorganizing overlapping signatures. Each modification step, along with its rationale, was recorded to ensure reproducibility. Refinement continued until additional adjustments would have measurably reduced true-positive coverage, resulting in a curated and balanced rule set that achieves high precision without sacrificing generalization.

All validated YARA rules were compiled into a single file, `rules.yara`, where each rule is named after its corresponding anti-analysis technique. The complete rule set is publicly accessible for research and operational use (https://gist.github.com/sebastienkanj-upc/d514bc23c6d10cb55c692759fcc5226e, accessed on 22 February 2026). The repository is intended as a living library, open for peer review and community contributions to ensure continued coverage of emerging anti-analysis behaviors.

While YARA rule repositories are widely used in operational cybersecurity contexts, most existing collections are designed for malware family identification or indicator of compromise detection, rather than for systematically capturing anti-analysis logic. These repositories are typically curated in an ad hoc manner, with limited documentation of rule derivation, validation methodology, or false-positive behavior.

In contrast, the rule set presented in this study is constructed through a systematic, technique-driven process grounded in a comprehensive classification framework. Each rule is explicitly linked to a documented anti-analysis technique and is evaluated at scale against both malware and goodware datasets using a consistent precision-based methodology. This approach enables reproducibility, facilitates comparative analysis across technique categories, and provides a structured foundation for extending the rule set as new evasion methods emerge.

6. Discussion of the Results

This section analyzes and interprets the results obtained after executing the final set of 82 YARA rules against the malware and goodware datasets. The objective is to assess which anti-VM and anti-sandbox techniques are most prevalent in Windows malware, evaluate their detectability through static signatures, and identify trends over time. We also highlight implications for future research and operational detection strategies.

It is important to note that YARA operates purely on static artifacts. As a result, its effectiveness is bounded by the visibility of the underlying implementation details within the analyzed binaries. Malware samples employing obfuscation, packing, or runtime code generation may conceal these artifacts, causing the true prevalence of certain techniques to be underestimated. Consequently, the metrics reported here should be interpreted as lower-bound estimates of observable evasion behaviors among unobfuscated Windows samples.

6.1. Overall Rule Performance

All 82 YARA rules developed in this study were executed against the malware corpus. Among them:

- 77 rules yielded at least one match on malware samples.
- 5 rules produced zero matches; no false positives were observed, and manual inspection confirmed the patterns were valid but absent from the corpus.

Based on measured precision, rules were classified as follows:

- 42 rules (51.22%) were labeled Accurate, achieving precision above 75%.
- 16 rules (19.51%) were labeled Marginal, with precision between 50% and 75%.
- 24 rules (29.27%) were labeled Unreliable, with precision below 50%, and thus excluded from the final repository.

Figure 3 illustrates the precision distribution for all rules. Accurate rules cluster above the 75% threshold, while marginal and unreliable ones are more dispersed below this cutoff, reflecting the heterogeneity of static artifacts across techniques.

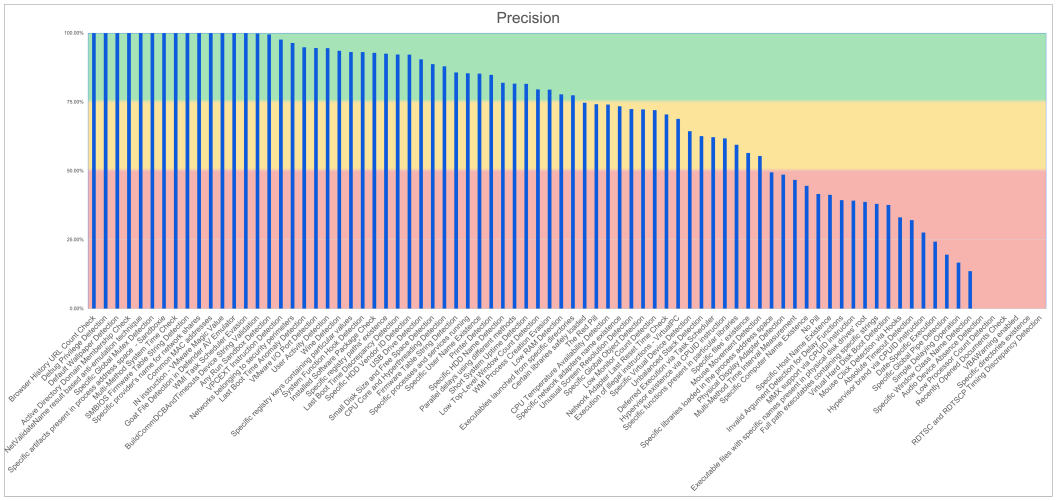

Figure 3. Precision per YARA rule across the malware and goodware datasets.

While YARA detections provide an automated means to infer the presence of evasion indicators, a rule match alone does not guarantee that the corresponding technique is executed by the sample. To mitigate this limitation, we manually reviewed a representative subset of matched binaries, particularly those belonging to families known to implement these checks, and confirmed that the matched byte sequences aligned with the expected functionality. Thus, our evaluation provides empirically supported approximations of technique occurrence rather than definitive behavioral confirmation. Because ground-truth labels for the presence of individual anti-analysis techniques are not available for the full corpus, recall cannot be measured directly. Instead, we focus on precision as a conservative reliability indicator, complemented by manual validation of representative samples. This approach reflects a common constraint in large-scale malware studies, where technique-level annotations are unavailable.

To complement rule-level precision, we report aggregate coverage statistics at the sample level. Across the evaluated corpus, 308,471 malware samples (67.13%) were matched by at least one rule, compared to 91,993 goodware samples (99.44%). On average, malware samples triggered 3.78 rule matches per file (median = 3), whereas goodware samples triggered 1.81 matches (median = 1).

While a higher proportion of goodware samples exhibit at least one rule match, this is expected given that many of the detected artifacts (e.g., API usage or system interactions) may also appear in benign software. However, the key difference lies in the density of detections: malware samples tend to trigger a significantly higher number of rules per file, reflecting the presence of multiple co-occurring indicators associated with anti-analysis behavior.

The distribution of rule matches further supports this interpretation. Considering only samples that triggered at least one rule, 27.8% of malware samples matched exactly one rule, while over 22% matched more than five rules. In contrast, 65.5% of goodware samples triggered only a single rule, and fewer than 4% exceeded five matches. This disparity

suggests that matches in goodware are typically isolated, whereas malware samples more frequently exhibit multiple concurrent indicators. As a result, multi-rule activation emerges as a stronger signal of potentially evasive behavior than individual rule matches.

These statistics provide an approximate indication of detection coverage at the corpus level. However, they should not be interpreted as recall, since the ground truth for the presence of individual anti-analysis techniques is not available. Instead, they reflect the prevalence and co-occurrence of statically observable artifacts captured by the rule set.

6.2. Top-Performing and Underperforming Rules

The rule targeting *Specific network adapters name existence* achieved the highest malware coverage, matching 162,805 samples with a precision of 73.48%. Although slightly below the 75% "Accurate" threshold, its high coverage indicates that network adapter enumeration remains a common virtualization check in modern malware.

Only one rule, *RDTSC and RDTSCP Timing Discrepancy Detection*, generated more false positives than true positives ($FP_r > TP_r$), primarily due to overlaps with benign performance-monitoring code. This rule was therefore classified as Unreliable and omitted from the public release.

6.3. Category-Level Observations

Aggregating results at the category level reveals clear behavioral patterns among malware samples:

- Timing-Based Evasions. This category produced the largest number of total matches (595,557 hits) but exhibited moderate precision, averaging 53.78%. Timing discrepancies, sleep-skipping, and clock skew detection remain pervasive in malware, yet their static signatures overlap significantly with legitimate application behavior (e.g., scheduling, benchmarking).
  - Implication: Timing checks are widespread but inherently difficult to capture with static signatures alone.
  - Recommendation: Combining static detection with short, lightweight dynamic execution (e.g., confirmation of timing anomalies) could enhance confidence while retaining scalability.
- Firmware and BIOS Checks. Although comparatively rare, this category achieved the highest mean precision (92.87%). Rules targeting SMBIOS or UEFI vendor strings (e.g., "SeaBIOS," "VMware") proved highly reliable indicators of virtualization.
  - Implication: Firmware- and BIOS-level detections yield high-confidence signals with minimal false positives.
  - Recommendation: These rules are well-suited for operational deployment in triage or attribution pipelines.

6.4. Temporal Trends in Technique Usage

To assess the temporal evolution of anti-analysis behaviors, we measured monthly detection counts per category between February 2020 and December 2024. As illustrated in Figure 4, both timing-based and instruction-based techniques increased sharply from late 2021 through 2023, coinciding with the widespread adoption of virtualization in enterprise and cloud environments. This likely reflects adaptation by malware developers to the increasing reliance on virtualized and sandboxed infrastructures for security analysis.

The inclusion of total sample counts per month (gray bars in Figure 4) enables a more nuanced interpretation of these trends. While detection counts generally correlate with dataset size, several technique categories—particularly timing-based and instruction-based evasions—exhibit disproportionate increases relative to the number of samples, suggesting

that the observed growth is not solely attributable to variations in dataset volume. Instead, these patterns indicate a genuine increase in the adoption of specific anti-analysis strategies.

Timing-based techniques, in particular, remain consistently dominant across all periods, both in absolute terms and relative to sample counts, highlighting their role as a baseline evasion strategy in modern malware. In contrast, hardware fingerprinting and system-level checks tend to scale more proportionally with dataset size, suggesting that these techniques are widely used but relatively stable over time.

It is important to note that the reported counts are based on raw rule matches and are not normalized by the total number of samples per month. Because the dataset exhibits variability in monthly sample volume, part of the observed fluctuations may be influenced by changes in dataset size. Nevertheless, the consistent upward trends across multiple independent categories support the interpretation that these observations reflect broader shifts in malware behavior rather than purely sampling effects.

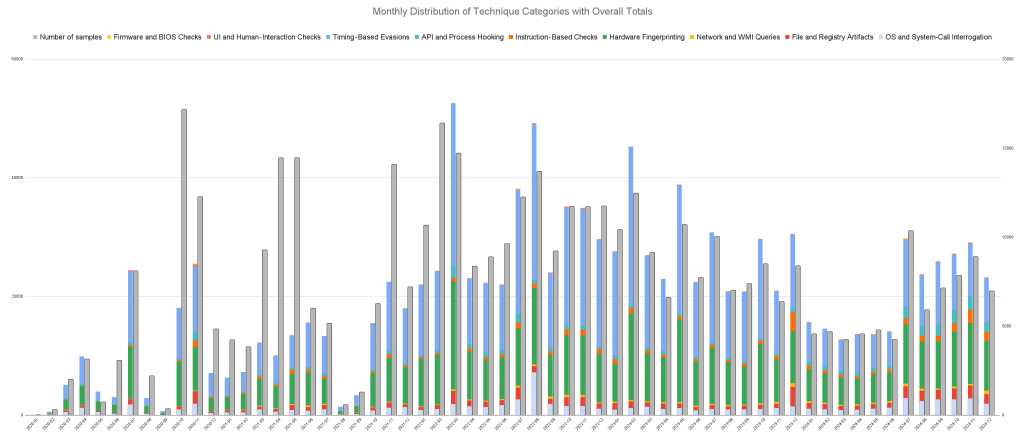

Figure 4. Temporal distribution of detected anti-analysis categories and total sample counts (February 2020–December 2024).

7. Conclusions and Future Work

This work systematically identified, categorized, and statically detected anti-VM and anti-sandbox techniques employed by Windows malware. Through an extensive review of academic literature, open-source repositories, and threat-intelligence sources, we enumerated 94 distinct evasion methods organized into nine mechanistic categories. For 82 of these techniques, we developed or refined YARA rules and validated their performance using large malware and goodware corpora. The iterative refinement process, comprising pattern tightening, conditional logic, and redundancy elimination, produced 42 Accurate, 16 Marginal, and 24 Unreliablerules. Firmware- and BIOS-based detections achieved the highest mean precision, whereas timing-based heuristics, despite their prevalence, proved more challenging to detect statically without generating false positives.

Beyond rule-level precision, our evaluation reveals important differences in how anti-analysis artifacts manifest across malware and benign software. While a high proportion of both malware and goodware samples exhibit at least one detectable artifact, malware samples tend to trigger a significantly higher number of rules per file. This indicates that anti-analysis behaviors in malware are typically composed of multiple co-occurring mechanisms, whereas matches in goodware are more often isolated and incidental. As a result, the density of rule activations—rather than their mere presence—emerges as a more informative signal for distinguishing structured evasion logic from benign overlap.

Because YARA operates on static representations, our measurements represent conservative estimates of evasion prevalence. Code obfuscation, packing, and polymorphic

transformations can conceal instruction sequences, modify control flow, or delay payload exposure, thereby preventing static signatures from triggering even when the evasion logic is present. Moreover, YARA matches in this study indicate the presence of static artifacts consistent with the implementation of anti-analysis techniques, but do not guarantee that such techniques are executed at runtime. Consequently, the reported measurements should be interpreted as approximations of observable implementation rather than definitive behavioral confirmation. Hybrid validation combining static signatures with brief dynamic execution will therefore be necessary to confirm behavioral activation and to mitigate the inherent limitations of purely static detection.

All validated rules were consolidated into a single file, `rules.yara`, and released publicly to foster reproducibility and community collaboration. The rule set is intended to support practitioner workflows for static detection of VM and sandbox-evasion behaviors, particularly as a source of interpretable indicators rather than a standalone classification mechanism. Temporal analysis further revealed a notable surge in timing- and instruction-based evasion between 2021 and 2023, aligning with increased enterprise virtualization.

The results directly address the research questions posed in this study.

- RQ1 is answered by showing that a substantial portion of anti-analysis behavior can be detected statically: more than half of the implemented rules achieved reliable precision, demonstrating that static YARA signatures can capture many implementation-stable evasion patterns. At the same time, sample-level analysis shows that these patterns tend to co-occur more densely in malware, reinforcing their relevance as indicators of structured evasion logic.
- RQ2 is clarified through the observed variability in rule performance: hardware- and artifact-based checks proved highly amenable to static detection, whereas timing logic and environment-sensitive triggers remained difficult due to benign overlap and implementation diversity.
- RQ3 is supported by the large-scale validation process itself, which revealed both the achievable reliability limits of static detection and the importance of iterative refinement using mixed malware and goodware corpora. In particular, empirical testing exposed rule brittleness, redundancy, and contextual dependencies that would not have been evident from design alone.

A key limitation of this study is the absence of technique-level ground-truth annotations across the full dataset, which prevents direct measurement of recall. While precision provides a robust indication of rule reliability, it does not capture the proportion of all instances of a given technique that are successfully detected. Addressing this limitation will require curated datasets with explicit technique labeling, controlled proof-of-concept samples, or targeted dynamic analysis to validate the presence and execution of specific evasion behaviors.

The proposed rule set can be applied in practical analysis workflows, such as large-scale malware triage or sandbox result enrichment. For example, static scanning of incoming samples with the rule library can provide early indicators of potential anti-analysis behavior, allowing analysts to prioritize samples that are more likely to evade dynamic inspection. Additionally, the presence of multiple evasion-related matches within a single binary may signal layered evasion strategies, informing decisions about deeper manual or dynamic investigation.

Several research directions remain open. First, techniques resistant to static detection could benefit from hybrid approaches combining static signatures with brief dynamic validation to confirm behavioral triggers. Second, as malware increasingly targets containerized and cloud environments, future extensions of this framework should address container-specific evasion and hypervisor-level concealment. Third, correlating multiple

co-occurring evasion patterns within individual samples could provide insights into layered evasion strategies and inform the design of composite detection models, particularly by leveraging rule activation density as a feature for higher-level analysis.

Finally, we plan to maintain and extend the public YARA repository through community feedback, continuous integration of new techniques, and the use of machine-learning-assisted pattern mining to propose candidate rules. A parallel research effort will investigate the potential of these YARA detections for forensic attribution, examining whether consistent rule activation patterns—especially their frequency and co-occurrence—can help associate malware with particular developer groups or threat actors. Achieving this will require longitudinal validation, family-labeled datasets, and stability metrics across time and campaigns. Through these ongoing efforts, we aim to strengthen both the detection coverage and analytical utility of static YARA signatures for threat research and operational defense.

Author Contributions: Conceptualization, S.K.; methodology, S.K. and J.P.; validation, J.P.; formal analysis, S.K. and G.V.; investigation, S.K. and G.V.; data curation, S.K.; writing—original draft preparation, S.K.; writing—review and editing, S.K. and J.P.; supervision, J.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Chair of Cybersecurity CARISMATICA, supported by the European Social Fund Plus and the Recovery, Transformation, and Resilience Plan, financed by the European Union (NextGenerationEU) under the auspices of INCIBE. This research was also funded by the Departament de Recerca i Universitats of the Generalitat de Catalunya through the Doctorats Industrials program (2021 DI 00092).

Data Availability Statement: The YARA rule set generated in this study is publicly available on GitHub at https://gist.github.com/sebastienkanj-upc/d514bc23c6d10cb55c692759fcc5226e/ (accessed on 22 February 2026). The malware samples analyzed were obtained from the publicly accessible MalwareBazaar repository (https://bazaar.abuse.ch/, accessed on 22 February 2026). The goodware datasets used in this study are available from the Assemblage project (https://assemblage-dataset.net/, accessed on 22 February 2026).

Acknowledgments: The authors would like to thank their colleagues in the Department of Telematics Engineering, Universitat Politècnica de Catalunya (ENTEL-UPC), for their valuable support and assistance. The authors also acknowledge INCIDE Digital Data S.L. for their technical support and collaboration in facilitating this research.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

YARA	Yet Another Recursive Acronym
VM	Virtual Machine
CTI	Cyber Threat Intelligence
EDR	Endpoint Detection and Response
PE	Portable Executable
TP	True Positive
FP	False Positive
CPU	Central Processing Unit
API	Application Programming Interface

References

1. Or-Meir, O.; Nissim, N.; Elovici, Y.; Rokach, L. Dynamic malware analysis in the modern era—A state of the art survey. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–48. [CrossRef]
2. Vasilescu, M.; Gheorghe, L.; Tapus, N. Practical malware analysis based on sandboxing. In *Proceedings of the 2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*; IEEE: New York, NY, USA, 2014; pp. 1–6.
3. Sihwail, R.; Omar, K.; Ariffin, K.Z. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. *Int. J. Adv. Sci. Eng. Inf. Technol.* **2018**, *8*, 1662–1671. [CrossRef]
4. Gao, Y.; Lu, Z.; Luo, Y. Survey on malware anti-analysis. In Proceedings of the Fifth International Conference on Intelligent Control and Information Processing, Dalian, China, 18–20 August 2014; pp. 270–275. [CrossRef]
5. VirusTotal. YARA Documentation. 2025. Available online: https://yara.readthedocs.io/en/stable/ (accessed on 7 November 2025).
6. Mahdi, R.H.; Trabelsi, H. Detection of malware by using yara rules. In *Proceedings of the 2024 21st International Multi-Conference on Systems, Signals & Devices (SSD)*; IEEE: New York, NY, USA, 2024; pp. 1–8.
7. Naik, N.; Jenkins, P.; Cooke, R.; Gillett, J.; Jin, Y. Evaluating automatically generated YARA rules and enhancing their effectiveness. In *Proceedings of the 2020 IEEE Symposium Series on Computational Intelligence (SSCI)*; IEEE: New York, NY, USA, 2020; pp. 1146–1153.
8. Lockett, A. Assessing the effectiveness of yara rules for signature-based malware detection and classification. *arXiv* **2021**, arXiv:2111.13910. [CrossRef]
9. VX-Underground. Bazaar Sample Collection. Available online: https://vx-underground.org/Samples/Bazaar%20Collection (accessed on 7 November 2025).
10. Liu, C.; Saul, R.; Sun, Y.; Raff, E.; Fuchs, M.; Southard Pantano, T.; Holt, J.; Micinski, K. Assemblage: Automatic binary dataset construction for machine learning. *Adv. Neural Inf. Process. Syst.* **2024**, *37*, 58698–58715.
11. Jadhav, A.; Vidyarthi, D.; Hemavathy, M. Evolution of evasive malwares: A survey. In *Proceedings of the 2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*; IEEE: New York, NY, USA, 2016; pp. 641–646.
12. Biondi, F.; Given-Wilson, T.; Legay, A.; Puodzius, C.; Quilbeuf, J. Tutorial: An overview of malware detection and evasion techniques. In *Proceedings of the International Symposium on Leveraging Applications of Formal Methods*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 565–586.
13. Afianian, A.; Niksefat, S.; Sadeghiyan, B.; Baptiste, D. Malware dynamic analysis evasion techniques: A survey. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–28. [CrossRef]
14. Galloro, N.; Polino, M.; Carminati, M.; Continella, A.; Zanero, S. A Systematical and longitudinal study of evasive behaviors in windows malware. *Comput. Secur.* **2022**, *113*, 102550. [CrossRef]
15. D'Elia, D.C.; Invidia, L.; Palmaro, F.; Querzoni, L. Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. *Digit. Threat. Res. Pract. (DTRAP)* **2022**, *3*, 1–13. [CrossRef]
16. Filho, A.S.; Rodríguez, R.J.; Feitosa, E.L. Evasion and countermeasures techniques to detect dynamic binary instrumentation frameworks. *Digit. Threat. Res. Pract. (DTRAP)* **2022**, *3*, 1–28. [CrossRef]
17. Gavrila, R.; Zacharis, A. Advancements in Malware Evasion: Analysis Detection and the Future Role of AI. In *Malware: Handbook of Prevention and Detection*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 275–297.
18. Mills, A.; Legg, P. Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques. *J. Cybersecur. Priv.* **2020**, *1*, 19–39. [CrossRef]
19. Botacin, M.; da Rocha, V.F.; de Geus, P.L.; Grégio, A. Analysis, anti-analysis, anti-anti-analysis: An overview of the evasive malware scenario. In Proceedings of the Anais do XVII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais, Brasilia, Brazil, 6–9 November 2017; pp. 250–263.
20. Kim, M.; Cho, H.; Yi, J.H. Large-scale analysis on anti-analysis techniques in real-world malware. *IEEE Access* **2022**, *10*, 75802–75815. [CrossRef]
21. Chen, P.; Huygens, C.; Desmet, L.; Joosen, W. Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware. In *Proceedings of the IFIP International Conference on ICT Systems Security and Privacy Protection*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 323–336.
22. Mirza, M.M.; Alsuwat, R.S.; Alqurashi, Y.M.; Alharthi, A.A.; Alsuwat, A.M.; Alasamri, O.M.; Hussain, N.A. DIGITRACKER: An Efficient Tool Leveraging Loki for Detecting, Mitigating Cyber Threats and Empowering Cyber Defense. *J. Cybersecur. Priv.* **2026**, *6*, 25. [CrossRef]
23. Cao, V.L.; Dai Nguyen, D. Bypassing anti-emulation methods for malware detection. *J. Comput. Sci. Cybern.* **2024**, *40*, 233–248. [CrossRef]

24. Bhattacharjee, S.; Patil, P.; Patil, V.; Nage, P. Hybrid Multi-Stage Framework for Advanced Malware Detection: Enhancing Accuracy & Resilience. In *Proceedings of the 2025 7th International Conference on Energy, Power and Environment (ICEPE)*; IEEE: New York, NY, USA, 2025; pp. 1–6.
25. Fukushima, Y.; Sakai, A.; Hori, Y.; Sakurai, K. A behavior based malware detection scheme for avoiding false positive. In *Proceedings of the 2010 6th IEEE Workshop on Secure Network Protocols*; IEEE: New York, NY, USA, 2010; pp. 79–84.
26. Check Point Research. Evasions Cheatsheet. 2024. Available online: https://evasions.checkpoint.com/ (accessed on 22 February 2026).
27. DarkCoderSc and fr0gger_ . UnprotectProject. 2024. Available online: https://unprotect.it/ (accessed on 22 February 2026).
28. Branco, R.R.; Barbosa, G.N.; Neto, P.D. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat* **2012**, *1*, 1–27.