



SAPIENZA
UNIVERSITÀ DI ROMA

Dissecting Malware Dynamics: Gleaning Behavioral and Semantic Insights for Modern Threats

Faculty of Information Engineering, Informatics, and Statistics
Ph.D. Program in Cybersecurity (XXXVIII cycle)

Giorgia Di Pietro

ID number 1751145

Advisor

Prof. Daniele Cono D'Elia

Co-Advisor

Prof. Leonardo Querzoni

Academic Year 2025/2026

Thesis defended on May 27, 2026

in front of a Board of Examiners composed by:

Prof. Roberto Caldelli, CNIT / Università Mercatorum, Italy

Prof. Pierluigi Perri, Università degli Studi di Milano, Italy

Prof. Valerio Schiavoni, University of Neuchâtel, Switzerland

The thesis was submitted for revision on February 28, 2026, to the following external reviewers:

Prof. Andrea Continella, University of Twente, Netherlands

Prof. Kevin Leach, Vanderbilt University, USA

Dissecting Malware Dynamics: Gleaning Behavioral and Semantic Insights for Modern Threats

Ph.D. Thesis. Sapienza University of Rome

© 2026 Giorgia Di Pietro. All rights reserved

This thesis has been typeset by L^AT_EX and the Sapthesis class.

Author's email: g.dipietro@diag.uniroma1.it

Abstract

Malware remains a pervasive and rapidly evolving threat to modern computing systems, with adversaries continuously adapting their techniques to evade defenses. This sustained arms race exposes fundamental limitations in prevailing detection paradigms, particularly those relying on low-level artifacts, isolated execution traces, or narrowly scoped behavioral heuristics. Such approaches lack principled abstraction mechanisms capable of capturing malicious intent across implementation variants, concealment strategies, and execution-dependent behavior. Advancing dynamic malware detection therefore requires abstractions that remain stable under variability while preserving discriminative power.

This thesis advances dynamic malware detection along two complementary dimensions: abstraction over technique variability and over execution variability.

First, we investigate process injection, a widely adopted evasion technique in which malicious payloads execute within the address space of legitimate processes to bypass process-level monitoring. We introduce a behavior-centric detection methodology grounded in the observation that injection variants—despite surface diversity—rely on three fundamental operational primitives: memory allocation, payload writing, and execution triggering. By detecting the correlated co-occurrence of these operations rather than technique-specific signatures, the approach generalizes across variants and raises the evasion bar. An empirical study of malware collected in the wild characterizes the prevalence and evolution of injection techniques and reveals systematic blind spots in existing analysis frameworks.

Second, we address execution-induced behavioral variability, a defining characteristic of modern malware arising from environmental adaptation, anti-analysis mechanisms, and conditional logic. Single-execution analysis therefore yields an incomplete and potentially misleading view of program intent. We introduce a multi-trace analysis framework that consolidates behavioral evidence across executions using an augmented API call graph abstraction and structure-aware exploration. We examine how aggregation, representation, and exploration interact under statistical constraints. While multi-trace aggregation improves detection sensitivity, it introduces a reliability–stability trade-off in which repeated evaluation increases exposure to false positives. Structural abstraction provides inductive bias toward connectivity and dependency patterns, while guided exploration mitigates redundant execution and limits error accumulation.

Taken together, the results demonstrate that robust malware detection cannot rely solely on incremental accuracy improvements. Instead, it requires principled abstraction mechanisms that connect low-level behavioral evidence to higher-level structural reasoning under statistical control. By combining behavior-centric detection of stealth techniques with structured multi-trace modeling, this thesis contributes to more resilient, interpretable, and statistically robust dynamic malware analysis.

Contents

1	Introduction	1
1.1	Contributions	2
1.2	Thesis Outline	4
2	Background	6
2.1	Malware Analysis	7
2.1.1	Static Analysis	8
2.1.2	Dynamic Analysis	9
2.2	Evasion and Stealth Mechanisms in Modern Malware	13
2.2.1	Anti-Analysis Techniques	14
2.2.2	Process Injection	15
2.3	Multi-path Execution for Malware Analysis	19
2.3.1	Fuzzing Fundamentals	20
2.3.2	Fuzzing for Malware Analysis	21
2.4	Machine Learning for Malware Detection	21
2.4.1	Learning Paradigms and Tasks	22
2.4.2	Feature Extraction and Representation	23
2.4.3	Classification Algorithms	25
3	State of the Art	27
3.1	Detecting Process Injection	27
3.1.1	Memory Forensics Approaches	27
3.1.2	Behavioral Tracking Approaches	28
3.1.3	Environment-Based and Data-Driven Approaches	29
3.1.4	Limitations of Existing Solutions	31
3.2	Fuzzing Applications in Malware Analysis	32
3.3	Graph-based Behavioral Representations in Malware Analysis	34
3.3.1	Methodological Limitations and Structural Gaps	35
4	Unmasking Process Injection: Systematic Detection in Windows Malware	37
4.1	Introduction	38
4.1.1	Contributions	40
4.2	Process Injection Landscape and Threat Model	40
4.2.1	Process Injection Technique Landscape	40
4.2.2	Threat Model	41

4.3	Analysis Methodology	42
4.3.1	Overview	43
4.3.2	Main Challenges	43
4.3.3	Implementation	48
4.4	Experimental Setup	51
4.4.1	Dataset	51
4.4.2	Analysis Environment	53
4.5	Experimental Findings	54
4.5.1	Adoption of Process Injection Techniques	54
4.5.2	Families and Process Injection Techniques	59
4.5.3	Feasibility of Downstream Integration	61
4.6	Discussion	65
5	Beyond Single Traces: Multi-Trace Semantic Modeling of Malware Behavior	67
5.1	Introduction	68
5.1.1	Contributions	70
5.2	Problem Definition and Threat Model	71
5.2.1	Behavioral Variability and Partial Observability	71
5.2.2	Formal Model of Execution Traces	71
5.3	Methodology	72
5.3.1	Overview	73
5.3.2	Multi-Trace Collection	74
5.3.3	Augmented Structural Representation via API Call Graphs	75
5.3.4	Structure-Aware Exploration	78
5.3.5	Implementation	79
5.4	Experimental Evaluation	84
5.4.1	Experimental Setup	84
5.4.2	Feature Extraction	85
5.4.3	Models	85
5.4.4	Behavioral Aggregation and Detection Stability	86
5.4.5	Structural Representation and Robustness	89
5.4.6	Structure-Aware Exploration Efficiency	91
5.4.7	Statistical Limits of Large-Scale Behavioral Exploration	92
5.4.8	Discussion	93
6	Conclusions and Future Work	96
6.1	Conclusions	96
6.2	Future Works	97
	Bibliography	99

List of Figures

2.1	Process injection overview	15
2.2	Comparison of Control Flow Graphs and Function Call Graphs . . .	24
4.1	Framework architecture for process injection detection	44
4.2	API trace of Process Hollowing using shared memory sections	46
4.3	Misleading background activity in API traces	47
5.1	Construction of the augmented API call graph	76
5.2	False positive rate under trace aggregation	89

List of Tables

3.1	Comparison of prior approaches for detecting process injection . . .	31
4.1	Windows APIs mapped to alloc/write/execute primitives	49
4.2	Yearly composition of the malware dataset by vhash cluster	52
4.3	Behavioral consistency of vhash-based malware clusters performing process injection	53
4.4	Yearly prevalence of process injection in the balanced malware dataset	55
4.5	Yearly distribution of process injection techniques in the balanced malware dataset	56
4.6	Top malware families ranked by number of samples	60
4.7	Process injection techniques across top malware families	61
4.8	Widely used benign applications in the evaluation	62
4.9	Wine conformance workload and runtime overhead	65
5.1	Representative instrumented Windows APIs by category	80
5.2	Linear detector performance across aggregation levels	86
5.3	Sample-level performance under trace aggregation	88
5.4	Guided exploration using the flip classifier	91

Chapter 1

Introduction

Malware, short for *malicious software*, remains one of the most pervasive threats to modern computing systems. It enables attackers to infiltrate, disrupt, or gain unauthorized control over information systems, targeting assets ranging from personal devices to critical infrastructures [91, 137]. The rapid evolution of this threat is striking: according to the AV-TEST Institute, more than 450,000 new malicious programs and potentially unwanted applications emerge every day [76]. These figures highlight not only the ubiquity of malware but also its dynamism, posing persistent challenges to security researchers and practitioners alike.

In response, decades of research have produced a rich ecosystem of malware detection and defense technologies. Signature-based methods, heuristic rules, and behavioral analysis have all played pivotal roles in improving the security and reliability of computing systems. Given the scale and complexity of the problem, malware analysis has emerged as a cornerstone of cybersecurity research—as it provides the empirical basis for understanding malicious behavior, designing countermeasures, and driving the development of automated detection systems [130].

As defense technologies advance, malware authors continuously adapt their methods, fueling a well-known arms race where concealment and detection evolve in parallel. Modern malware employs a wide range of anti-analysis and evasion strategies to maintain stealth, exploiting the assumptions and blind spots of existing solutions.

Process injection exemplifies this trend, standing out as a particularly effective technique used in real-world attacks to evade process monitoring-based defenses by executing malicious code in the address space of a separate, legitimate process [15]. This capability enables injected malicious payloads to blend seamlessly into benign system activity, thereby complicating behavior-based detection. Despite significant prior work, current detection solutions for process injection still face critical limitations that constrain their coverage and robustness. Understanding and addressing these shortcomings remains essential to enhancing the resilience of next-generation defenses against increasingly stealthy threats.

In practice, even widely adopted malware analysis tools and sandboxes—considered standard in both academia and industry—exhibit significant blind spots. As demonstrated in Chapter 4, several public analysis frameworks fail to reliably identify behaviors related to process injection.

This observation is symptomatic of a broader challenge: modern malware increasingly operates within trusted system boundaries, where malicious actions are difficult to distinguish from benign activity at the level of raw events or individual execution traces. Consequently, reliable detection cannot rely solely on low-level artifacts; it must instead be grounded in behavioral abstractions that capture the underlying semantics of malicious activity rather than known implementation traits.

This need for abstraction extends beyond the narrow goal of detecting particular techniques. As adversaries adopt evasion, anti-analysis strategies, and even deliberate randomness, malware’s observable behavior becomes context-dependent and variable across executions. Any single execution reveals only a fragment of the program’s behavioral space—a defining property of contemporary malware that undermines analysis pipelines built on the assumption that one trace faithfully represents intent.

These observations motivate treating abstraction not merely as a detection mechanism, but as a semantic representation of malware behavior across multiple executions. Rather than considering each run as definitive, analysis can model behavior as a distribution of possible manifestations, each revealing different aspects of underlying intent. This perspective calls for structured models that integrate evidence from multiple executions into coherent behavioral representations.

This thesis highlights that incremental improvements in detection accuracy alone cannot outpace adversarial evolution. Robust malware defense instead requires abstraction mechanisms explicitly designed to link low-level behavioral evidence with higher-level semantic understanding. Toward this end, this thesis pursues two complementary directions for improving the detection and analysis of Windows malware:

- **Behavioral detection** of process injection attacks, focusing on identifying malicious code that conceals its execution within legitimate processes to evade process-level defenses;
- **Semantic representation** of malware behavior, abstracting execution traces into structured multi-trace models that capture cross-execution invariants and support generalization, robust behavioral detection, and controlled exploration of decision-relevant behavior.

These directions address distinct but tightly interconnected dimensions of malware concealment. The first provides a principled foundation for detecting deceptive process-level behaviors that evade conventional tools. The second transforms this low-level evidence into structured, interpretable representations enabling reasoning across executions and variants. Together they contribute to bridging the gap between observing malicious behavior and understanding it, advancing both the theoretical and practical state of the art in malware detection research.

1.1 Contributions

This thesis advances the state of the art in malware analysis by addressing two tightly coupled challenges: the reliable detection of process injection attacks and the semantic interpretation of complex malware executions.

Unmasking Process Injection: Systematic Detection in Windows Malware

Process injection enables malware to evade detection by executing code within the trusted context of another process, hiding malicious activity under the guise of the victim process and undermining process-level defenses. Despite its prevalence in real-world threats, prior academic research has limited its focus to individual injection techniques or narrowly scoped heuristics, rather than systematic analysis.

In Chapter 4, we introduce a principled behavioral detection methodology centered on the observation that all known process injection variants share three fundamental operations: (1) memory allocation in the target process, (2) payload writing into that memory, and (3) execution triggering [86]. Rather than precisely recognizing every individual step, the proposed approach detects the co-occurrence of a *minimal subset* (at least two) of these operations, correlating them via memory address identity and equivalence classes to account for implementation variability.

This design shifts detection from fragile, technique-specific signatures—hallmarks of prior work—to a behaviorally grounded abstraction. Attackers are thus forced to implement not one, but two fundamental operations in unforeseen ways—an intrinsically harder task that significantly raises the evasion bar—while defenders benefit from improved robustness and easier adaptation as new variants emerge.

We validate the approach on malware samples collected in the wild from 2017 to 2023, comprising 56,340 representative samples from 2,667 malware families. The evaluation addresses three main research questions: (1) which process injection techniques are most commonly adopted by modern malware, and how does public release of information influence their adoption; (2) whether malware families exhibit characteristic preferences for specific techniques over time; and (3) whether the proposed methodology can suit downstream security solutions (e.g., false positives).

The results confirm process injection as a dominant and evolving strategy, with threat actors continuously adapting their choices in response to defensive pressure and community knowledge. Comparative analysis shows that the proposed method consistently outperforms dedicated solutions and mainstream sandboxes, correcting earlier underestimates of prevalence due to strict pattern matching and lack of anti-evasion provisions in prior work.

Beyond Single Traces: Multi-Trace Semantic Modeling of Malware Behavior

Behavioral variability enables malware to evade detection by exhibiting different operational facets across executions. Traditional dynamic analysis relies on a single-execution paradigm, which reveals only one facet of a program’s behavioral space. Adversaries craft programs that adapt to system configuration, timing constraints, internal logic, and environmental feedback. As a result, detection decisions derived from isolated traces may be unstable and fail to generalize across execution contexts.

In Chapter 5, we investigate the implications of execution variability for machine-learning-based malware detection. Rather than treating additional executions as uniformly beneficial, we examine how behavioral evidence should be aggregated, represented, and explored under statistical constraints. The central challenge is not merely to collect more traces, but to reason about a distribution of traces that

constitute partial observations of a latent behavioral space.

To this end, we introduce a multi-trace analysis framework that consolidates behavioral evidence across executions through an augmented API call graph abstraction and a structure-aware exploration strategy. The graph representation merges information observed across runs while normalizing incidental artifacts that do not reflect semantic intent. Exploration is regulated through classifier feedback and threshold-based stopping, enabling controlled expansion under bounded resources.

Through a controlled empirical study, we investigate four central questions: (1) how aggregating behavioral evidence across multiple executions influences detection sensitivity and statistical stability; (2) how structural representation affects detection robustness and generalization across behavioral variants; (3) whether structure-aware exploration reduces redundant behavioral testing while preserving detection effectiveness; and (4) which statistical phenomena emerge under large-scale behavioral exploration and how they constrain scalability.

Using a dataset of 958 samples—half real malware and half benign software—from which approximately 34,800 dynamic execution traces were collected, we show that multi-trace analysis introduces a reliability–stability trade-off. While aggregation improves detection sensitivity, it can also induce false positive growth under repeated testing. Structural abstraction, in contrast, imposes an inductive bias toward connectivity and dependency, supporting improved robustness under behavioral diversity. Finally, exploration policy materially affects both computational cost and statistical stability, demonstrating that scalability depends not merely on increased behavioral coverage, but on principled evidence integration.

Taken together, these findings indicate that multi-trace analysis is necessary but not sufficient for robust behavioral detection. Scalable malware detection requires joint consideration of aggregation rules, representation choices, and exploration policy under explicit statistical constraints. Detection quality does not scale monotonically with execution count; instead, it depends on how behavioral evidence is structured and integrated across executions.

In contrast to the large-scale benchmarking setting of Chapter 4, the evaluation in this chapter is intentionally focused on controlled investigation rather than exhaustive corpus-scale validation. Because each sample requires repeated executions and cross-trace consolidation, the computational cost per analyzed program is substantially higher than in conventional single-execution analysis. Consequently, the study prioritizes depth of behavioral characterization over corpus breadth in order to expose and analyze the statistical and structural phenomena underlying multi-trace detection. Larger-scale validation remains an important direction for future work.

1.2 Thesis Outline

This thesis is structured as follows:

- **Chapter 2** introduces the background required to contextualize this work, including malware analysis paradigms and their trade-offs, attacker evasion and stealth mechanisms with a focus on process injection, multi-path malware execution through fuzzing, and an overview of machine learning fundamentals for malware detection.

- **Chapter 3** reviews the state of the art in three key areas addressed by this thesis: detection of process injection attacks, fuzzing-based exploration of multi-path behavior in malware, and graph-based representations for malware detection. The chapter critically analyzes existing approaches, identifies their limitations and open gaps, and motivates the need for behavior-centric detection and structured multi-trace reasoning.
- **Chapter 4** presents the first core contribution of this thesis, also described in the paper *Can You Run My Code? A Close Look at Process Injection in Windows Malware* [53], published at the 20th ACM Asia Conference on Computer and Communications Security (ASIA CCS 2025). It introduces a principled methodology for detecting process injection through the co-occurrence of three fundamental operations—memory allocation, payload writing, and execution triggering—and provides an empirical analysis of their prevalence, evolution, and adaptation in response to defensive pressure and public disclosure.
- **Chapter 5** reframes dynamic malware analysis under execution variability, examining how behavioral evidence should be represented and integrated across multiple runs. Developed in collaboration with Texas A&M University during a visiting research period, it presents a multi-trace analysis framework that combines augmented API call graph abstraction with structure-aware exploration. The chapter analyzes how behavioral aggregation, structural representation, and exploration policy interact under statistical constraints.
- **Chapter 6** summarizes the contributions of the thesis, reflects on their implications for robust malware detection as well as human-centered analysis, and identifies concrete directions for future research.

Chapter 2

Background

Malware is a generic term for intrusive software intentionally designed to infiltrate, disrupt, or gain unauthorized control over computer systems, servers, or networks. It spans a broad range of threats—including spyware, viruses, ransomware, and Trojan horses—that exploit software or system vulnerabilities to achieve objectives such as data exfiltration, persistent unauthorized access, or extortion via encryption [31, 131]. Unlike conventional software, malware is explicitly adversarial: it is designed not only to execute malicious functionality, but also to evade detection and resist analysis.

The rapid growth in malware volume and diversity poses significant challenges to modern detection systems. As computing platforms and defensive technologies evolve, adversaries respond with increasingly sophisticated techniques that increase operational complexity and expand the attack surface for next-generation malware.

Traditional antivirus (AV) systems rely primarily on matching stable syntactic patterns (e.g., byte sequences or hash values) against extensive databases of known threats. While highly effective for previously observed samples, this signature-based paradigm struggles in the presence of novel variants. Polymorphic and metamorphic malware, for instance, preserve semantic functionality while substantially altering their binary representation, thereby defeating purely pattern-based matching. Consequently, modern defense mechanisms increasingly incorporate behavior-oriented techniques that capture higher-level execution semantics rather than superficial syntactic characteristics [6].

Behavior-based detection relies on runtime observations obtained through controlled program execution, including Application Programming Interface (API) invocations, system calls, process activity, file system and registry modifications, and network interactions. These dynamic signals are generally harder to conceal than static artifacts and provide a natural input for statistical and machine learning (ML) models. By moving beyond exact signatures, such models can detect previously unseen malware families and zero-day threats. However, malware frequently incorporates anti-analysis mechanisms—such as packing, control-flow obfuscation, encryption, anti-debugging techniques, and virtual machine (VM) detection—that attempt to hide internal logic or suppress malicious behavior under observation [97, 148].

Malware detection is the process of determining whether a given program exhibits malicious intent or not [6], and represents the first line of defense in protecting information systems. Malware analysis underpins this effort by providing

the methodological foundation for this task. Through systematic examination of program structure and execution, analysis enables defenders to reconstruct operational semantics, infer attacker intent, and derive robust behavioral indicators that generalize beyond individual samples.

At the same time, the analysis process itself has grown increasingly complex. New-generation malware often spans multiple processes, dynamically injects code into remote address spaces, and conditionally activates payloads depending on environmental stimuli. These characteristics challenge both purely static inspection and naive runtime monitoring. Addressing such complexity requires principled program analysis techniques capable of capturing fine-grained execution behavior in adversarial settings. In particular, dynamic program analysis and Dynamic Binary Instrumentation (DBI) have emerged as foundational technologies for obtaining precise runtime visibility into binary execution without requiring source code access.

This chapter provides the conceptual and technical foundations necessary to understand the contributions of this thesis on process injection detection and structured malware representation. Specifically, it: (1) introduces malware analysis paradigms and their trade-offs, including static and dynamic approaches, as well as the program analysis techniques that underpin runtime behavioral inspection; (2) surveys prevalent evasion and stealth mechanisms in modern malware, with particular emphasis on process injection; (3) discusses multi-path malware execution through fuzzing; and (4) outlines machine learning fundamentals for malware detection.

2.1 Malware Analysis

Malware analysis refers to the systematic examination of malicious software to understand its structure, functionality, and intended behavior, as well as to reconstruct the sequence of actions performed during an incident. This knowledge supports both preventive defenses (e.g., detection rules and behavioral models) and reactive security capabilities (e.g., incident response and forensic investigation) [31, 131].

Malware analysis can be leveraged to derive two broad categories of detection signatures: *host-based* and *network-based* [131]. Host-based signatures capture observable effects on a system, such as files created or modified by malware, configuration changes, or registry modifications. Unlike traditional antivirus signatures that primarily rely on static code characteristics, host-based signatures emphasize system-level effects and are therefore more resilient to polymorphism and packing techniques. Network-based signatures, in contrast, identify malicious activity through inspection of network traffic, including anomalous communication patterns or command-and-control behavior. These signatures are widely employed by Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), and network security appliances to detect or block malicious communications in real time.

In many investigative scenarios, analysts are provided solely with a compiled binary, which is inherently opaque and non-human-readable. To derive semantic meaning, they employ analytical techniques that expose different aspects of the malware’s internal logic. Broadly, malware analysis is divided into two fundamental approaches: *static* and *dynamic*. Static analysis examines the malware without executing it, whereas dynamic analysis observes its runtime behavior during execution

in a controlled environment. These approaches are complementary: static analysis reveals structural properties and potential capabilities, while dynamic analysis exposes concrete runtime behavior. In practice, analysis pipelines often combine both approaches (*hybrid analysis*), using static inspection for rapid triage and hypothesis formation, and dynamic execution for in-depth behavioral profiling.

The Windows ecosystem. Microsoft Windows remains the primary target of malware campaigns, accounting for approximately 67.6% of the desktop operating system (OS) market as of January 2026 [70]. Its widespread adoption in enterprise environments, combined with extensive backward compatibility and the relative simplicity of the Portable Executable (PE) format, makes it an attractive platform for adversaries seeking broad distribution and impact.

Despite the prevalence of 64-bit architectures, many malware samples continue to be distributed as 32-bit PE binaries. Targeting PE32 maximizes compatibility across legacy systems and allows attackers to leverage mature packing and obfuscation toolchains [141]. Moreover, the proprietary nature of Windows internals, together with the limited awareness of low-level system behavior among users, complicates defensive research, while simultaneously offering attackers opportunities to exploit undocumented behaviors and legacy features.

2.1.1 Static Analysis

Static analysis examines a malware program without executing its code [31, 130, 131]. Instead of observing runtime behavior, analysts inspect intrinsic properties of the binary to infer likely functionality and intent.

Typical static analysis workflows begin with file format parsing (e.g., PE, ELF) to extract headers, sections, imports, exports, and compilation artifacts that reveal the target platform and potential capabilities. Analysts then perform string and resource extraction to identify URLs, commands, cryptographic keys, or anti-analysis markers embedded in the binary. Finally, code-level disassembly reconstructs control-flow and data-flow information for reasoning about algorithms and evasion tactics.

In operational settings, static analysis frequently relies on rule-based specification languages to formalize detection knowledge. For example, YARA [147] allows analysts to define structured rules that match byte patterns, strings, and structural file properties, thereby supporting large-scale scanning and threat intelligence correlation. While such rules are primarily syntactic in nature, they provide a flexible mechanism for capturing expert knowledge about malware families and reuse patterns. More recently, capability-oriented frameworks such as Capa [101] aim to infer higher-level behaviors directly from disassembled code by recognizing instruction sequences and API usage patterns associated with specific functionality (e.g., process injection, cryptographic routines, or persistence mechanisms). By abstracting low-level artifacts into semantic capabilities, these tools illustrate how static analysis can extend beyond simple signature matching toward structured behavioral interpretation of binaries.

A key advantage of static analysis is that it requires no code execution and thus can be performed safely without a live execution environment. This makes static techniques ideally suited for large-scale automated scanning and signature generation, as they can rapidly process vast numbers of samples with minimal

operational risk. Recent advances—including byte-level similarity analysis, opcode pattern extraction, and machine learning over control-flow graph features—have further enhanced scalability for enterprise deployment.

Nevertheless, static analysis has inherent limitations. Modern malware frequently employs packing, encryption, and sophisticated obfuscation to conceal its true semantics, preventing accurate reconstruction of runtime behavior. Purely static approaches may thus overlook critical functionality that only manifests during execution, motivating complementary dynamic analysis [48, 88, 107].

2.1.2 Dynamic Analysis

Dynamic analysis executes malware in controlled environments to observe its runtime behavior, system interactions, and externally visible side effects [31, 62, 130, 131]. Unlike static inspection, which reasons about potential capabilities from binary structure, dynamic analysis captures concrete execution semantics as they manifest during program execution. This enables analysts to reconstruct logic and behavioral artifacts that may remain obscured in packed or obfuscated binaries.

To mitigate operational risk, malware execution is typically performed within isolated environments such as virtual machines or sandbox systems. During execution, analysts employ monitoring tools to observe indicators of compromise (IoCs) and behavioral traces. Common instruments include process and system monitors (e.g., Process Explorer), network analyzers (e.g., Wireshark), and interactive debuggers (e.g., IDA Pro).

More fundamentally, dynamic malware analysis can be framed as a specialized instance of *dynamic program analysis*. Rather than merely recording observable side effects, dynamic program analysis instruments program execution to obtain fine-grained visibility into control flow, memory accesses, and interactions with the operating system. By operating directly on executing binaries, it enables reconstruction of execution semantics even in the absence of source code. This perspective is particularly important in adversarial settings, where malicious logic may be conditionally activated, environment-dependent, or deliberately concealed from static inspection.

Advanced analysis pipelines therefore rely on runtime instrumentation mechanisms to collect detailed execution traces, including instruction-level events, memory operations, and API or system call sequences. Such traces provide a structured representation of program behavior that can be analyzed for security properties, anomaly detection, or higher-level capability inference.

Despite its strengths, dynamic analysis demands expertise in operating system internals, debugging techniques, and sandbox configuration, and remains susceptible to anti-analysis mechanisms such as virtualization detection or instrumentation evasion, which may cause malware to alter its behavior accordingly [97].

Dynamic Program Analysis

Formally, dynamic program analysis studies software behavior by observing execution and leveraging runtime information to reason about semantic properties. By

operating on execution traces rather than static approximations, it achieves higher precision at the cost of instrumentation overhead and performance degradation.

Representative techniques include testing, profiling, runtime monitoring, and trace-based analysis. These approaches execute programs with selected inputs and collect execution data to build behavioral models grounded in observed semantics.

In the context of malware analysis, dynamic program analysis assumes particular importance. Malicious logic is frequently conditionally triggered, environment-dependent, or intentionally concealed through obfuscation and packing. Runtime observation therefore provides a principled mechanism for exposing behaviors that may remain opaque under purely static inspection.

To enable systematic and fine-grained runtime observation, dynamic analysis frameworks rely on instrumentation mechanisms capable of intercepting and augmenting execution. In practice, many security-oriented analyses are built atop DBI frameworks, which provide instruction-level visibility to registers, memory state, control-flow transfers, and operating system interactions without requiring source code access.

Behavioral interposition mechanisms. Among the techniques enabled by runtime instrumentation, execution interposition plays a central role in dynamic malware analysis. More generally, execution interposition refers to the ability to intercept and augment selected program interactions at runtime. By mediating interactions with the operating system or external environment, analysts can observe behavioral artifacts such as file operations, network communication, process creation, and memory manipulation.

A program’s behavior can often be effectively characterized by the sequence of interactions it performs with its execution environment and the associated arguments [25]. This insight underpins numerous behavior-based detection systems and motivates the adoption of runtime interposition techniques in this thesis.

Interposition can be implemented at multiple abstraction layers, including userspace API hooks, kernel-level interception mechanisms, or DBI-based wrapping. Each approach entails trade-offs in transparency, robustness, and performance overhead. Lower-level mechanisms provide comprehensive visibility into operating-system-level activity, whereas higher-level interception offers access to semantically richer abstractions of function invocations while avoiding the complexity and intrusiveness of kernel-level mechanisms.

Dynamic Binary Instrumentation

In malware analysis, obtaining reliable and fine-grained visibility into program execution is essential, as many malicious behaviors manifest only at runtime. This objective is achieved through instrumentation, i.e., the insertion of additional instructions into a program to observe, record, or modify its execution.

DBI frameworks have emerged as a dominant paradigm for runtime program analysis. They enable transparent observation and modification of execution at the instruction level without requiring access to source code, recompilation, or changes to the operating system or hardware. As a result, DBI has become a cornerstone of

modern malware research, underpinning both academic prototypes [53] and practical analysis tools [83].

Instrumentation can be performed either statically (at compile time) or dynamically (at runtime). Static instrumentation inserts analysis code before execution at the source or binary level, whereas dynamic instrumentation introduces probes and user-defined routines during execution without modifying the original binary on disk. Because malware is typically distributed in binary form and frequently obfuscated, this thesis relies exclusively on dynamic instrumentation.

Conceptually, a DBI system can be defined as an execution runtime running in user space for process-level virtualization [51]. It interprets the Instruction Set Architecture (ISA) of the target platform while providing flexible instrumentation capabilities that enable analysis tools to observe, inspect, and modify program instructions and data during execution.

Rather than allowing native execution, the DBI framework takes control of the program, translates instructions into an internal representation, optionally augments them with analysis routines, and emits instrumented native code for execution. This architecture provides comprehensive visibility into program behavior while preserving execution transparency and maintaining compatibility with the underlying operating system. At the same time, it enables fine-grained access to the architectural state of the target program, including registers, memory contents, stack frames, and control-flow transfers, thereby supporting behavioral inspection and manipulation.

Most DBI frameworks rely on just-in-time (JIT) compilation. When execution reaches previously unseen code, the DBI engine intercepts, decodes, instruments, and recompiles it into native instructions stored in a code cache. A dispatcher manages control transfers between cached fragments and newly translated regions. Frequently executed paths may be combined into larger traces to reduce overhead.

The fundamental compilation unit in DBI systems is typically a trace, defined as a straight-line sequence of instructions ending with an unconditional control transfer and possibly including side exits for conditional branches.

In practice, the DBI engine continuously builds, instruments, and caches these traces on demand. Unlike traditional ahead-of-time compilers, this process does not optimize source code but dynamically rewrites binaries to facilitate analysis.

From the analyst’s perspective, DBI frameworks expose general-purpose APIs that support the development of analysis clients (also called DBI tools) executing alongside the target program. Such clients may inspect or modify registers, replace instructions, alter function arguments or return values, or redirect execution to user-defined routines. This flexibility makes DBI a powerful foundation for malware analysis and other security research tasks.

Intel Pin. Intel Pin [99] is one of the most widely adopted DBI frameworks, particularly in malware analysis and security research. Pin operates as a user-mode runtime system that transparently interposes between a target application and the operating system, taking control of execution without modifying the original binary on disk. Rather than executing application code natively, Pin dynamically translates instructions into a private, writable code cache where instrumentation logic is inserted before execution.

Pin employs a JIT compilation model in which code is translated trace by trace. Before a trace is emitted into the code cache, user-defined analysis routines are invoked to specify instrumentation that is incorporated into the generated code. During execution, Pin preserves architectural state by automatically saving and restoring registers and providing contextual information (e.g., register values, memory addresses, and system call parameters) to analysis callbacks. This design enables observation and, when necessary, modification of execution while maintaining transparency with respect to the application’s behavior.

The framework supports multiple instrumentation granularities, including instruction, basic block, trace, routine, and image levels, enabling a wide range of analyses such as dynamic profiling, taint tracking, coverage measurement, and fine-grained malware behavior tracing without requiring source code access or recompilation.

Pin supports IA-32, x86-64, and ARM architectures and operates entirely in user space. Although it is closed-source—a point sometimes criticized in the research community—it remains widely adopted due to its stability, performance, and extensive ecosystem of existing analysis tools. For this thesis, Intel Pin was selected as the underlying DBI framework for implementing the proposed research prototypes.

Microsoft Detours. For scenarios in which only API-level interception is required on the Windows platform, Microsoft Detours [40] provides a lightweight alternative to full-fledged DBI frameworks. Detours is a proprietary Microsoft library that enables dynamic interception of arbitrary Win32 functions within a process, allowing instrumentation logic to be inserted without modifying binaries on disk.

Detours operates by rewriting the in-memory image of a target function at runtime. Specifically, it replaces the function’s entry instructions with an unconditional jump to a user-defined detour routine. The displaced instructions are preserved in a dynamically generated trampoline, which subsequently transfers control back to the remainder of the original function. This mechanism preserves the semantics of the intercepted function while enabling pre- and post-processing around its execution.

Unlike DBI frameworks, which instrument execution at the instruction or trace level through dynamic translation, Detours operates at the function level and does not virtualize execution. Consequently, it offers lower overhead and simpler deployment, making it well suited for lightweight API tracing and monitoring. However, because interception relies on explicit code rewriting, it provides reduced transparency and is more susceptible to detection or interference by defensive mitigations.

Despite these limitations, Detours remains widely used in Windows-based security research and malware analysis [53, 72], particularly in scenarios where fine-grained instruction-level visibility is unnecessary.

Virtual Analysis Environments

Dynamic malware analysis requires controlled execution environments that prevent harm to real systems. Executing malicious code directly on a regular workstation risks unpredictable and potentially irreversible consequences. Similarly, analyzing malware on a dedicated standalone machine that must be reinstalled after each experiment is operationally impractical due to substantial time and maintenance overhead [29]. For these reasons, analysts rely on virtual analysis environments

that allow malware to execute in isolation while providing reproducibility and instrumentation for safe experimentation.

Such environments are typically implemented using virtual machines or sandboxing frameworks built atop a virtualization layer. This layer abstracts the underlying hardware and emulates a realistic operating system while maintaining strict separation from the host. As a result, malware executes as if running on a native system, yet any modifications remain confined to the isolated environment. This containment prevents lateral movement, unintended data exfiltration to real networks, and damage to the analyst's infrastructure.

A major advantage of virtualization is the ability to rapidly restore the system to its original or previous clean state. Analysts can create snapshots representing a trusted baseline prior to execution. After a dynamic analysis run, the compromised disk image can be discarded and replaced with a clean snapshot, enabling near-instant restoration [29]. This capability is crucial because malware frequently modifies system settings, installs persistence mechanisms, or corrupts files to achieve its objectives. Compared to reinstalling a physical system, snapshot-based recovery is significantly more efficient and practical.

From a performance perspective, modern virtualization technologies introduce modest performance overhead, enabling malware to execute at near-native speed. This ensures that observed behavior remains realistic while preserving the safety and containment guarantees required for large-scale experimentation.

Despite these advantages, virtual environments have inherent limitations. Many modern malware families include anti-virtualization techniques designed to detect execution within a VM and evade analysis. Such techniques may involve checking for virtualization artifacts, measuring timing discrepancies, or inspecting hardware and system properties indicative of a hypervisor (see Section 2.2.1). If detection succeeds, malware may alter behavior, remain dormant, or terminate prematurely.

Notwithstanding these challenges, virtualized analysis platforms remain indispensable for dynamic malware research. They enable systematic observation of interactions between applications and the operating system, including system calls, API usage, and other low-level behavioral signals, while maintaining a practical balance between realism, safety, and observability. Accordingly, all experimental evaluations in this thesis are conducted within controlled virtualized environments.

2.2 Evasion and Stealth Mechanisms in Modern Malware

Despite significant advances in defensive tooling, malware remains a pervasive and rapidly evolving threat. Static and dynamic analysis, sandboxing, and endpoint monitoring have raised the technical bar for attackers. Nevertheless, both the volume and complexity of malware incidents continue to increase. This persistence is largely driven by the systematic adoption of techniques designed to hinder inspection and evade detection, thereby concealing malicious behavior during analysis [25, 87].

Modern malware ecosystems encompass a broad spectrum of challenges, including command-and-control (C2) infrastructures, staged payload delivery, persistence strategies, privilege escalation, lateral movement, and data exfiltration. While each

of these components plays a critical role in real-world campaigns, this thesis focuses on techniques that undermine analysis processes and repurpose trusted operating system functionalities to achieve stealthy execution. These aspects directly underpin the contributions developed in the subsequent chapters.

Within this scope, two prominent and complementary trends warrant closer examination. First, malware increasingly incorporates mechanisms that identify analysis environments and suppress or delay malicious activity until execution occurs on a genuine victim system. Second, attackers routinely abuse legitimate system primitives—originally designed for benign functionality—to bypass security controls and blend malicious operations with normal system behavior. Together, these strategies significantly reduce the effectiveness of both automated systems and manual investigative workflows.

The remainder of this section examines these developments from both defensive and offensive perspectives. We first discuss anti-analysis and evasion strategies that challenge traditional inspection frameworks, and then consider process injection as a representative and widely adopted technique through which native system mechanisms are leveraged for covert execution.

2.2.1 Anti-Analysis Techniques

Malware rarely relies solely on malicious functionality; it also employs mechanisms specifically designed to conceal its operation and avoid detection. These *anti-analysis techniques* aim to obscure intent from users, security systems, and analysts, thereby increasing the likelihood of successful infection and sustained persistence.

Early evasion strategies primarily relied on packing, obfuscation, and encryption to frustrate static inspection and reverse engineering. Obfuscation transforms code into syntactically valid yet semantically opaque representations, complicating structural reasoning. Encryption stores the payload in protected form and reconstructs it at runtime via a decryption stub. Polymorphism extends this approach by generating syntactically distinct variants with differing decryption routines while preserving identical semantics, undermining signature-based detection.

As static defenses evolved, attackers increasingly shifted their focus toward dynamic analysis platforms such as sandboxes, virtual machines, and instrumented execution frameworks. Because behavioral monitoring enables analysts to infer intent and reconstruct attack logic, modern malware integrates mechanisms that detect, evade, or actively interfere with monitored execution contexts [37, 67, 100].

A prominent class of techniques leverages environmental fingerprinting. Malware inspects artifacts associated with virtualization, emulation, or debugging to determine whether it is executing inside a controlled setting [23]. When such indicators are detected, execution may be suppressed, delayed, or redirected toward benign behavior. Activation logic is often conditioned on host-specific properties—such as hardware configuration, installed software, locale, or domain membership—that are more likely to be present on real victim machines than analysis environments [38]. Additionally, timing- and side-channel probes exploit latency discrepancies introduced by instrumentation to infer observation.

Samples exhibiting such conditional behavior are commonly referred to as *environment-sensitive* or *evasive*. Rather than executing malicious logic uncon-

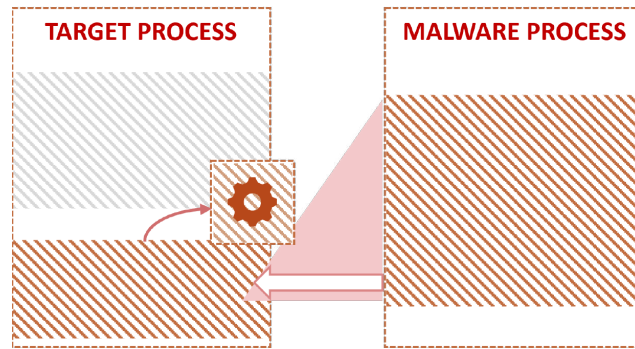


Figure 2.1. Simplified representation of process injection. A malware-controlled process allocates or modifies memory within a separate target process and triggers execution of injected code inside the target’s address space.

ditionally, they gate payload activation on attacker-defined predicates derived from runtime observations [25]. Prior work has studied this phenomenon from multiple perspectives, including sandbox evasion [97], targeted activation logic [146], and hypervisor fingerprinting [60], highlighting the breadth and sophistication of mechanisms available to adversaries.

In practice, evasive malware combines multiple heterogeneous checks, ranging from virtualization artifacts and hardware characteristics to system metadata, debugging indicators, and high-resolution timing discrepancies, into layered decision logic. These checks are frequently interleaved with payload code, fragmenting execution paths and reducing the reliability of single-run dynamic analysis. When the environment is identified as monitored, execution may stall, terminate, or expose only benign behavior, misleading both automated detection systems and human investigators [46, 121].

To counter these techniques, defensive approaches attempt to increase transparency by masking artifacts or emulating realistic host characteristics [60, 85]. However, recent evidence [38] demonstrates that artifact removal alone is insufficient: malware can still discriminate analysis environments using context-dependent or semantically meaningful host attributes that cannot be arbitrarily altered without compromising realism. As a result, effective analysis increasingly requires advanced techniques capable of exposing hidden behaviors or reasoning about activation predicates, rather than relying solely on passive behavioral observation.

2.2.2 Process Injection

Among the most impactful stealth mechanisms in modern malware is *process injection*, also referred to as *code injection*. This attack paradigm enables adversaries to execute malicious code within the address space of another process, typically one that is benign or trusted by the operating system [15, 86]. Figure 2.1 presents a simplified conceptual view of this interaction.

By shifting execution into a separate process, malware can obscure its true origin and cause malicious activity to appear as if it was initiated by a legitimate application. This can significantly reduce the likelihood of detection by host-based defenses such as antivirus and Endpoint Detection and Response (EDR) systems.

Depending on the variant, injected code may further facilitate persistence (e.g., via registry or service manipulation) or enable privilege escalation to maintain an attacker's foothold across system reboots [138].

At a conceptual level, process injection involves transferring attacker-controlled code into the virtual address space of a target process and triggering its execution within that context. Once active, the injected logic can inherit the privileges, trust relationships, and behavioral profile of the victim process. In some cases, this enables the circumvention of allow-listing policies and weakens process-based trust assumptions.

Klein et al. [86] classify Windows process injection techniques into three broad categories. *Process spawning* techniques create a new instance of a legitimate executable and modify it before execution; although straightforward, these approaches are typically noisy and more likely to appear suspicious. *Injection during process initialization* introduces malicious code as the process begins execution, often requiring User Account Control (UAC) [57] elevation due to interactions with privileged resources. Finally, *injecting into running processes* (often referred to as “true process injection”) targets already executing processes and is generally more stealthy, as it avoids suspicious process creation events. Prior to injection, malware typically enumerates active processes on the system to identify suitable victims.

Despite the diversity of concrete realizations, most process injection techniques follow three main steps [86]:

1. Allocating one or more memory regions within the target process;
2. Writing the malicious payload into the allocated region(s);
3. Scheduling execution to the injected content within the victim's context.

The mechanisms used to implement each step define the observable characteristics and stealth properties of a given variant. Listing 2.1 presents a simplified illustration of classic shellcode-based process injection on Windows.

Process injection is also central to so-called *fileless* attacks, in which malicious logic resides primarily in memory and leaves minimal or no artifacts on disk [4, 81]. Such attacks represent a significant and growing trend in the contemporary threat landscape [98]. Because they leverage legitimate system mechanisms and operate largely in volatile memory, injection-based techniques pose significant challenges to traditional detection approaches.

A wide range of concrete injection strategies has been documented in both academic and industry literature. While they differ in trigger mechanism and implementation detail, they share the same operational core described above. Representative examples include:

- *Shellcode Injection* [113], which writes raw shellcode into memory allocated within the virtual address space of a target process, and executes it via a newly created thread.
- *PE Injection* [19], which writes an entire PE image into memory allocated in the virtual address space of a target process, and transfers execution to its entry point via a new thread.

- *DLL Injection from disk* [17], which writes the path of a malicious Dynamic-Link Library (DLL) into memory allocated within the virtual address space of a target process, and triggers its loading via a remote invocation of `LoadLibrary`. The semantics of this API prescribe that the requested DLL must be stored on, and thus read from, disk.
- *APC Injection* [16], which schedules injected code for execution by attaching it to the Asynchronous Procedure Call (APC) queue of an existing victim thread when it enters an alertable state.
- *Process Hollowing* [21], which creates a victim process in a suspended state and, as shown in Chapter 4, may optionally unmap its original executable image from memory, replaces it with malicious code, and resumes execution from the injected entry point.
- *Thread Execution Hijacking* [22], which suspends a running thread in the target process, modifies its execution context to point to injected code, and resumes execution.
- *Early Bird* [115], which combines suspended process creation with APC-based execution to trigger malicious code at an early execution stage.
- *Hook Injection* [117], which registers Windows hook callbacks that cause a target process to load and execute attacker-controlled code. Such code is hosted in a DLL that the sample initially loads in itself, and the registered hook will eventually cause the victim process to load it as well.
- *Atom Bombing* [123], which abuses the global atom table to write a malicious payload in the virtual address space of the target process. The injected code is then executed by attaching it to the APC queue of a victim thread.
- *Extra Window Memory Injection* [18], which leverages Extra Window Memory allocated during window class registration to store and execute injected code in a victim process’s context.
- *Process Doppelgänger* [20, 96], which exploits Windows Transactional NTFS (TxFS) [54] to create and execute an executable modified image without committing visible disk changes.
- *Passive Injection*, which manipulates specific registry-based loading mechanisms (e.g., `AppCertDlls` [9], `AppInit_Dlls` [10], Application Shimming [11], COM Hijacking [12], and Image File Execution Options [13]) to induce other processes to load attacker-supplied libraries.

While individual APIs and low-level primitives evolve across Windows versions, the underlying operational semantics of process injection remain comparatively stable: cross-process memory modification followed by execution redirection under inherited context. Detection strategies tied narrowly to specific API sequences are therefore fragile, whereas approaches grounded in modeling these core semantic behaviors offer stronger resilience against implementation diversity and evasive

adaptation. This observation forms a central premise of the methodology developed in this thesis.

```
1  int main(int argc, char* argv[]) {
2
3      // Attacker-controlled payload (shellcode)
4      unsigned char shellcode[] = {
5          /* malicious machine code bytes */
6      };
7
8      // Target process identifier supplied as argument
9      DWORD targetPid = (DWORD)atoi(argv[1]);
10
11     // Obtain a handle to the target process
12     HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
13                                     targetPid);
14
15     // Step 1: Allocate executable memory inside the target process
16     LPVOID remoteBuffer = VirtualAllocEx(
17         hProcess,
18         NULL,
19         sizeof(shellcode),
20         MEM_COMMIT | MEM_RESERVE,
21         PAGE_EXECUTE_READWRITE
22     );
23
24     // Step 2: Write the payload into the allocated memory region
25     WriteProcessMemory(
26         hProcess,
27         remoteBuffer,
28         shellcode,
29         sizeof(shellcode),
30         NULL
31     );
32
33     // Step 3: Execute the injected code by creating a remote
34     // thread
35     HANDLE hThread = CreateRemoteThread(
36         hProcess,
37         NULL,
38         0,
39         (LPTHREAD_START_ROUTINE)remoteBuffer,
40         NULL,
41         0,
42         NULL
43     );
44
45     // Cleanup
46     CloseHandle(hThread);
47     CloseHandle(hProcess);
48
49     return 0;
50 }
```

Listing 2.1. Simplified example of Shellcode Injection on Windows. The code illustrates the canonical steps involved in remote thread injection. Error handling and security checks are omitted for clarity.

2.3 Multi-path Execution for Malware Analysis

Understanding and modeling malware behavior is fundamental to a wide range of security applications. This typically involves executing samples inside an instrumented environment to collect system- and API-level traces, which are then analyzed to reconstruct runtime behavior. Dynamic malware analysis has traditionally relied on single-run sandbox executions, where a sample is observed under a fixed set of environmental and execution conditions. While this approach is efficient and widely adopted in practice, it provides only a limited view of the program’s behavior.

This limitation becomes fundamental when malware embeds path-dependent logic or environment-sensitive checks [24]. In these cases, behavior may vary significantly across executions, and any single run exposes only a subset of the sample’s full functionality. The partial observability constrains the completeness of behavioral profiles and can negatively affect downstream tasks such as detection or clustering.

Early research explicitly addressed this challenge by attempting to systematically explore alternative execution paths. Moser et al. [106] pioneered multi-path exploration techniques that leverage dynamic taint analysis to identify environment- and input-dependent branches and expose hidden behaviors. In parallel, symbolic and concolic execution approaches (e.g., [41]) attempted to derive path constraints and synthesize concrete inputs capable of triggering alternative paths. Later, Peng et al.’s X-Force [120] introduced forced execution as an alternative to bypass missing inputs and environment dependencies; its successor PMP [149] refined this strategy with more principled path management.

Despite demonstrating that systematic behavioral diversification is feasible, these approaches face structural limitations that hinder their practical adoption. Symbolic and concolic techniques suffer from path explosion and constraint-solving overhead, which severely limit scalability on real-world malware. All approaches further rely on accurate semantic modeling of instructions, libraries, and system interactions [34], assumptions that are fragile in adversarial settings and difficult to maintain at scale. Forced execution explicitly sacrifices soundness, while symbolic reasoning often remains incomplete in practice due to unsupported semantics and solver complexity. Moreover, taint- and constraint-based analyses can be actively subverted by anti-analysis logic designed to mislead or neutralize information-flow tracking [43]. Other specialized systems (e.g., [85, 97]) further demonstrated per-sample behavioral variability under changing environmental conditions, reinforcing the need for scalable multi-path exploration.

Taken together, this body of work establishes both the necessity and the difficulty of systematic multi-path exploration in malware analysis. In this context, modern fuzzing-based techniques have emerged as a promising direction for driving large-scale behavioral exploration in malware analysis [34, 38, 72]. Rather than explicitly solving path constraints, fuzzing frames path discovery as an optimization problem, enabling scalable exploration across diverse samples. In this section, we provide a brief overview of fuzzing fundamentals and subsequently discuss their application to the malware domain.

2.3.1 Fuzzing Fundamentals

Fuzzing is one of the most widely adopted automated testing techniques for discovering software vulnerabilities. At a high level, fuzzing consists of repeatedly executing a program under test (PUT) with many automatically generated inputs (test cases) that are syntactically or semantically malformed. The goal is to induce unexpected behaviors such as crashes, memory safety violations, logic errors, or other anomalous conditions [102]. Test cases that exercise new program behavior are retained as seeds for further mutation, while redundant or uninformative inputs are discarded.

Fuzzing operates through iterative cycles, commonly referred to as a fuzzing campaign. In practice, the PUT is first instrumented either at compile-time or at the binary level to enable runtime feedback collection. An initial seed corpus is then assembled, often with care to reduce redundancy while preserving diversity. New test cases are generated from this corpus using either generation-based (model-driven) or mutation-based approaches.

Generation-based fuzzers rely on explicit models of valid input formats, such as grammars or encoders. These models guide construction of structured test cases that conform to expected syntax while still exploring edge cases and invalid conditions. In contrast, mutation-based fuzzers start from one or more valid seeds and generate new inputs by applying random or guided transformations, including bit or byte flips, arithmetic changes, splicing of existing inputs, and dictionary-based substitutions.

Because mutation-based fuzzers depend heavily on the quality of the initial seeds, seed corpus management becomes a critical practical challenge. This includes selecting representative seeds, reducing redundancy, prioritizing promising inputs, and dynamically updating the corpus as new behaviors are discovered. These considerations play a central role in the effectiveness and efficiency of fuzzing-based multi-execution approaches.

The dominant paradigm in contemporary fuzzing is *coverage-guided fuzzing*, exemplified by popular tools such as AFL [71] and WinAFL [150]. Rather than relying on purely random testing, these systems use lightweight instrumentation to monitor code coverage and preferentially mutate inputs that lead to new execution paths. Because the full state space of a program is generally intractable to observe, coverage serves as a practical proxy for behavioral diversity. Higher coverage typically correlates with deeper exploration of previously unexplored program states.

Programs are commonly decomposed into basic blocks—straight-line code fragments with a single entry and exit point. Instructions within a basic block execute sequentially, and control flow changes only at block boundaries. Basic blocks provide convenient granularity for measuring coverage: function-level tracking is often too coarse, while instruction-level tracking can be unnecessarily detailed and expensive. Consequently, coverage is typically measured at the level of basic blocks or control-flow edges, which offer a practical balance between precision and efficiency.

Coverage-guided fuzzing can be viewed as an optimization process that seeks to maximize behavioral exploration while minimizing redundant executions [93]. A typical fuzzing loop repeatedly selects a seed, assigns it a budget of mutation attempts, generates mutated test cases, executes them, and retains those that crash or produce new coverage for further mutation. This process continues until predefined time or resource limits are reached.

These multi-path exploration challenges share structural similarities with malware analysis and establish a principled foundation for leveraging fuzzing to uncover latent malicious behaviors.

2.3.2 Fuzzing for Malware Analysis

Applying fuzzing to malware requires a conceptual shift from traditional software testing. Classical fuzzers in offensive security perturb explicit program inputs (*e.g.*, files, streams, or user data). In contrast, malware samples are often self-contained executables that do not meaningfully consume external inputs during analysis, or rely on inputs generated by loaders that are unavailable at fuzzing time. Consequently, the primary source of variability in malware execution is not its input data but rather its execution environment, which is made available to the malware samples via the system and library APIs [34].

Typical API fuzzers work by mutating function arguments across multiple executions. However, this approach is not suitable for malware analysis because arguments are semantically critical to the malware’s operation (*e.g.*, they might be file paths, registry keys, or IP addresses) and may themselves constitute important IoCs for characterization. Therefore, in malware-oriented fuzzing, the goal is not to mutate the sample’s arguments, but rather to alter what they do with the arguments. In practice, this entails fuzzing API return values rather than the arguments themselves. The rationale is that by changing the information the environment provides via the distinct return codes, the malware samples will present distinct behaviors [34].

This idea is closely related to earlier multi-VM approaches in which samples were repeatedly executed under different configurations [146]. Fuzzing generalizes and automates this strategy by performing variations in place, guided by runtime feedback rather than manual environment design.

Overall, fuzzing-based multi-execution represents a practical and scalable complement to traditional dynamic analysis. Rather than replacing existing sandboxes, it augments them with systematic exploration capabilities that reduce partial observability and produce more reliable behavioral profiles for detection and analysis.

2.4 Machine Learning for Malware Detection

Machine learning has become a central component of modern malware detection systems. Unlike traditional signature-based approaches, which rely on manually engineered rules and known patterns, ML-based systems automatically learn discriminative characteristics from the data. This capability is particularly important in the malware domain, where adversaries continuously evolve their techniques to evade static signatures [139].

In malware analysis, raw artifacts obtained from static or dynamic analysis are transformed into structured feature representations. Each sample is represented as a feature vector in a high-dimensional space, enabling statistical learning algorithms to model the boundary between benign and malicious software.

A central objective of ML-based detection systems is *generalization*: the ability to correctly classify previously unseen samples. By leveraging a wide range of features extracted from malware samples, these models can generalize beyond known threats

and identify previously unseen variants that share semantic similarities with existing malware families. This requirement is especially challenging in cybersecurity due to high-dimensional feature spaces, class imbalance [44], adversarial manipulation [103], and concept drift, where malware characteristics evolve over time [45]. Consequently, robustness, resistance to overfitting, and adaptability are critical considerations.

2.4.1 Learning Paradigms and Tasks

Machine learning approaches to malware analysis can be categorized according to the learning paradigm adopted, which is largely determined by data availability and the operational objectives of the system.

- *Supervised learning* assumes the availability of labeled datasets and formulates malware analysis as a statistical inference problem. In its most common formulation, malware detection corresponds to a binary classification task that distinguishes malicious from benign software. More granular settings extend this formulation to multi-class classification, where samples are assigned to specific malware families or campaigns. The effectiveness of supervised approaches critically depends on the representativeness and reliability of the labeled training data.
- *Unsupervised learning* operates without labeled examples and instead seeks to uncover latent structures within the data. Clustering methods group samples exhibiting similar structural or behavioral characteristics, thereby supporting the discovery of malware families and variants. Anomaly detection techniques model normal software behavior and identify statistically significant deviations, potentially revealing previously unseen or zero-day threats.
- *Semi-supervised learning* combines a limited set of labeled samples together with a larger pool of unlabeled data. This paradigm is particularly relevant in cybersecurity contexts, where high-quality labeling is costly, time-consuming, and sometimes ambiguous. By exploiting both labeled and unlabeled information, semi-supervised methods aim to improve generalization while reducing annotation effort.

From a task-oriented perspective, machine learning applications in malware analysis primarily aim to assign samples to predefined categories or to organize them into structurally meaningful groups. Two main objectives dominate the field:

- *Classification*, which assigns previously unseen samples to known categories (e.g., benign versus malicious, or to specific malware families) [124]. This task directly supports automated detection pipelines and real-time defense mechanisms, where the system must decide whether to allow, quarantine, or block a program.
- *Clustering*, which groups samples according to similarity without relying on predefined labels. Although less immediately applicable to real-time decision-making, clustering plays a central role in threat intelligence and malware forensics, enabling analysts to identify emerging families, track evolutionary trends, and detect previously unknown strains [94].

2.4.2 Feature Extraction and Representation

The effectiveness of any ML-based detection system depends critically on the quality of its feature representation. Feature extraction translates raw artifacts—such as binary content, execution traces, or communication logs—into structured numerical representations that learning algorithms can process. This transformation relies on data mining and program analysis techniques to identify meaningful patterns within large and often noisy datasets [6]. Features may be derived from static analysis, dynamic analysis, or hybrid approaches that combine both perspectives.

From a static perspective, malware samples are characterized using structural properties extracted from the PE format, including sections, imports, compiler metadata, and embedded strings. Byte-level representations are also widely adopted, capturing local syntactic patterns through contiguous byte or opcode sequences.

In dynamic analysis, behavioral features are extracted through runtime observation within sandboxes or emulated environments. Program analysis techniques such as disassembly, control-flow reconstruction, and data-flow analysis further support feature extraction by revealing internal structural relationships in the binary. API and system call sequences are particularly informative, as they reflect how a program interacts with the operating system [69]. Network- and filesystem-level activities provide additional behavioral indicators, including command-and-control communication, persistence mechanisms, and configuration changes.

N-gram-based models. The n -gram model is among the earliest and most widely adopted feature extraction techniques in malware detection. It represents a sequence (e.g., bytes, instructions, or API/system calls) as a multiset of contiguous subsequences of length n , thereby capturing local, order-sensitive co-occurrence patterns. This formulation implicitly assumes that dependencies are predominantly confined within a fixed-size window, making the approach applicable to both static artifacts and dynamic execution traces.

Despite their empirical effectiveness, n -gram representations exhibit inherent limitations. By construction, they model short-range dependencies and cannot explicitly capture long-range structural or semantic relationships within a program. Sequential adjacency does not necessarily imply semantic relevance, and minor transformations—such as instruction reordering, insertion of no-op operations, or behavioral padding—may significantly alter the extracted feature distribution. Moreover, the dimensionality of the feature space grows exponentially with n , leading to sparsity, increased computational cost, and reduced scalability [6].

For these reasons, while n -grams provide a simple and computationally convenient baseline, modern malware detection increasingly favors representations that encode structural organization and relational dependencies more directly.

Graph-based models. Graph-based models have gained prominence due to their ability to encode structural and relational dependencies within programs. In this representation, a program or its execution trace is modeled as a graph $G = (V, E)$, where V denotes nodes (e.g., functions, basic blocks, or system calls) and E represents relationships between them (e.g., control-flow, data-flow, or invocation edges) [6].

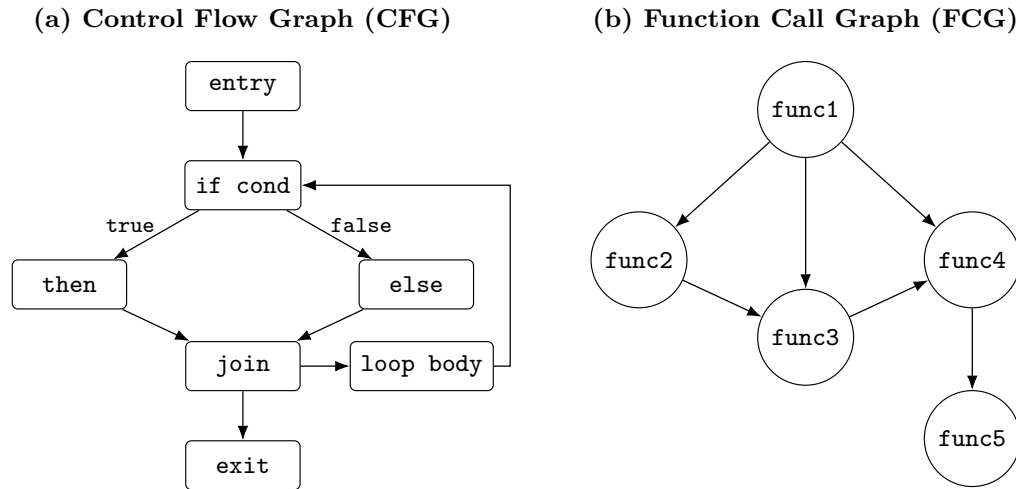


Figure 2.2. Conceptual comparison between two common program graph representations. (a) A Control Flow Graph (CFG) models intra-procedural control transfers between basic blocks within a function, explicitly capturing branching, merging, and looping structures. (b) A Function Call Graph (FCG) represents inter-procedural relationships, where nodes denote functions and directed edges encode invocation dependencies among them. The figure is simplified and intended to emphasize the difference in granularity, scope, and semantic abstraction between the two graph models.

Unlike flat vector representations, graphs capture relational structure, enabling learning algorithms to exploit local neighborhoods and higher-order dependencies. This relational expressiveness is particularly relevant in malware analysis, where behavioral semantics emerge from non-local interactions among program components and may remain observable even under code obfuscation or reuse techniques.

Graph abstractions can be derived from static program structure or dynamic runtime behavior, depending on the analysis setting. In static analysis, graphs are typically constructed from disassembled code. A *Control Flow Graph* (CFG) represents intra-procedural execution flow, where nodes correspond to basic blocks and directed edges denote possible control transfers within a function. CFGs encode explicitly branching, merging, and looping constructs, modeling execution semantics at a fine-grained level.

At a higher level of abstraction, a *Function Call Graph* (FCG) captures inter-procedural relationships across the program. In an FCG, nodes represent functions and directed edges encode invocation dependencies among them. While CFGs provide detailed structural information within individual functions, FCGs emphasize program organization and interaction patterns across functional components.

Figure 2.2 illustrates the conceptual distinction between these two representations.

Learning over graph structures is performed using neural architectures designed for relational data. Graph Neural Networks (GNNs) learn node- or graph-level representations through iterative message passing, where each node updates its embedding by aggregating information from neighbors. As layers stack, the receptive field expands, enabling the model to capture multi-hop dependencies and higher-order structural patterns while remaining invariant to node ordering.

By integrating structural context with node-level attributes, GNN-based approaches can model complex program semantics and behavioral patterns arising from interactions among distributed components. This capability makes them particularly well suited for malware detection settings in which structural organization and behavioral variability play a central role.

2.4.3 Classification Algorithms

Supervised malware detection is commonly formulated as a statistical classification problem, for which a variety of machine learning algorithms have been employed. Given a feature representation $x \in \mathbb{R}^d$ extracted from a software sample, the objective is to learn a function $f: \mathbb{R}^d \rightarrow \{1, \dots, C\}$ that maps the sample to one of C predefined classes (e.g., benign or malicious).

Classification algorithms differ in their modeling assumptions, decision boundaries, computational complexity, and robustness to high-dimensional data. In the following, we briefly review several widely adopted methods in malware research.

- *Naive Bayes (NB)*. Naive Bayes is a probabilistic classifier based on Bayes' theorem. It models the posterior probability by estimating class-conditional likelihoods under the assumption that features are conditionally independent given the class label. Despite this strong independence assumption, NB is computationally efficient, scales well to high-dimensional feature spaces, and requires relatively small training datasets. However, its simplifying assumption may limit its ability to capture complex feature interactions commonly present in malware behavior.
- *Logistic Regression (LR)*. Logistic Regression is a discriminative classifier that directly models the posterior probability using a logistic function applied to a linear combination of features. It is particularly effective when the decision boundary between classes is approximately linear in the feature space. LR is computationally efficient and interpretable, but its performance may degrade when relationships between features and labels are highly non-linear unless appropriate feature transformations are applied.
- *Support Vector Machines (SVM)*. Support Vector Machines are margin-based classifiers that seek a decision boundary maximizing the geometric margin between classes. For linearly separable data, SVM constructs a hyperplane in the feature space; through kernel functions, it can implicitly project data into higher-dimensional spaces to model non-linear relationships. SVMs are well-suited to high-dimensional problems and often demonstrate strong generalization performance. However, their computational cost can become significant for large-scale datasets, and careful selection of kernel functions and hyperparameters is required.
- *Random Forest (RF)*. Random Forest is an ensemble learning method that aggregates multiple decision trees constructed via bootstrap sampling and random feature selection. By combining the predictions of independently trained trees, RF reduces variance and mitigates overfitting compared to

a single decision tree. Random Forest models are robust to noise, handle high-dimensional data effectively, naturally support multi-class classification, and provide measures of feature importance that enhance interpretability in security applications.

- *k-Nearest Neighbors (k-NN)*. The k-Nearest Neighbors algorithm is an instance-based learning method. Classification is performed by assigning a sample the majority label among its k closest training instances according to a predefined distance metric. While conceptually simple and flexible, k-NN incurs high computational cost at prediction time and is sensitive to feature scaling and the choice of distance function. In high-dimensional settings—common in malware analysis—the curse of dimensionality may reduce the discriminative power of distance-based methods.

Beyond classical machine learning approaches, deep learning (DL) methods have increasingly been applied to malware detection. In contrast to traditional models that rely on manually engineered features, deep neural networks aim to automatically learn hierarchical representations directly from raw or minimally processed data. Convolutional Neural Networks (CNNs) have been applied to byte-level representations of binaries, treating executables as one-dimensional signals or image-like structures. Recurrent Neural Networks (RNNs) and transformer-based architectures have been used to model sequential behavioral data such as API call traces or system call sequences. Deep learning models are particularly effective when applied to large-scale or highly expressive representations, where complex non-linear dependencies must be captured.

Overall, the suitability of these algorithms for malware detection depends on factors such as dataset size, feature dimensionality and quality, interpretability requirements, computational constraints, and the presence of noise. In this thesis, the Random Forest model is adopted as the primary supervised classification method. The methodological motivations for this choice are detailed in Chapter 5.

Chapter 3

State of the Art

A rigorous assessment of the state of the art is essential for positioning the contributions of this thesis within the broader research landscape. This chapter examines the principal research directions in the detection and analysis of malware behaviors, identifying their methodological assumptions and the gaps that remain unresolved.

The discussion unfolds along three complementary dimensions. We begin by reviewing existing approaches to detecting process injection, outlining current techniques and their structural limitations. We then examine prior work that applies fuzzing methodologies to malware analysis, with particular attention to how these techniques enhance data collection and behavioral exploration. Finally, we survey graph-based behavioral representations for malware detection, evaluating current modeling strategies and highlighting the limited integration between trace acquisition, structural abstraction, and learning-based classification.

These perspectives reveal both detection-specific weaknesses and broader representational and statistical gaps. This synthesis motivates principled behavioral abstraction and multi-trace reasoning, thereby framing the contributions presented in the subsequent chapters.

3.1 Detecting Process Injection

Despite its widespread adoption by modern malware, process injection has received comparatively limited dedicated attention within the academic literature. In this section, we review prior efforts on detecting this technique.

Existing approaches can be broadly grouped into three main research directions: (i) memory forensics-based techniques that analyze memory artifacts to identify injected code, (ii) behavior-based approaches that monitor runtime activities indicative of injection, and (iii) alternative and hybrid solutions, including methods that track environmental side effects associated with injection attacks, as well as data-driven approaches leveraging machine learning (ML).

3.1.1 Memory Forensics Approaches

Memory forensics-based approaches aim to detect process injection by analyzing artifacts extracted from system memory, typically in the form of memory dumps

collected during execution. Memory represents a rich source of valuable evidence [130], particularly in the context of fileless malware, where malicious logic resides primarily in volatile memory rather than on disk.

The continuous improvement of industry frameworks such as Volatility [65] has led analysts to increasingly integrate memory analysis into their investigative workflows. In this setting, the ability to accurately and efficiently identify injected code within memory dumps is crucial. Several tools support this task, including Volatility’s *malfind* [66] and *hollowfind* [2]. However, these tools exhibit notable limitations. *Malfind* is known to suffer from high false positive rates, whereas *hollowfind* is limited to detecting a subset of injection techniques, primarily those resembling process hollowing.

In academia, Barabosch et al. [26] proposed Quincy, a memory-based detection system designed to address several of these shortcomings. Quincy employs a rich heuristic composed of 38 features drawn from seven categories commonly associated with injected code, including memory region permissions, region sparseness, and shellcode-related indicators. These features are embedded into a vector representation, and consecutive memory pages are classified as malicious or benign. While memory-based heuristics offer a wealth of discriminative indicators and considerable design flexibility, they also introduce important limitations. Snapshot-based analysis captures only a single system state, potentially missing transient behaviors or short-lived injection artifacts. Moreover, acquiring memory dumps at a frequency suitable for near real-time monitoring is operationally expensive and may not scale to production environments. Certain injection techniques may further evade post-mortem detection, particularly when adversaries deliberately wipe or overwrite injected regions prior to memory acquisition.

To partially mitigate the inherent limitations of post-mortem analysis, Nasereddin et al. [109] explored a live memory analysis approach that eliminates the need for periodic memory dumping. Their framework adopts a multi-agent architecture in which distributed monitoring components observe execution artifacts, memory protection features, PE header integrity, registry contents, and suspended processes, while also processing execution traces collected via Event Tracing for Windows (ETW) [56]. The system is evaluated against six representative injection techniques, namely classic DLL Injection, Reflective DLL Injection, Process Hollowing, Hook Injection, Registry-based Injection, and .NET DLL Injection. By coordinating parallel monitoring agents through a centralized controller, the framework reduces detection latency and resource overhead compared to traditional centralized solutions, while maintaining continuous visibility into injection-related activity.

3.1.2 Behavioral Tracking Approaches

A complementary line of research focuses on behavioral tracking, that is, the collection and analysis of execution traces at the operating system (OS) level to identify process injection and related in-memory attacks.

Korczynski et al. [87] introduced Tartarus, an automated system designed to capture and analyze malware propagations involving dynamically generated code, code injection, and code-reuse attacks. Tartarus performs system-wide tracing through

dynamic taint analysis: it taints the malware’s in-memory code and propagates taint across the OS to follow execution, even when control flow is transferred through code-reuse mechanisms. By avoiding assumptions based on API hooking, Tartarus can identify injection events and pinpoint code-reuse activity while providing detailed contextual information about them.

Similarly, Arefi et al. [111] proposed FAROS, a reverse-engineering framework built on whole-system dynamic information flow tracking to detect stealthy in-memory injection attacks. FAROS is designed without assumptions regarding the structure of injected payloads or their persistence in memory, allowing it to capture a wide range of injection variants. Beyond flagging injection activity, FAROS provides fine-grained provenance information for the injected content, thereby reducing the manual effort required during forensic analysis.

Whole-system dynamic information flow tracking significantly slows down execution and increases resource consumption, which complicates deployment in production environments or integration into performance-sensitive systems such as antivirus engines and sandboxes. This highlights a fundamental trade-off between semantic precision and practical scalability in behavioral tracking approaches.

More recently, Starink et al. [135] conducted a systematic study of code injection techniques in Windows malware and proposed a taxonomy grouping them into classes based on shared operational traits. Building on this taxonomy, the authors implemented behavioral models for 17 injection techniques and used them to perform a large-scale longitudinal measurement study. Injection activities are modeled using behavior graphs that represent API calls and their dependencies within Petri nets [108]. While this modeling framework enables structured reasoning about injection patterns, it relies on exact behavioral schemas. Such rigidity can reduce robustness when injection variants deviate from predefined patterns or when higher-level event abstraction is required. The study further distinguishes injection techniques into active and passive methods based on their level of interaction with the victim process. Passive techniques do not directly interact with a target process but instead induce execution indirectly, for instance by tampering with registry-based loading mechanisms. Notably, the MITRE ATT&CK framework does not classify these methods as injection but rather as Event Triggered Execution techniques [8].

3.1.3 Environment-Based and Data-Driven Approaches

This category encompasses alternative solutions that monitor environmental changes associated with injection attacks and data-driven methods leveraging machine learning. Unlike memory forensics or low-level behavioral tracing, these approaches infer injection activity indirectly, either by observing deviations in controlled environments or by learning discriminative patterns from system-level telemetry.

Barabosch et al. [27] proposed Bee Master, a host-based code injection detection system that adapts the honeypot paradigm and monitors the environment for changes associated with injection. Rather than relying on OS-specific indicators such as API-hooking signatures, Bee Master monitors agnostic primitives common to modern systems (e.g., processes, threads, loaded modules, and memory pages) to detect deviations from a known baseline. A supervising component (“Queen Bee”)

spawns and controls multiple decoy processes (“Worker Bees”) and periodically inspects them for anomalies, including unexpected threads, newly loaded modules, or suspicious memory regions. When deviations are observed, the compromised process memory can be dumped to support subsequent analysis. The authors implement prototypes for both Windows and Linux and report qualitative and quantitative evaluations across several malware families. However, the decoy-based strategy introduces intrinsic limitations: detection depends on whether malware chooses (or can be lured into choosing) the deployed decoy processes, which may not occur if strong victim-selection criteria are employed. Furthermore, the approach does not generalize to all injection-like behaviors and may fail to capture recursive or non-standard injection patterns.

Snow et al. [132] presented SHELLOS, a hardware virtualization-assisted framework for fast detection and analysis of shellcode. By leveraging hardware virtualization extensions, SHELLOS executes suspicious instruction sequences directly on the CPU inside a lightweight guest kernel, enabling accurate inspection of buffers without relying on software emulation. This design reduces exposure to evasion strategies that exploit discrepancies between emulated and native execution environments. Experimental evaluation on real-world attack corpora, including network exploits and malicious PDF samples, demonstrates substantial scalability improvements over traditional emulation-based approaches, alongside enhanced robustness to emulator-targeted evasion.

More recently, data-driven techniques have been explored to detect stealthy infection vectors, including process injection. Wang et al. [145] introduced PROVIDE-TECTOR, a provenance-based system for detecting stealthy impersonation malware. The core intuition is that even stealthy malware must interact with the underlying OS, leaving traces within system-level provenance graphs. PROVIDE-TECTOR first applies a selection algorithm to identify potentially malicious subgraphs within provenance data, and subsequently employs neural embedding and machine learning models to detect deviations from benign behavioral baselines. Evaluation on a large enterprise provenance dataset reports high detection performance (average F1 score of 0.974), and the authors further conduct interpretability analyses to better understand the learned decision boundaries. However, as with many ML-based approaches, performance depends on the representativeness of training data and may degrade under distribution shifts or adversarial adaptation.

Wang et al. [144] proposed ProcGuard, a framework specifically targeting process injection behaviors through fine-grained monitoring of sensitive function calls and call-chain characteristics. The system collects API invocation sequences associated with injection activity and analyzes their structural relationships to identify malicious execution patterns. An enhanced Recurrent Convolutional Neural Network (RCNN) model is employed to improve detection over tampered or obfuscated memory segments. Experimental results report an accuracy of 81.58% with reduced false-positive rates compared to baseline systems, along with improvements in detection latency and runtime overhead. Nevertheless, similar to other supervised learning approaches, robustness against unseen injection variants and evolving adversarial strategies remains an open challenge.

Table 3.1 summarizes the key detection signals and recurring structural limitations across existing approaches.

Table 3.1. Qualitative comparison of prior approaches for detecting process injection and related in-memory attacks, highlighting their primary detection signals and recurring structural limitations.

Work	Category	Detection mechanism	Main limitations
Malfind [66]	Memory forensics	Heuristic inspection of memory dumps (VADs and page permissions)	High false-positive rate; post-mortem analysis; costly memory acquisition
Hollowfind [2]	Memory forensics	Detection of PE/VAD inconsistencies indicative of hollowing	Limited coverage (primarily hollowing variants); post-mortem analysis
Barabosch et al. [26]	Memory forensics	Page-level feature extraction and classification over memory dumps	Snapshot-based analysis may miss transient artifacts; evasion via memory wiping or overwrite
Nasereddin et al. [109]	Live memory analysis	Multi-agent monitoring of execution artifacts, memory protection, PE integrity, registry state, and ETW traces	Evaluated on a limited set of techniques; potential blind spots in monitored sensors
Korczynski et al. [87]	Behavioral tracking	Whole-system dynamic taint analysis to follow in-memory code across OS boundaries	Significant runtime and memory overhead; difficult to deploy at scale; vulnerable to anti-analysis logic
Arefi et al. [111]	Behavioral tracking	Whole-system dynamic information flow tracking with provenance reconstruction	Similar scalability and overhead constraints inherent to full-system taint analysis
Starink et al. [135]	Behavioral modeling	API dependency graphs modeled via Petri nets to represent injection schemas	Relies on predefined behavioral templates; reduced robustness to variants and abstraction shifts; vulnerable to evasion
Barabosch et al. [27]	Environment-based (honeypot)	Decoy processes periodically inspected for anomalous threads, modules, and memory regions	Detection depends on malware interacting with decoys; limited coverage of non-standard or indirect injection patterns
Snow et al. [132]	Virtualization-assisted analysis	Native execution of suspicious buffers inside a lightweight virtualized kernel	Not injection-specific (focuses on shellcode inspection); requires hardware virtualization support
Wang et al. [145]	Data-driven	Selection and embedding of provenance subgraphs followed by ML-based classification	Strong dependence on training data; performance degradation under distribution shift or adversarial adaptation
Wang et al. [144]	Data-driven	Modeling of sensitive API sequences and call-chain structure using RCNN	Generalization to unseen injection variants uncertain; depends on completeness of monitored APIs

3.1.4 Limitations of Existing Solutions

Despite their contributions, existing approaches exhibit structural limitations that hinder a comprehensive understanding of process injection and constrain their effectiveness in realistic settings.

Broadly, prior work tends to suffer from two primary limitations. First, several approaches rely on computationally intensive analyses (full memory dumping or dynamic taint tracking) that incur significant runtime and resource overhead. Beyond scalability concerns, these techniques may still fail to observe injection behaviors if execution conditions are not met during analysis, or if evasive mechanisms suppress the relevant activity. Consequently, high analysis cost does not necessarily translate into behavioral completeness.

Second, many detection strategies adopt a classification-driven paradigm based on predefined rules or behavioral schemas derived from documented injection implementations. While effective against known patterns, minor implementation variations or previously undocumented triggering mechanisms can evade detection. As a result, coverage may degrade as malware evolves or diversifies its injection logic.

A further practical limitation concerns the interaction between injection and evasion. Injection activity is frequently guarded by environment-sensitive checks that inhibit execution under analysis conditions. Although evasion is often treated as an orthogonal problem, the absence of explicit anti-evasion provisions in prior work can lead to a systematic underestimation of injection prevalence. Moreover, existing studies rarely account for variations in how malware implements process injection techniques. To quantify this effect, in Chapter 4 we conducted a restricted experiment within our dataset. Under an analysis configuration without dedicated anti-evasion mechanisms, 47.3% of samples performing injection behaviors were not observed to execute the relevant injection logic. This highlights the extent to which analysis environment assumptions influence empirical conclusions.

Taken together, these observations motivate the need for a detection methodology that (i) captures the operational semantics underlying injection rather than surface-level implementation patterns, (ii) remains resilient to behavioral variation and evasive gating, and (iii) balances analytical depth with practical scalability.

3.2 Fuzzing Applications in Malware Analysis

Although the research community has long addressed environment-sensitive and evasive malware by making analysis platforms more transparent and disguising artifacts that reveal the analysis environment [60, 67, 97, 100], such malware continues to challenge existing defenses. In this section, we review recent works on environment-focused multi-path exploration that introduce or formalize fuzzing as a systematic technique for malware analysis, demonstrating how it can be integrated into sandbox workflows to bypass anti-analysis checks and uncover latent payload logic.

Gorter et al. [72] proposed *Enviral*, a hybrid evasive-malware analysis framework that combines sandbox execution with fuzzing-driven environmental perturbations to systematically defeat environment-based evasion. *Enviral* assumes abort-if-present evasive logic and repeatedly re-executes a sample while mutating the outcomes of environment-query system calls, retaining only those mutations that lead to increased interesting unique system-call activity. Architecturally, *Enviral* consists of a user-level hooking layer that intercepts and modifies Windows system calls, and a controller that iteratively explores mutation configurations guided by a lightweight coverage signal based on distinct system calls deduplicated by call stack. The system distinguishes between definite mutations—handcrafted substitutions for known evasive queries tied to known artifacts—and volatile mutations, which are speculative and preserved only when they increase coverage. In an evaluation on 8,440 deduplicated Windows malware samples, *Enviral* increases unique system-call activity in 26.1% of cases. In a focused comparison, it exposes hidden behavior in more samples and with larger gains in unique system-call activity against the transparency-oriented baseline *BluePill* [60]. Conceptually, *Enviral* establishes environment-query mutation with

a lightweight feedback signal as a scalable way to surface additional behavior in evasive malware, positioning itself between transparency-only approaches and heavier multi-path analyses. However, it still relies on manually curated definite mutations and lacks systematic backtracking over earlier exploration decisions.

Botacin et al. [34] conducted a Systematization-of-Knowledge (SoK) and a large-scale experimental comparison of multi-path malware analysis techniques, including fuzzing, coverage-guided fuzzing, forced execution, pure symbolic execution, and concolic execution. They reimplemented representative tools for each category and executed them in parallel on a dataset of approximately 21,000 real-world Windows malware samples, generating more than 21 million execution traces. Their evaluation compares techniques along three dimensions: path discovery, code coverage, and computational cost. The study revealed several important insights about multi-path behavior in practice. First, natural run-to-run variation is common: even repeated executions without external stimulation frequently expose alternate paths, implying that standard single-run sandbox analyses observe only a small fraction of a program’s potential behavior. Second, fuzzing function returns (rather than inputs) increases both path discovery and code coverage, though it exhibits diminishing returns beyond an optimal mutation intensity. Third, forced execution discovers the largest number of distinct paths overall, but at substantial computational cost due to the exponential number of required runs. Fourth, pure symbolic execution was found largely impractical at scale: most samples—especially packed binaries—produced little meaningful output within reasonable time limits. Fifth, concolic execution emerges as a promising middle ground, discovering many distinct paths per run and achieving path diversity comparable to forced execution, but with lower overhead; its main limitations stem from incomplete Windows API summaries, limited support for packed binaries, and taint-tracking challenges. Based on these findings, the authors articulate practical recommendations for analysts, sandbox developers, and researchers: (i) execute samples multiple times to capture natural behavioral variability, (ii) integrate multi-path exploration capabilities into sandboxes rather than relying on single traces, and (iii) develop malware-specific path prioritization and evaluation metrics. Overall, this work positions (return-value) fuzzing as a practical way to increase behavioral and path diversity in malware analysis.

Bottura et al. [38] introduced PFuzzer, a principled environment-fuzzing approach that automatically discovers configurations that maximize behavioral diversity while minimizing redundant executions via coverage-guided fuzzing. Rather than assuming a single “correct” environment, the system systematically explores alternative environments to elicit multi-path behavior in real-world Windows malware. Concretely, the authors model alternative execution environments as an input-generation problem. They introduce execution policies as a unified mechanism to intercept and modify a sample’s interactions with environmental information sources; each policy consists of a set of mutation rules that alter those interactions in a sound and reproducible way. To guide exploration, PFuzzer employs a novel coverage feedback mechanism that combines (i) external coverage—sensitive to observable effects on the host such as environmental queries and conspicuous system-state changes—and (ii) internal coverage based on basic block execution, so that partial progress is not overlooked. External coverage separately records accesses to environmental

sources (natural mutation targets) and conspicuous actions that change system state (as indicators that some expectations were satisfied). These coverage facts jointly steer both mutation selection and the fuzzing scheduler. The system explores policies in epochs, maintaining a queue of promising environments. Each policy is ranked using a significance score that prioritizes novel, externally visible effects, and an energy score that controls how much further mutation effort it receives. PFuzzer mutates 68 environment-related Windows APIs and relevant instructions (e.g., `cpuid`, `rdtsc`) with operators such as force success, force failure, retargeting, and time manipulation, while preserving execution soundness. To evaluate the approach, the authors assemble a manually annotated corpus of 1,078 environment-sensitive Windows malware samples, carefully curated from over 315,000 specimens via deduplication, static filtering, family balancing, and dynamic triage. In this dataset, PFuzzer reveals conspicuous activity missed by the best prior system for 36.09% of samples, and activates dormant evasive tactics for 70.64% of samples that would appear non-evasive in a baseline run. Additional activity often stems not merely from bypassed evasions, but from previously hidden malicious behaviors that depend on “right” environmental conditions. Conceptually, the work demonstrates that coverage-guided fuzzing over environments can be a profitable approach to multi-path malware analysis, bridging the gap between transparency-only defenses and heavyweight symbolic or forced-execution approaches. However, its impact on large-scale, real-world analysis pipelines—particularly for downstream tasks such as classification—remains largely unexplored.

3.3 Graph-based Behavioral Representations in Malware Analysis

Graph-based representations have emerged as a powerful paradigm for modeling malware behavior, as they encode structural relationships among program entities that are difficult to capture using flat feature vectors or raw API sequences. In API call graphs, nodes represent invoked functions and edges encode relationships such as call transitions, dependencies, or interaction frequencies [30]. Compared to sequential encodings that primarily capture local co-occurrence patterns, graph structures explicitly model connectivity, branching behavior, and long-range dependencies, thereby providing a richer abstraction of behavioral structure.

Most existing work employs graph representations as inputs to graph learning models, particularly Graph Neural Networks (GNNs) [84], for end-to-end malware classification. In these approaches, graphs are derived from static analysis (e.g., function call graphs extracted from binaries) or dynamic execution traces (e.g., API transition graphs obtained from sandbox runs), and embeddings are learned for downstream classification. While these methods demonstrate strong empirical performance, they typically operate on a single graph per sample, implicitly assuming that one static analysis or one execution trace sufficiently represents program behavior.

Early work by Jiang et al. [79] introduced DLGraph, combining static function call graphs with API presence vectors to derive structural embeddings for malware detection. The approach constructs a caller-callee graph and applies representation learning techniques to obtain fixed-size embeddings prior to classification. Although

pioneering in demonstrating the potential of structural abstractions, DLGraph relies on accurate static analysis and does not explicitly model behavioral variability across executions.

Subsequent research shifted toward dynamic behavioral graphs derived from sandbox execution. Zhang et al. [151] proposed the Spectral-based Directed Graph Network (SDGNet), modeling directed API interaction graphs and applying spectral graph convolution techniques to learn graph-level embeddings. Similarly, Li et al. [95] construct weighted API transition graphs and apply Graph Convolutional Network (GCN)-based architectures for classification. These approaches emphasize learning discriminative structural embeddings from execution-derived graphs, showing that modeling directionality and transition strength can improve detection performance.

More advanced frameworks incorporate richer semantic information into graph representations. Li et al. [92] integrated API argument features and heterogeneous node attributes, combining structural and attribute information within a unified graph model. More recently, Oliveira et al. [50] further explored end-to-end graph convolutional models for behavioral malware detection using sandbox-generated execution graphs. Collectively, these works confirm that structural modeling enhances representational expressiveness compared to flat sequential encodings.

3.3.1 Methodological Limitations and Structural Gaps

Despite their representational advantages, existing graph-based approaches exhibit several recurring limitations that constrain robustness and generalization.

A central limitation is the reliance on single-execution analysis. Behavioral graphs are typically constructed from one static extraction pass or one sandbox execution, implicitly treating that observation as representative of the program’s behavioral space. Variability across executions—caused by environmental adaptation, conditional logic, anti-analysis mechanisms, or nondeterminism [34]—is therefore not consolidated into a unified structural abstraction.

A related constraint stems from dependence on sandbox-generated traces. Runtime-derived graphs provide contextual behavioral information but remain vulnerable to environment-aware malware and staged execution strategies. Because graph construction is tightly coupled to the execution environment, incomplete or adversarially manipulated traces may directly affect the learned representation.

Evaluation practices further limit generalizability. Many studies rely on private or domain-specific datasets, complicating reproducibility and cross-dataset validation. Moreover, reported results predominantly emphasize classification accuracy, with limited analysis of how performance evolves under repeated execution, increasing behavioral diversity, or controlled distributional shifts.

Temporal modeling also remains comparatively underexplored. Although API call graphs capture structural relationships, they are frequently treated as static objects, abstracting away fine-grained sequencing, causality, and behavioral evolution across runs. Such abstraction may obscure execution-dependent divergence and time-dependent strategies characteristic of modern malware.

Beyond these methodological concerns, a more fundamental structural gap persists: limited integration between trace acquisition, representation design, and decision-making. Existing approaches typically treat trace generation as a fixed

preprocessing step, while graph construction and classification are developed independently of how behavioral evidence is collected. Exploration strategy, aggregation policy, and structural modeling are rarely analyzed together.

Collectively, these observations indicate that current graph-based malware detection frameworks emphasize structural embedding and classification performance, yet devote comparatively less attention to execution variability, multi-trace consolidation, and the statistical consequences of repeated behavioral observation. Addressing these gaps requires treating structural representation, evidence aggregation, and exploration control as coupled components of a unified multi-trace analytical framework in realistic settings.

Chapter 4

Unmasking Process Injection: Systematic Detection in Windows Malware

This chapter presents the first main contribution of this thesis: a novel and systematic methodology for the detection and analysis of process injection attacks.

Existing approaches typically focus on specific injection techniques, which limits the ability to reason about the phenomenon in a generalized and unified manner. To address this limitation, we demonstrate how a minimal subset of operations can be leveraged by analysts and security systems to improve behavioral visibility and enhance detection capabilities.

We begin by defining the challenges of analyzing process injection attacks in malware and by motivating our approach. We then provide an overview of the system's design and describe the key implementation details, walking through each component that enables automated injection detection. Finally, we evaluate the system with experiments designed to answer three research questions: **(RQ1)** What are the most common process injection techniques adopted by modern malware? Does the release of information about these techniques influence their adoption? **(RQ2)** Is there a correlation between malware families and the injection techniques they exploit? Have the injection techniques adopted by each family evolved over time? **(RQ3)** Can the proposed methodology be effectively integrated into practical security solutions, and if so, with what level of performance (e.g., false positives)?

The research contents presented in this chapter are also described in the paper *Can You Run My Code? A Close Look at Process Injection in Windows Malware* [53], published in the proceedings of the 20th ACM Asia Conference on Computer and Communications Security (ASIACCS 2025).

4.1 Introduction

Malware, the primary tool of cybercriminals, enables diverse attacks and affects numerous victims. Its rapid evolution poses significant challenges to the security community, which must balance detection accuracy against performance and deployment constraints [5]. While defenders strive to improve monitoring and analysis techniques, malware authors actively adapt their tools to evade them.

A particularly effective strategy for evading process monitoring-based defenses consists of running malicious code within the address space of another process. This is the core concept of *process injection*, a general attack methodology that enables the execution of arbitrary code inside a victim process, thereby masking malicious activities behind legitimate execution contexts. Process injection is also a key enabler of fileless attacks [81], which today represent a dominant threat class [98].

By design, process injection increases the stealth of malware operations. It allows adversaries to bypass allowlist-based defenses and, in some cases, escalate privileges or abuse trusted processes. From a detection perspective, injected payloads are substantially more challenging to identify than standalone malware for several reasons. Most injection techniques leave no trace on disk, inherently evading static analysis and signature-based detection. During dynamic analysis, if monitoring systems fail to capture the precise moment at which the injected payload begins execution, the malicious activity may blend indistinguishably with the victim process's legitimate behavior [52]. Even post-mortem memory forensics [2, 65, 66] may fail to detect injected payloads, for example when malware employs memory-wiping provisions.

A viable defense strategy would therefore be to detect and prevent injection attempts directly. However, this task is far from trivial. Most injection techniques target Windows systems and leverage an extensive range of legitimate built-in features designed for application interoperability and to enhance functionality. Unfortunately, skilled malware authors have repeatedly repurposed these capabilities to implement injection in diverse and creative ways. This ingenuity is well documented in practitioner-oriented analyses [74, 112], technical reports [15, 86], and, to a lesser extent, academic research [135].

In operational environments, defenders face additional constraints. Antivirus (AV) products cannot sustain fine-grained execution inspection for extended periods without incurring prohibitive overhead. Endpoint Detection and Response (EDR) solutions have lately proposed high-level heuristics and detection rules for characterizing injection behavior [63, 129, 134], but these guidelines are not always directly applicable, tend to cover only a subset of known techniques, and remain fragile against implementation variability.

Despite its prevalence in real-world malware, process injection has received comparatively limited attention from the academic community. Prior work has mostly focused on individual techniques or narrow classes of attacks [27, 86, 87, 144]. A comprehensive analysis of injection techniques and their behavioral patterns only appeared recently [135]. As a result, process injection remains an under-explored yet pivotal malware behavior, warranting deeper and more systematic study.

Our approach. This chapter introduces a detection and analysis methodology that targets fundamental operational traits inherent in all known variants of process

injection. Our approach synergically capitalizes on two intuitions leveraged separately by prior work, and further relaxes the assumptions behind their deployment that limit their applicability in practice.

Starink et al. [135] surveyed common injection techniques and derived behavior graphs that capture the co-occurrence and dependencies of API calls underlying each technique. They applied these graphs to real-world malware by identifying operations that manipulate *congruent memory addresses*, a criterion that effectively groups related activity while removing the need for expensive taint tracking used in other works [87, 111]. We adopt memory range congruence as a lightweight yet powerful proxy for associating injection-related operations.

To identify candidate injection behavior, we recall instead how Klein et al. [86] established that process injection tends to take place in *three core operations*: allocating memory in a target process, writing a payload into that memory, and transferring execution to it. Their work surveyed techniques and identified the Windows APIs commonly used to implement each step.

The key idea behind our proposal is to combine the co-occurrence of these operational primitives with memory range congruence, while explicitly accounting for implementation diversity. We introduce *equivalence classes* that model alternative ways of realizing each core operation. This addresses a main limitation of prior graph-based approaches [135], which require exact API matches and depend on the identification of non-core operations.

But even such a method may fail if an adversary performs one of the three operations in an unanticipated way. To address this limitation, we relax the assumption that all three operations must be observed. We experimentally demonstrate that the presence of *at least two of the three core operations*, when linked by memory congruence, is sufficient to reliably indicate process injection. This enables detection even when the injection sequence is not fully recognized or visible.

This design raises the bar for attackers, who must now conceal or reimplement at least two core operations in unforeseen ways, which is intrinsically harder. At the same time, it eases the burden on defenders by facilitating the identification of variants as techniques evolve, since equivalence classes can be incrementally updated when new implementation variants are discovered.

Notably, using this approach, we spotted an implementation variant of Process Hollowing that is now dominant in the wild, a technique that Starink et al. [135] identify as the most popular but underestimate in prevalence due to strict matching requirements for the analysis.

Finally, we explicitly consider an aspect that prior work treated as orthogonal, namely malware evasion. By integrating our method with a state-of-the-art system for countering evasive behavior and performing an ablation study, we show that malware employing injection techniques is frequently also evasive. This allows us to rectify incomplete conclusions about the real-world prevalence of injection attacks.

We apply our methodology to a longitudinal dataset of Windows malware samples observed between 2017 and 2023. Our corpus consists of 334,885 samples collected from two qualified collections: the VirusTotal Academic dataset (2017–2021) and the VX Underground Bazaar collection (2022–2023). To avoid bias from overrepresented families, we apply balancing techniques and filter out near-duplicates, obtaining a dataset of 56,340 representative malware samples.

Our results show that process injection remains a recurring choice for malware authors, with an average of 16.5% of analyzed samples employing at least one injection technique and a higher prevalence in recent years. A fine-grained temporal analysis suggests that threat actors adapt their technique choices in response to evolving defenses and community knowledge. In a comparative evaluation, we further observe that mainstream open-source and commercial sandboxes fail to detect injection activity in 21.2% of the samples flagged by our approach.

4.1.1 Contributions

This work aims to advance malware analysis systems and foster a clearer understanding of an under-explored yet pivotal class of malware behavior. We make the following contributions:

- A novel methodology for detecting process injection based on the (partial) co-occurrence of three operational primitives linked by memory range congruence. Our approach relaxes strict matching assumptions of existing approaches, thereby improving detection accuracy while remaining robust to implementation diversity and partial observation of the injection sequence in realistic monitoring and analysis settings.
- An experimental longitudinal study estimating the prevalence and evolution of process injection techniques in Windows malware between 2017 and 2023. We provide statistics and behavioral insights into attacker choices, showing how threat actors adapt injection strategies over time in response to defensive knowledge and infrastructure.

4.2 Process Injection Landscape and Threat Model

In this section, we frame the core research problem addressed in this chapter and establish the assumptions that guide our analysis. We first discuss the landscape of process injection techniques to delineate a comprehensive set to study. We then formalize the threat model that defines the scope of our study and the monitoring environment under which our methodology is developed and evaluated.

4.2.1 Process Injection Technique Landscape

As discussed in Section 2.2.2, process injection departs from traditional malware execution models, posing unique detection and analysis challenges.

Despite their diversity, process injection techniques share a common logical workflow consisting of three fundamental stages: memory allocation within a target process, payload placement, and execution redirection [86]. This abstraction provides a stable foundation for reasoning about both classical and emerging variants.

For example, Shellcode Injection [113] allocates a buffer in a victim process, writes shellcode into it, and executes it by creating a remote thread. DLL Injection from disk writes the path of a malicious library into the victim’s memory and forces its loading through a legitimate API call. Process Hollowing [21] creates a suspended process, unmaps its original memory (optionally, as our findings indicate), replaces

it with malicious code, and resumes execution from the new entry point. Other variants hijack existing threads or use Asynchronous Procedure Calls (APCs) [55] to queue payloads for execution within legitimate threads.

For over two decades, process injection has challenged malware analysis. Its effectiveness is recognized not only by threat actors, who extensively use it, but also by security researchers, who have leveraged it as a vector for offensive capabilities [33, 52, 78]. Defensive research faces a unique challenge: the versatility of process injection has produced numerous variants and edge cases, each with its own implementation styles, favored by the extensive set of Windows APIs and advanced features (e.g., transacted files that become opaque to conventional scanning).

Industry and academic work have attempted to structure this space, often from proof-of-concepts [64, 74, 112]. MITRE ATT&CK enumerates 12 sub-techniques covering “true injection” (into running processes) and victim spawning [15]. Klein et al. [86] cataloged true injection techniques based on three primitives: allocation, writing, and scheduling. Starink et al. [135] further expanded this view by grouping 17 techniques into classes and analyzing behavioral characteristics.

Building on these efforts, we focus on process injection as it appears in realistic attack scenarios rather than theoretical constructs. We analyzed technical documentation, red-team reports, malware analysis blogs, and existing research to select 12 representative techniques that: (i) reflect what is actually observed in the wild, (ii) cover true injection and spawning, (iii) align with MITRE ATT&CK categories, and (iv) provide a solid foundation for a thorough and effective study. For a detailed description of these techniques, we refer the reader to Section 2.2.2.

Where appropriate, we consolidated overly granular categories from prior taxonomies [135], for instance by grouping closely related registry-based mechanisms under a broader category aligned with MITRE ATT&CK’s Event Triggered Execution [8], while retaining necessary distinctions. We also refined classifications to separate conceptually distinct behaviors (e.g., distinguishing PE Injection from generic Shellcode Injection) and included additional techniques not covered in [135], like Early Bird, Atom Bombing, and Process Doppelganging to account for more exotic but relevant approaches.

We emphasize that even recent and sophisticated methods recognized by MITRE ATT&CK and the wider security community—such as Reflective DLL Injection [119] and ListPlanting [14]—still adhere to the same three core operations, albeit with additional complexity and conjectures in how these primitives are realized. In practice, many advanced techniques, along with other emerging methods used by adversaries, can be viewed as specializations or extensions of broader families like Shellcode or PE Injection, which we treat as base cases in our methodology.

We argue that our selection of techniques provides a comprehensive yet practical framework for understanding process injection, balancing theoretical completeness with empirical relevance. This selection underpins the empirical evaluation presented later in this chapter.

4.2.2 Threat Model

This work considers a Windows-based environment. Our analysis and experimental methodology focus on user-space malware, namely adversarial code operating within

the standard privileges of a process without exploiting vulnerabilities in the operating system kernel. We deliberately exclude rootkits or samples that abuse kernel vulnerabilities for stealthy operation. This choice is motivated both by scope and by empirical observation: to the best of our knowledge, we are not aware of malware of this kind that leverages process injection in practice.

For execution monitoring, we assume that the dynamic instrumentation framework is trusted and resilient to techniques designed to evade API call logging. By this we mean that the monitoring layer is expected to capture relevant user-space events. This assumption can be further relaxed if orthogonal mitigations—such as those proposed by Assaiante et al. [7]—are employed to strengthen the implementation against stolen-code attacks and similar exploits.

At the same time, we do not assume perfect visibility. In realistic environments, monitoring systems inevitably experience transient blind spots, timing noise, and benign behaviors that resemble malicious activity. Rather than treating these limitations as failures, we view them as inherent characteristics of any practical security deployment and design our analysis accordingly.

Given these assumptions, our objective is not to claim comprehensive prevention of all malware but to improve the ability to recognize when process injection is occurring, to distinguish malicious from benign uses of the same mechanisms, and to provide meaningful insight into how the injection unfolded. This framing positions our work as a bridge between the theoretical systematization of process injection and its practical detection in real-world environments.

4.3 Analysis Methodology

Despite their diversity, virtually all known injection techniques share fundamental behaviors centered around the manipulation of memory areas in a target process. This occurs in three operational steps: obtaining suitable memory regions, populating them with a payload, and scheduling their execution. Understanding these underlying primitives and their relationships provides a powerful lens through which to reason about process injection behavior. As we demonstrate in this work, this abstraction not only clarifies the nature of the problem but also enables the detection of injection instances, even as new techniques continue to emerge and evolve.

The qualifier “virtually all” reflects that this three-step characterization leaves out the techniques systematized as passive injections by Starink et al. [135]. Our stance is that these techniques, as also defined in the MITRE ATT&CK framework [15], primarily pertain to the persistence realm of malware techniques, yet can still be interpreted as injections, albeit using an unconventional vessel to initiate execution. However, by stretching the definition of memory allocation in the first step, our model can accommodate these cases within the same analytical framework.

A key practical advantage of our methodology lies in its simple deployment: it requires only the tracing of selected Windows API calls and their argument values. Although developed as an analytical tool to better understand how malware uses process injection, we demonstrate that this approach has potential applicability to security solutions such as sandboxes and antivirus products due to its simplicity, efficacy in detection, and limited run-time overhead.

4.3.1 Overview

Building on two intuitions that were only partially explored in prior works [86, 135], our analysis approach generalizes the three operational primitives behind process injection and correlates them by identifying a high-level workflow in which congruent memory buffers are manipulated across a sequence of related primitive actions.

This correlation-based reasoning constitutes the conceptual foundation of our analysis methodology. By defining a minimal set of operations sufficient to characterize injection behavior, we obtain a more comprehensive view of the phenomenon while simultaneously improving detection capability.

Consistent with most dynamic malware analysis efforts, including recent proposals in this subfield [135], our approach relies on observing sequences of Windows API calls that a program executes at run time. By reasoning about the occurrence of operations from specific functional classes and examining both their temporal (ordering) and spatial (memory) relationships, we can identify injection patterns across heterogeneous techniques and their implementation variants.

Data capturing is central to our investigation. Depending on the features of the injection technique and the choices of the malware author, each operation primitive may be instantiated using different Windows APIs, which inevitably adds complexity (and increases the risk of missing relevant operations) when tracing. To manage this variability, we build functionally equivalent classes of APIs to realize each primitive action, encompassing both high-level and Native (i.e., `Nt` system calls in Windows terminology) APIs. For instance, to allocate memory in the address space of another process, we treat `VirtualAllocEx` (and its variants) and `NtAllocateVirtualMemory` as equivalent realizations of the same primitive. During execution, we only need to track the arguments and return values of these functions, without resorting to more expensive means such as memory dump analysis or taint tracking explored in other efforts.

Additionally, analyzing the activities of a single process in isolation may be insufficient to fully characterize sophisticated injection attacks. Sophisticated malware may perform multi-level or recursive injections, using an initial victim process to compromise additional targets, fragment the attack, and complicate detection and attribution. In our evaluation, we observe that approximately 2.0% of injecting samples engage in such multi-level injection chains. To capture this behavior, we extend our monitoring instrumentation to track not only the original process but also any child processes it spawns or manipulates, recursively following the injection chain where applicable.

Figure 4.1 presents the overall architecture of our framework. At a conceptual level, the system combines dynamic monitoring with correlation-based analysis: execution traces are collected under controlled instrumentation and examined to identify spatially and temporally coherent injection workflows. The individual components and their internal logic are described in detail in the following sections.

4.3.2 Main Challenges

Accurate detection of process injection requires addressing several critical aspects that, if mishandled, can substantially degrade performance. In what follows, we outline

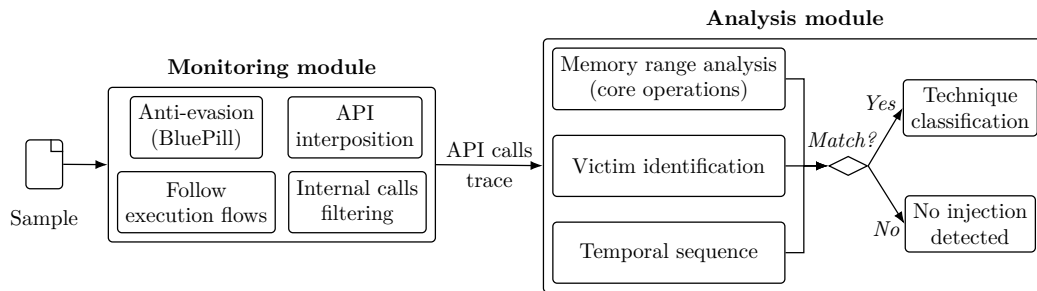


Figure 4.1. Architecture of the framework. The monitoring module instruments program execution and records API-call traces under anti-evasion conditions. The analysis module correlates memory ranges, identifies potential victims, and analyzes temporal sequences to detect coherent injection workflows and classify the corresponding technique.

the three main challenges that guided the design of our methodology: (i) identifying distinctive features of injection behavior, (ii) filtering out false positives arising from legitimate inter-process interactions, and (iii) reasoning about the temporal relationships among relevant operations. The remainder of this section expands on each of these challenges and our corresponding design choices.

Execution monitoring-based detection hinges on the effective interpretation of observed Windows API calls. While we mitigate implementation diversity by reasoning over classes of semantically equivalent actions, adversaries may still implement operations in unforeseen ways or exploit gaps in API monitoring. Moreover, the instrumentation itself may be incomplete due to technical or performance constraints. To address this, we adopt a strategy centered on observing a *minimal* set of core operations that is sufficient to infer the presence of an injection attack. Any additional information available in the trace is then leveraged to refine the classification of the specific technique in use.

To reduce the risk of false positives, we build on insights from Starink et al. [135] by analyzing the memory regions accessed within the victim process and considering only operations that act on *congruent address ranges*. This spatial correlation enables us to discriminate between benign uses of inter-process APIs and malicious injection behavior: only in the latter case do we consistently observe a coherent set of operations targeting the same memory objects.

Finally, since modern malware frequently orchestrates injection using multiple concurrent execution flows [135], we analyze both the ordering and timing of operations to validate their semantic consistency. This temporal analysis is not only crucial for reliable detection but also provides deeper insight into the internal strategy of a malware sample.

Distinctive features. A canonical implementation of process injection would visibly execute the three fundamental steps described earlier, possibly relying on the customary APIs documented for this purpose. In practice, real-world malware and its inherent heterogeneity often result in behavior that deviates from textbook patterns and appears considerably more complex.

A primary source of complexity is implementation diversity. Malware authors tend to substitute functionally equivalent APIs for each step, either to evade security

monitoring or to hinder behavioral fingerprinting. In more advanced cases, they repurpose APIs beyond their canonical usage—for example with atom tables or transacted files—to achieve equivalent effects. Therefore, a detection strategy that requires all three primitives to be observed is vulnerable to adversaries who successfully conceal one step by implementing it in an unanticipated way.

A second complication arises from the fact that some primitives may be implicit or combined. For instance, memory allocation may be unnecessary when malware targets pre-allocated regions, or allocation and writing may occur simultaneously by leveraging the code-sharing mechanism that Windows enables through memory section objects [58]. In such cases, fewer distinct steps appear in the execution trace than a naive model would expect.

Through extensive analysis of real malware samples and validation against benign software, we conclude that *the presence of at least two out of the three primitives* is sufficient to reliably indicate injection behavior. Accordingly, our analysis correlates any two such operations to determine whether an injection has occurred. Any additional information present in the recorded events remains valuable for refining technique classification and better understanding adversarial design choices.

This design preserves detection efficacy without introducing additional false negatives for the analyzed techniques, while remaining robust to implementation variability. It also holds practical value: in our evaluation, 19.9% of injecting samples exhibited only two observable primitives. As demonstrated in our experimental study, this minimal model generalizes across both known and previously unseen injection variants.

To illustrate this scenario, Figure 4.2 presents a representative API trace extracted from a sample of the *neurevt* malware family. The malware first creates a victim process in a suspended state and unmaps its memory using `NtUnmapViewOfSection`, consistent with the canonical Process Hollowing workflow. However, instead of explicitly allocating memory, it creates a shared section via `NtCreateSection` to share parts of its memory with the victim process. As a writing operation, the malware resorts to `NtMapViewOfSection` twice: first locally, to make the section point to the malicious code, and then remotely, to map it into the victim’s address space. At the end of this step, the target process’s memory is filled with the desired malicious code. Execution is ultimately triggered via `SetThreadContext` and `ResumeThread` [116]. In this case, the sequence does not expose an explicit remote allocation primitive. Under our abstraction, the remote section mapping is interpreted as a *write* operation, yielding a workflow consisting solely of writing and execution. Despite the absence of a distinct allocation step, the behavior is correctly identified as injection.

Our methodology is expressly designed to capture such nuanced behaviors. This advantage is particularly evident in Process Hollowing, where our findings diverge from prior models [135]. Comparisons with VirusTotal behavioral reports further reveal that popular sandboxes often fail to flag injection in similar cases.

We further observe that, when modeling injection through behavior graphs, Starink et al. [135] assumed a fixed set of primitives to reach an accepting state. This makes their system less robust when fewer primitives are exercised. Their approach also relies on auxiliary events such as handle acquisition. While necessary for executing process injection, malware can obscure such events more stealthily,

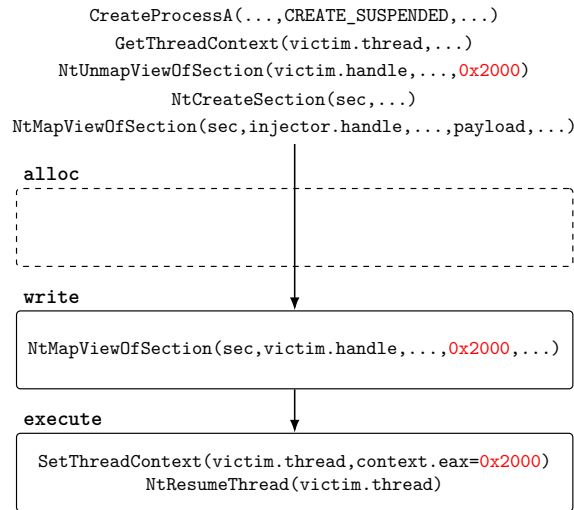


Figure 4.2. API trace of a sample using Process Hollowing based on shared memory sections. The dashed *alloc* stage reflects the absence of an explicit remote allocation primitive. Instead, the remote invocation of `NtMapViewOfSection` is interpreted as a *write* operation under our abstraction. Address `0x2000` denotes the base address of the mapped section and the entry point at which execution is resumed.

potentially causing the analysis to fail. As a result, their system may incur false negatives in realistic settings.

Once an injection is detected, we classify techniques using *distinctive* APIs as primary indicators and operational patterns as secondary identifiers to resolve ambiguities. For example, in the Process Hollowing case shown in Figure 4.2, the technique is classified based on the creation of a suspended victim process.

When distinctive APIs are unavailable or overly general, we differentiate techniques based on the characteristics of memory allocation and writing behavior. For instance, in Shellcode Injection, the payload written into the victim process exhibits characteristics that differ from those of the otherwise similar PE Injection technique. The latter can be identified by its larger payload size or by a greater number of write operations (e.g., one per section). This approach allows us to capture subtle distinctions by closely examining the specific details of each technique.

Relevant operations. A key design consideration in detecting process injection is that many Windows APIs used for malicious purposes are also routinely invoked by benign applications and, in turn, by the system libraries on which they depend. For instance, Windows legitimately allows processes to write to each other’s memory to support functionalities such as debugging, application interoperability (e.g., via atom tables), and system monitoring. Consequently, execution traces frequently contain numerous invocations of these APIs that are unrelated to injection attempts. An effective analysis must therefore filter out such benign activity without compromising sensitivity to real attacks.

Our approach hinges on the principle that injection operations *consistently operate on congruent memory areas*, a concept previously introduced by Starink et al. [135]. During trace analysis, we examine the virtual address regions accessed

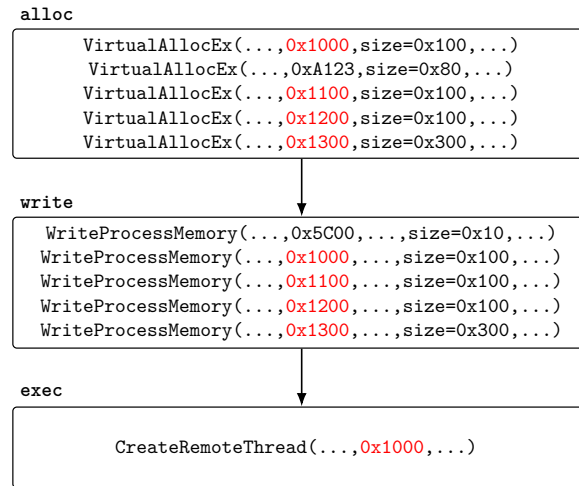


Figure 4.3. API trace illustrating misleading background activity in a sample targeting a victim process. While several allocation and write operations are observed, only those targeting congruent address ranges (highlighted in red) form a spatially coherent *alloc–write–execute* sequence. The allocation at `0xA123` and the write at `0x5C00` are not range-consistent with any other operation and are therefore discarded by our interval-based analysis.

within the target process and require that different primitives operate on overlapping ranges. Concretely, if a memory region is allocated in the interval $[\alpha, \alpha + \beta]$, successful injection necessarily entails writing malicious code within that range and eventually transferring execution control to it.

Noisy background activity can further complicate this analysis. Figure 4.3 presents a representative API trace extracted from a sample of the *salinity* malware family. The trace contains numerous memory operations spanning multiple address ranges, some of which may superficially resemble injection behavior and potentially mislead the analysis by generating false positives. For clarity, operations targeting processes other than the actual victim have already been excluded; without such filtering, additional spurious correlations would arise. Our methodology addresses this challenge by intersecting accessed memory intervals through efficient range lookups, isolating only those sequences that exhibit spatial coherence across primitives and distinguishing valid injection attempts from incidental activity. This interval-based reasoning is applied across multiple processes, enabling the detection of concurrent or chained injections targeting different victims.

Although Windows exposes several functionally similar APIs for disparate tasks, we observe that the functions relevant to injection form a finite set. Even if attackers devise new strategies over time, these must ultimately revolve around—or be reducible to—the three core primitives established in prior literature. While implementation limitations or newly identified APIs may obscure one primitive, our two-out-of-three rule mitigates the risk of false negatives. This design reduces the element of surprise for defenders while preserving robustness against evolving threats.

Temporal correlations. Because modern malware frequently employs multiple concurrent execution flows within a process, spatial reasoning alone is insufficient to validate an injection attempt. We therefore incorporate temporal analysis to ensure that candidate sequences reflect a *logical and coherent progression*. At a high level, we enforce constraints on the primitives involved in process injection, recognizing that memory allocation generally precedes writing, which is followed by code execution. Even when these steps overlap or combine, we preserve the integrity of the logical flow by mapping them to corresponding sub-schemes for validation.

Some techniques involve auxiliary operations that may appear outside the canonical order yet remain semantically meaningful. For example, in DLL Injection from disk, the resolution of the `LoadLibrary` function’s address (which serves as the entry point for the execution unit) can occur at different stages of execution, but it must precede the actual transfer of control. To handle this variability, we encode flexible ordering constraints over a minimal set of essential primitives rather than relying on a rigid sequence of API calls.

By jointly reasoning about spatial congruence and temporal consistency, our methodology captures all valid permutations of relevant API sequences while excluding semantically inconsistent traces. This combination enables high-fidelity detection of suspicious behavior while maintaining precise discrimination among different process injection techniques.

4.3.3 Implementation

To evaluate our approach and gather insights into how threat actors resort to process injection in malware, we designed and implemented an automated dynamic analysis system capable of detecting and dissecting process injection behaviors. The system first identifies the presence of potential injection attempts and subsequently classifies the specific technique(s) employed.

A malware sample performs each step by resorting to one or more Windows API calls. These events are observable through standard API interposition mechanisms commonly used in security solutions. Our system monitors relevant APIs—including common variants such as `A/W` and `Ex` versions—to capture both high-level behavior and low-level implementation details of different injection techniques.

We implement detection routines by instrumenting the target executable at run time. Furthermore, because malware often employs evasion techniques to disrupt analysis systems—leading to incomplete or misleading investigations [100]—we build upon off-the-shelf anti-evasion mechanisms [60] to pursue a more comprehensive analysis.

Our architecture consists of two modules (Figure 4.1):

1. A dynamic monitoring module that executes samples in a controlled environment and records detailed execution traces;
2. An analysis module that processes these traces to detect and classify process injection behaviors.

Monitoring module. Our approach can be integrated with any dynamic analysis system that supports API interception. For this study, we built our implementation

Table 4.1. Windows APIs abstracted into the *alloc*, *write*, and *execute* primitives used by our framework.

Primitive	API	Description
Alloc	VirtualAllocEx	Allocates memory within the virtual address space of the specified process.
	VirtualAllocExNuma	
	NtAllocateVirtualMemory	
Write	WriteProcessMemory	Writes data to an area of memory in the specified process.
	NtWriteVirtualMemory	Maps a view of a section into the virtual address space of the specified process.
	NtMapViewOfSection	
	LoadLibraryA/W	Loads a module into the calling process (used in hook-based injection).
	LoadLibraryExA/W	
	LdrLoadDll	
Execute	GlobalAddAtomA/W	Adds a string to the global atom table (used in atom-based injection).
	NtAddAtom	
	CreateRemoteThread	Creates a thread in a remote process to start execution at a specified address.
	CreateRemoteThreadEx	
	NtCreateThreadEx	
	RtlCreateUserThread	
	QueueUserAPC	Queues a user-mode APC to trigger execution in a target thread.
	NtQueueApcThread	
	NtQueueApcThreadEx	
	SetThreadContext	Modifies the execution context of a specified thread.
	NtSetContextThread	
	<i>paired with</i> ResumeThread	Resumes execution of a suspended thread.
	NtResumeThread	
	NtAlertResumeThread	
	SetWindowsHookExA/W	Installs a hook procedure that may execute attacker-supplied code.
	SetWindowLongA/W	Writes a value to a window memory structure (GUI-based injection).
	SetWindowLongPtrA/W	

on *BluePill* [60], an extensible dynamic analysis framework designed to examine potentially evasive malware. BluePill leverages Dynamic Binary Instrumentation (DBI) through Intel Pin [99], enabling fine-grained monitoring of program execution.

We selected BluePill for its ability to neutralize many known evasion techniques commonly deployed against automated analysis tools. Its extensibility allowed us to interpose on Windows APIs associated with different process injection techniques.

Specifically, we instrumented APIs related to process and memory manipulation to inject code from one process to another, which, as discussed earlier, are sufficient for detection. Additionally, we included APIs that handle registry keys, window memory, hook procedures, and transactional file system (TxF) activity to improve our ability to identify the specific technique in use. In total, we monitor 74 distinct APIs (or 101 when accounting for A/W variants) relevant to process injection.

Table 4.1 lists the APIs used to identify the three fundamental stages of process injection. For additional APIs employed in fine-grained technique characterization, we refer readers to our implementation documentation. We implemented hooks at both the library and system-call levels in BluePill, redirecting intercepted calls to custom handlers that record argument values, return addresses, and execution context.

To capture multi-process behavior, we configured Pin with the `-follow_execv` option, ensuring that our instrumentation extends to child processes and newly created threads spawned by the monitored sample. This is crucial, as many injection

techniques involve creating or manipulating secondary processes.

To reduce noise from benign internal Windows activity, we adopted a filtering strategy inspired by prior work [61]. We consider an API call relevant only if it originates from the code under analysis and eventually returns to it. To implement this policy, we maintain a binary interval tree containing memory ranges corresponding to legitimate Windows DLL code sections as they are loaded and unloaded. If an API call returns to an address within these ranges, we treat it as internal system activity and discard it as irrelevant.

We acknowledge a residual risk of false negatives from filtering internal calls in two rare scenarios. First, an attacker may hijack control flow within a Windows DLL API and redirect execution to malicious code. Such an attack is highly constrained in practice, as it requires precise knowledge of the DLL version in use on the victim system, the presence of a suitable vulnerability, and possibly bypassing Windows Control Flow Guard mitigations. Second, an attacker may set the return address to a gadget in the DLL code that subsequently transfers execution back to malware code. This risk can be mitigated by validating the legitimacy of return addresses at API call or return time.

Albeit far from perfect in terms of transparency, DBI-based techniques offer an effective trade-off between transparency and analytical power. They enable detailed behavioral inspection while supporting the integration of anti-evasion techniques. Other recent studies in malware analysis have similarly relied on DBI frameworks [25, 51], including large-scale analyses of evasive malware. As mentioned before, addressing evasive behavior is necessary to mitigate the risk of missing injection behaviors from samples that, for example, exhibit benign behavior once they recognize an analysis environment [25, 100].

To estimate the impact of evasions, we also implemented a baseline version of our system using Microsoft Detours [40], a library for dynamically intercepting arbitrary Win32 binary functions and adding instrumentation. While Detours provides lightweight and reliable API hooking, it relies on conspicuous trampolines and, unlike BluePill, lacks built-in defenses against virtual environments or delays during the analysis. These characteristics make this implementation a suitable baseline for ablation studies on the impact of evasions.

Analysis module. The analysis module processes the execution traces collected by the monitoring component to identify and classify process injection attempts. We first group monitored APIs into equivalence classes based on their functional role (e.g., memory allocation, memory writing, or execution transfer). This abstraction allows us to reason about behavior at a conceptual level rather than focusing on implementation-specific details of individual APIs. For example, memory may be written using a byte-copying API, by mapping a section into memory, by writing to the atom table, or by requesting Windows to explicitly load a module.

We then correlate memory allocation, writing, and execution operations in the call trace. Our matching mechanism employs *interval lookup* operations to verify the congruence of memory ranges across operations. We perform this correlation on a per-process basis, tracking operations by process identifier and referenced object. Rather than comparing raw handle values directly, we resolve them to unique

underlying object identifiers within the monitoring module, preventing ambiguities due to handle reuse.

We further verify that all core operations (i.e., allocation, writing, and execution) occur on coherent memory addresses, as partial matches could otherwise produce false positives¹. Whenever an operation involves handle values, we treat them as opaque entities in the monitoring module and map them to real object identifiers (whereas Starink et al. [135] attempt to match handle values directly from logs). This approach avoids potential ambiguities arising from handle reuse or similar effects.

We then analyze the temporal ordering of operations using their timestamps to reconstruct the sequence of injection activities. If we detect a valid sequence containing at least two of the three core steps, we perform deeper trace analysis to identify the specific injection technique(s) used.

This final step relies on detecting distinctive API patterns associated with known techniques. For example, Process Hollowing is characterized by the creation of a suspended process followed by memory manipulation and thread context modification.

Through this multi-stage analysis pipeline, our system achieves both broad detection of injection behavior and precise classification of the underlying technique.

4.4 Experimental Setup

This section describes the construction of our malware dataset and the experimental environment used to study process injection techniques. The objectives of this setup are twofold: (i) to obtain a representative and unbiased collection of real-world malware samples spanning multiple years, and (ii) to ensure the analysis environment accurately captures execution behaviors relevant to process injection attacks.

4.4.1 Dataset

To build our dataset, we collected samples from two complementary sources: the VirusTotal Academic dataset [143] and the Vx Underground Bazaar collection [140]. The VirusTotal Academic dataset contains malware samples observed between 2017 and 2021 and has been widely adopted in recent academic research on malware analysis. However, this dataset has not been updated since 2021, as confirmed through private communication with VirusTotal. To extend our study to more recent years, we therefore incorporated samples from the Bazaar collection, a publicly accessible and actively maintained repository widely used by security researchers and practitioners.

Our initial corpus consisted of 334,885 Windows PE executable samples. To impose temporal consistency and reduce noise, we sorted all samples according to their first submission date on VirusTotal, which serves as a practical proxy for the time at which a sample first appeared in the wild. We further discarded Bazaar samples that were flagged as malicious by fewer than 15 antivirus engines, following the same labeling criterion used in the VirusTotal Academic dataset to ensure consistency across sources.

¹For example, the analyzer of [135] checks for matching memory addresses only for allocation and write operations in Process Hollowing and Thread Execution Hijacking. However, semantic validity of execution is not guaranteed unless the entry point is coherently updated in the thread’s context.

Table 4.2. Yearly composition of the malware dataset filtered by *vhash* value (2017–2023).

Year	Total	w/o vhash	w/ vhash	Distinct vhash
2017	26,093	575	25,518	6,804
2018	64,355	3,520	60,835	16,611
2019	59,098	315	58,783	11,475
2020	52,512	97	52,415	16,368
2021	25,176	42	25,134	5,625
2022	59,452	0	59,452	8,555
2023	48,199	0	48,199	8,484
Total	334,885	4,549	330,336	73,922

We then performed deduplication using VirusTotal’s *vhash*, a proprietary static similarity fingerprint designed to cluster related malware variants. This step was necessary to avoid the over-representation of near-duplicate samples, which could otherwise bias our analysis toward specific malware families or campaigns.

To mitigate family-level bias, we ensured that no single malware family accounted for more than 5% of the samples selected per year. We used AVClass [128] to infer plausible malware family labels in a systematic and reproducible manner. After deduplication and balancing, the final dataset contained 56,340 samples.

Representativeness of the malware dataset. A central challenge in large-scale malware studies is selecting a dataset that is both manageable and representative of real-world threats. Our goal was therefore to reduce redundancy while preserving behavioral diversity across seven years of malware observations.

To this end, we leveraged VirusTotal’s *vhash* as the primary feature for clustering similar samples. This fingerprint is designed to group similar binaries even in the presence of code obfuscation or minor modifications [59]. Although the underlying algorithm remains proprietary and undocumented, a recent authoritative study [142] has demonstrated that *vhash* serves as an effective feature for clustering malware based on structural similarity.

We retrieved *vhash* values for all 334,885 samples in our initial dataset. Among these, 4,549 samples lacked a valid *vhash* and were excluded from further analysis. For the remaining samples, we observed that many binaries shared identical *vhash* values within each year, indicating the presence of near-duplicate variants.

To reduce redundancy while maintaining temporal diversity, we selected one representative sample per unique *vhash* value for each year. This process resulted in 73,922 distinct samples prior to family balancing. Table 4.2 summarizes the dataset composition after *vhash*-based filtering.

We next evaluated whether *vhash*-based clustering preserves process injection behavior, which is central to our study. For this purpose, we compared our dataset with that of Starink et al. [135], currently the most comprehensive study on process injection techniques. Because both studies relied on VirusTotal as a primary source, their dataset substantially overlapped with ours. We retrieved the *vhash* values of their published samples and mapped them to our collection, allowing us to reconstruct 95.6% of their dataset within our corpus. From the intersection of both datasets, we randomly selected 990 *vhash* clusters, ensuring equal representation of injecting and

Table 4.3. Behavioral consistency of vhash-based clusters containing samples that exhibit process injection. For each year, the table reports the number of distinct vhash clusters observed in both the full dataset and the random subset of injecting samples. Columns report the number (and percentage) of injecting clusters in which (i) all samples exhibit process injection; (ii) at least 80% of samples exhibit process injection; and (iii) all samples employ the same injection technique.

Year	Distinct vhash clusters	Random injecting clusters	100% injection consistency	$\geq 80\%$ injection consistency	100% technique uniformity
2017	6,804	100	85	86	100
2018	16,611	100	69	81	98
2019	11,475	100	67	83	98
2020	16,368	91	75	82	90
2021	5,625	100	80	90	94
Total	56,883	491	376 (76.6%)	422 (86.0%)	480 (97.8%)

non-injecting samples. This yielded 14,629 samples for comparative analysis.

Our analysis showed strong behavioral consistency within vhash clusters. Table 4.3 provides a detailed breakdown of vhash-based clustering for samples performing process injection according to our methodology. Among the 491 vhash values associated with injecting samples, 76.6% exhibited process injection behavior across all samples in the cluster. When relaxing this requirement and allowing injection behavior to appear in at least a subset of samples within a cluster, over 86.0% of clusters showed injection behavior in at least 80.0% of their samples. Furthermore, 97.8% of clusters were uniform with respect to the specific injection technique employed. For clusters labeled as non-injecting by our methodology, 99.6% consistently exhibited non-injecting behavior, with only two exceptions.

These findings indicate that vhash is a reliable proxy for clustering malware with similar process injection behaviors. Consequently, our use of vhash-based deduplication is methodologically sound and preserves meaningful behavioral diversity while eliminating redundant samples and improving scalability.

4.4.2 Analysis Environment

Our analysis framework was deployed on five VirtualBox virtual machines hosted on a local server equipped with an Intel Xeon E5-2620 v3 CPU @ 2.40 GHz (24 cores) and 64 GB of RAM. Each virtual machine was configured with 2 GB of RAM and 4 logical CPUs and ran Microsoft Windows 7, an operating system chosen to maximize compatibility with malware samples spanning different years.

To approximate a realistic victim system, we introduced typical user artifacts such as documents, browser history, and commonly installed applications, following established best practices for malware experimentation environments [105]. In addition, we included a local network service simulator that emulated common internet services, enabling malware samples to generate and interact with network traffic without granting them unrestricted external connectivity.

Each virtual machine was managed by a centralized analysis controller responsible for orchestrating the execution workflow. Specifically, the controller instantiated the VM, transferred the sample to an in-guest analysis agent, monitored execution,

and collected the resulting behavioral traces. After each run, the virtual machine was restored to a clean state using a live snapshot to prevent cross-contamination between experiments. Together, these provisions substantially reduced the risks of executing potentially harmful malware.

To determine an appropriate execution window, we conducted preliminary experiments on 26,391 samples (46.8% of the final dataset), executing each for up to eight minutes. We observed that 95.2% of samples initiated their first process injection within 1.5 minutes, and all but 215 samples did so within two minutes. These values align with prior work [89], which shows that most malware behaviors manifest within the first two minutes of execution, with little additional information gained for longer observation periods.

Given that process injection typically occurs early in execution—often immediately after any initial evasive checks—we selected a four-minute execution window for the full study. This duration provides sufficient margin to capture relevant behaviors while accommodating potential delays introduced by DBI instrumentation overhead.

4.5 Experimental Findings

This section reports the results of our experimental analysis of process injection in modern malware. The evaluation is organized to address the following research questions:

RQ1: What are the most common process injection techniques adopted by modern malware? Does the release of information about these techniques influence their adoption?

RQ2: Is there a correlation between malware families and the injection techniques they exploit? Have the injection techniques adopted by each family evolved over time?

RQ3: Can the proposed methodology be effectively integrated into practical security solutions, and if so, with what level of performance (e.g., false positives)?

4.5.1 Adoption of Process Injection Techniques

General trends. We observed evidence of process injection in 16.5% of the samples (9,314) executed under our DBI-based monitoring system. The percentage ranged between 10.7% and 29.1% across different years (Table 4.4).

As described in Section 4.4.1, our dataset combines samples from two sources: VirusTotal (2017–2021) and Bazaar (2022–2023). Since we observed appreciable differences in the frequency of injection attempts between the two collections, we additionally tested 8,241 balanced Bazaar samples from 2021 to establish a consistent temporal ground for comparison.² These samples were generally more “active” in terms of conspicuous activities, with 31.6% exhibiting injection attempts.

²These additional samples are used only for cross-collection comparison and are not included in the aggregated statistics reported elsewhere in the study.

Table 4.4. Yearly prevalence of process injection in the balanced malware dataset (2017–2023). For each year, the table reports the total number of samples and the number (and percentage) of samples exhibiting process injection. The final row reports aggregated totals over the entire observation period.

Year	Samples	Injection
2017	5,859	1,321 (22.6%)
2018	14,650	1,800 (12.3%)
2019	9,978	1,181 (11.8%)
2020	5,321	567 (10.7%)
2021	5,042	553 (11.0%)
2022	8,067	2,348 (29.1%)
2023	7,423	1,544 (20.8%)
Total	56,340	9,314 (16.5%)

We attribute the discrepancy between VirusTotal and Bazaar primarily to differences in how the collections are constructed. The VirusTotal Academic dataset consists of samples submitted to the VirusTotal platform over unspecified time windows within each year (with one or two collections per year) and flagged by at least 15 antivirus engines. As such, it likely includes second-stage payloads that by nature do not need to resort to process injection, suspicious but potentially benign files, or infected installer programs.

By contrast, the Bazaar collection features samples systematically gathered and labeled as malicious by security practitioners on a monthly basis. These differences make both datasets valuable for general malware studies and plausibly explain the higher prevalence of intercepted injection attempts in the Bazaar samples.

Another interesting aspect relates to malware evasion. Recent longitudinal studies have documented a subtle but steady increase in malware evasion techniques over time [100, 67]. To assess how evasion might affect our measurements, we evaluated our approach using a baseline implementation based on Microsoft Detours on the balanced collection from 2019, which is the most recent year overlapping with the longitudinal studies cited above. Under the Detours-based system, only 6.2% of samples exhibited injection behavior, compared to 11.8% detected by the DBI-based approach. This substantial gap highlights the impact of evasive techniques on malware analysis and suggests that the actual prevalence of process injection is likely higher (i.e., several samples likely evaded BluePill as well). This finding also reinforces the limitations of insights from prior injection-related literature.

Techniques distribution and evolution. Table 4.5 reports the distribution of process injection techniques identified by our method from 2017 to 2023. Several techniques were observed throughout the entire study period, suggesting (as expected) that their first appearance likely predates our analysis timeframe. Likewise, techniques detected near the end of our dataset (December 2023) are likely to remain in use, as malware authors tend to retain methods that remain operationally effective while phasing out those that become obsolete for efficacy.

Our data reveal two high-level patterns: certain techniques are markedly more prevalent than others, and many of these techniques tend to cluster within specific

Table 4.5. Yearly distribution of process injection techniques in the balanced malware dataset (2017–2023). Values report absolute counts; percentages are computed with respect to the number of injection samples in the corresponding year.

Injection Technique	2017	2018	2019	2020	2021	2022	2023	Total
Process Hollowing	919 (69.6%)	880 (48.9%)	414 (35.1%)	148 (26.1%)	152 (27.5%)	2,104 (89.6%)	1,482 (96.0%)	6,099 (65.5%)
PE Injection	61 (4.6%)	357 (19.8%)	246 (20.8%)	191 (33.7%)	47 (8.5%)	172 (7.3%)	51 (3.3%)	1,125 (12.1%)
Shellcode Injection	74 (5.6%)	453 (25.2%)	200 (17.0%)	75 (13.2%)	61 (11.0%)	25 (1.1%)	5 (0.3%)	893 (9.6%)
Passive Injection	35 (2.7%)	91 (5.1%)	237 (20.1%)	151 (26.6%)	224 (40.5%)	45 (1.9%)	9 (0.6%)	792 (8.5%)
Hook Injection	77 (5.8%)	83 (4.6%)	137 (11.6%)	28 (5.0%)	33 (6.0%)	19 (0.8%)	4 (0.3%)	381 (4.1%)
Thread Execution Hijacking	9 (0.7%)	41 (2.3%)	10 (0.9%)	3 (0.5%)	58 (10.5%)	2 (0.1%)	0 (0.0%)	123 (1.3%)
Early Bird	161 (12.2%)	0 (0.0%)	1 (0.1%)	0 (0.0%)	1 (0.2%)	0 (0.0%)	1 (0.1%)	164 (1.8%)
Disk-based DLL Injection	3 (0.2%)	3 (0.2%)	18 (1.5%)	0 (0.0%)	3 (0.5%)	5 (0.2%)	1 (0.1%)	33 (0.4%)
APC Injection	0 (0.0%)	1 (0.1%)	0 (0.0%)	4 (0.7%)	1 (0.2%)	0 (0.0%)	1 (0.1%)	7 (0.1%)
Atom Bombing	0 (0.0%)	1 (0.1%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.0%)
Process Doppelg�nging	0 (0.0%)	1 (0.1%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	1 (0.0%)
Extra Window Memory Injection	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)

time periods, suggesting campaign-driven adoption rather than uniform use over time.

The most prevalent techniques are Process Hollowing (65.5%), PE Injection (12.1%), and Shellcode Injection (9.6%). Process Hollowing stands out due to its effectiveness in achieving stealth with relative simplicity, making it a preferred choice for malware writers. This is also reflected in how it exceeds the combined total of all other techniques in 2017, 2022, and with an even larger margin in 2023.

Beyond longitudinal trends, we observed significant vertical evolution within individual techniques. Process Hollowing offers a representative example: by focusing on fundamental behavioral primitives rather than fixed API sequences, our approach exposed implementation variants that were overlooked by prior studies.

In the traditional workflow, Process Hollowing unmaps memory from the target process, a step that gives the technique its defining characteristic. However, we observe a shift in attacker behavior: instead of hollowing out the original executable, attackers modify the victim’s address space directly. That is, they allocate a new memory region at a system-assigned address and write the malicious code there, leaving the original memory unmodified. This *no-unmap* variant, though occasionally mentioned in security blogs in 2016 [1] and 2022 [32], appears in our tests as early as 2017 with significant prevalence (36.0% of cases), and by the following year it had surpassed the classic version in frequency.

We interpret this shift as a preference for implementing relocation machinery to accommodate the payload within the remote addresses assigned randomly by Windows, rather than requesting the allocation at a fixed base address once the address space has been hollowed out. While relocation itself is not technically challenging, it significantly enhances stealth. The *no-unmap* variant avoids conspicuous API calls like `NtUnmapViewOfSection`, and leaves fewer detectable memory artifacts, making it harder for security solutions to identify anomalies when checking the integrity of loaded code modules. This evolution highlights why Process Hollowing remains dominant among malware authors—its versatility and adaptability have given rise to new variants, complicating detection and analysis efforts for security professionals.

While Process Hollowing occurrences increased over the years, other techniques appeared less frequently. We note that the choice of injecting more structured payloads (as is the case with Process Hollowing) is also reflected by the prevalence of PE Injection over simpler shellcode. This shift may be influenced by the abundance of white papers detailing advanced injection techniques [75, 110, 118].

Most techniques exhibit peak usage within specific, brief periods, likely driven by active malware campaigns where cybercriminals test new tactics. For example, Early Bird injection saw significant use in 2017, accounting for 12.0% of the samples, but sharply disappeared in subsequent years. Various sources indicate that Early Bird gained interest around 2018, as shown by the availability of security blog posts [49, 114, 125]. This aligns with the intuition that attackers often exploit techniques in earlier campaigns while they are novel and harder to detect, abandoning them once defensive tooling matures.

This dynamic may also explain the near disappearance of APC Injection, which, as an early technique exploiting asynchronous procedure calls in victim threads, was well-known and easily detected by AVs since its identification in 2018 or earlier [77].

Passive Injection techniques that manipulate registry keys showed their peak in activity between 2019 and 2021. This cluster aligns with the findings of Starink et al. [135], although they noted a rising adoption rate. This period also overlapped with the COVID-19 pandemic, during which the number of cyberattacks and threats grew exponentially, prompting attackers to explore alternative methods for executing malicious code and persistence mechanisms [74, 104].

DLL Injection from disk appears rarely in our dataset, likely because modern security systems can easily detect this approach. Nevertheless, the loading of a malicious DLL can still be accomplished in other ways, as with Reflective DLL Injection, which we classify under PE Injection. Analogously, we observed only a small fraction of samples performing Hook Injection, suggesting a broader shift toward fileless techniques that avoid obvious artifacts that may trigger AV alerts.

Finally, Atom Bombing and Process Doppelganging appeared rarely, with instances recorded only once in 2018. Similarly, we found no cases of Extra Window Memory Injection. Although proof-of-concept implementations exist [39, 73, 122], our tests confirmed that these techniques can be exploited in real-world settings and detectable using our methodology.

We assess the practical impact of detecting injection based on observing only two out of three primitive events. For 19.9% of samples flagged as injecting, the execution trace contains only two primitives. This effect is more pronounced for Process Hollowing (36.1%) and PE Injection (29.9%), which we attribute to the emergence of implementation variants that compress or obscure one step of the classic workflow.

Comparison with prior work. Our measured injection prevalence differs from that reported by Starink et al. [135], who observed injection in 8.3% (2017), 13.4% (2018), 13.6% (2019), 5.1% (2020), and 6.1% (2021) of samples. We attribute this discrepancy to three main factors:

- The authors did not balance the VirusTotal Academic dataset (which was their only dataset), leading to over-representation of certain families in their statistics;
- Their system lacks dedicated anti-evasion provisions, a limitation the authors acknowledge and partially mitigate by deploying their system atop a virtual

machine introspection-based sandbox (which remains vulnerable, e.g., to timing attacks to detect VMs or stall analyzers);

- Our primitive-based methodology captures a broader range of injection behaviors, including the *no-unmap* variant of Process Hollowing, which accounts for a significant fraction of injection cases.

Although the datasets are not identical, we applied our balancing procedure (Section 4.4.1) to their results and achieved 95.6% coverage of their dataset while identifying 64.1% more samples exhibiting injection behavior. Due to the generally higher percentages (and the more techniques) we observe, these results suggest that our study provides a more comprehensive view of injection prevalence and diversity.

Frequency of use and target processes. Another interesting metric is the number of injection techniques employed by each sample. The percentages for single years shown in Table 4.5 do not sum to 100% because some samples use multiple techniques. Specifically, 96.8% of samples employed exactly one injection technique, while 3.0% used two, and the remaining 0.2% utilized up to three. When multiple techniques are used, authors often diversify choices.

Malware may then attempt to inject multiple processes before achieving successful execution. Our observations reveal that 85.2% of samples executed a single injection sequence, 5.1% attempted two, 1.2% carried out three, and a minority tried four or more, with some failure cases looping over all running processes. These patterns suggest that malware authors typically prefer to rely on a single injection technique, targeting a limited set of processes to minimize system disruption and maximize the chances of a successful attack.

The most frequently targeted processes are those with high trust levels or commonly executed by the system. For simpler techniques like Shellcode and PE Injection, 83.2% of attacks target existing processes rather than creating a new one. The most common victims are *svchost.exe* (48.3%) and *explorer.exe* (33.2%), with smaller fractions targeting user applications such as *notepad.exe*. With these injection techniques, we observe no particular preference when the malware creates the victim process itself. By construction, Process Hollowing creates the victim process in a suspended state: therefore, identifying its typical targets can offer interesting insights. In 28.6% of cases, malware hollows common Windows background processes such as *svchost.exe* and *taskhost.exe*.

More interestingly, in 45.3% of cases, malware spawns a copy of itself in suspended state, solely for injection. After consultation with a senior malware analyst, we interpret this as a defense evasion strategy: executing the payload in a separate process can reduce scrutiny by endpoint defenses, and injecting into a child process is operationally easier because the malware already holds a valid handle to it (and opening a process with memory write privileges is a conspicuous action).

RQ1. Process injection is a relevant and dynamically evolving phenomenon in modern malware. In our analysis, 16.5% of samples employed at least one injection technique, with Process Hollowing and PE Injection as the most common choices. The temporal clustering of techniques and the emergence of variants (e.g., *no-unmap* hollowing) indicate that adoption is shaped by both attacker innovation and the dissemination of community knowledge.

4.5.2 Families and Process Injection Techniques

We now turn to the correlations between malware families and their use of process injection, with the aim of understanding whether injection behavior is characteristic of specific families and how it evolves over time.

Prevalence of injection across families. Our dataset contains samples belonging to 2,667 distinct malware families. Among these, 576 families (21.6%) include at least one sample that exhibits process injection behavior. Additionally, 7,219 samples (12.8%) could not be associated with any family by AVClass; these samples are excluded from the family-level analysis to avoid introducing classification noise.

Table 4.6 summarizes the injection behavior of the 20 most prevalent families in our dataset. Three key observations emerge from this analysis.

First, the year in which a family first appears in the dataset does not necessarily coincide with the year in which it first adopts process injection. For example, the *virlock*, *reline*, and *zard* families initially do not exhibit injection behavior but later integrate it into their operational repertoire.

Second, family size (column #) does not directly correlate with the likelihood of using injection. Some highly prevalent families exhibit limited or no injection activity, while others rely on it extensively. Notably, the *reline* family—associated with user account information stealing—exhibits injection in 99.3% of its samples, followed by the *stop* family at 64.5%.

Third, high injection prevalence does not necessarily imply high technique diversity. For instance, the *ramnit* and *bladabindi* families both employ up to two different techniques, yet their overall injection rates differ substantially, suggesting that adoption patterns are driven by family-specific operational goals rather than a uniform strategy.

Consistency of injection usage within families. We next assess whether malware families maintain consistent use of injection after its initial adoption.

Among families with multiple samples exhibiting injection activity, only 39 out of 482 (8.1%) consistently use injection across all their samples. In most cases, the rate of samples performing injection varies widely within the same family. These findings suggest that injection is often optional or context-dependent rather than a defining structural characteristic of a family. When considering families active over multiple years, only 20.6% display injection behavior in every year in which they appear. This supports the view that malware families evolve their tactics in response to defensive countermeasures or shifting objectives.

Table 4.6. Top 20 malware families in the dataset, ranked by number of samples. For each family, the table reports: total number of samples (**#**); year of first appearance in the dataset (**Date**); year of first observed injection activity (**First injection**); number and percentage of samples exhibiting injection (**# Injection**); and the maximum number of distinct injection techniques used by any single sample (**# Techniques**). A dash (–) indicates that no injection activity was observed.

Family	#	Date	First injection	# Injection	# Techniques
virlock	1,657	2017	2018	25 (1.5%)	2
upatre	1,288	2017	2017	10 (0.8%)	1
virut	1,278	2017	2017	84 (6.6%)	2
sality	1,203	2017	2017	436 (36.2%)	4
vobfus	901	2017	2017	45 (5.0%)	1
bladabindi	771	2017	2017	34 (4.4%)	2
dealply	757	2017	–	0 (0.0%)	0
reline	723	2021	2022	718 (99.3%)	1
stop	722	2021	2021	466 (64.5%)	1
ramnit	649	2017	2017	329 (50.7%)	2
mokes	641	2019	2019	152 (23.7%)	1
allapple	603	2017	–	0 (0.0%)	0
zard	600	2018	2022	75 (12.5%)	1
ursu	570	2017	2017	49 (8.6%)	2
sfone	513	2017	–	0 (0.0%)	0
gamarue	472	2017	2017	166 (35.2%)	3
zbot	471	2017	2017	133 (28.2%)	1
tofsee	470	2017	2017	30 (6.4%)	1
strab	470	2022	2022	148 (31.5%)	1
gandcrab	469	2018	2018	25 (5.3%)	2

Table 4.7 reports the set of techniques observed for each of the most prevalent families across years. Several families exhibit dynamic behavior: some adopt injection gradually, others temporarily abandon it, and some shift between techniques. For example, the *tofsee* family employs injection in its first two years, ceases its use for a period, and later resumes it. Such patterns indicate that injection is not a static trait of a family but rather an adaptable capability.

Consistency in technique selection. Given that samples from the same malware family often exhibit similar malicious behavioral traits, a further question is whether families consistently rely on the same injection techniques once they adopt injection.

To address this, we analyzed families exhibiting injection in at least two years, excluding single-year families to reduce noise. For each family, we computed the intersection of techniques used across its samples. Our analysis shows that for 138 of 251 (55.0%) families exhibiting injection behavior over multiple years, at least one technique was used consistently across all their samples. This indicates a meaningful degree of continuity in operational preferences.

However, the remaining families show substantial variability. Some alternate between affine techniques (e.g., PE Injection and Process Hollowing), while others switch to entirely different strategies over time. This suggests that while some families favor stable toolchains, others actively adapt to evolving detection and mitigation to achieve their objectives and maintain effectiveness.

Notably, Table 4.7 reveals multiple cases in which families initially rely on PE

Table 4.7. Process injection techniques observed per year in the top malware families, ranked by number of samples. **SC**: Shellcode Injection; **PE**: PE Injection; **PH**: Process Hollowing; **HK**: Hook Injection; **APC**: APC Injection; **TH**: Thread Execution Hijacking; **EB**: Early Bird; **P**: Passive Injection; **—**: No samples available for that family in the corresponding year.

Family	2017	2018	2019	2020	2021	2022	2023
virlock		PE	PE,HK	PE	PE	PE	PE,PH
upatre	PH	PH,P	PH			PE,PH	
virtut	SC,PH,P	SC,PE,PH,HK	SC		PH		
salinity	SC,PE	SC,PE,PH,P	SC,PE,PH	SC,PE	SC,PE	SC,PE,HK,P	SC,PH
vobfus	PH	PH	SC,PE,PH	PH	PH	PH	
bladabindi	PH,HK	DLL,PE,PH	PH	PH,HK		PH,HK	PH
reline	—	—	—	—		SC,PE,PH	PH
stop	—	—	—	—	PH	PE,PH	PE,PH
ramnit	PE,PH	PE,PH,APC	SC,PE,PH,HK	PE,APC	PE	PE	—
mokes	—	—	PH			SC,PH	PH
zard	—					PE,PH	PE,PH
ursu	PH,EB	SC,PH,TH	PH	PH	PH		—
gamarue	DLL,PE,PH	PE,PH	PH	SC,PE,PH	PE,PH	SC,PE,PH	SC,PH
zbot	SC,PE,PH,EB	SC,PE,PH	PE,PH	PE,PH		SC,PE,PH	PH
tofsee	PH	PH				PE,PH	PH
strab	—	—	—	—	—	PE,PH	PH
gandcrab	—	SC,PH,TH					

Injection but later transition toward Process Hollowing. This shift is consistent with our earlier findings: Process Hollowing has become increasingly attractive due to its flexibility and stealth, particularly for executing full malicious payloads.

In line with the observations of Starink et al. [135], our results confirm that the choice of injection technique alone is insufficient for reliable family attribution.

RQ2. Within a malware family that employs process injection, it is common to observe both samples that use injection techniques and samples that (in our executions) do not. While a substantial subset of families consistently rely on the same techniques, many others adapt their strategies to current trends over time. This variability indicates that technique selection alone is not a reliable discriminator for malware family identification.

4.5.3 Feasibility of Downstream Integration

The evaluation presented so far has focused on the value of our methodology for understanding the dynamics of process injection from a longitudinal perspective. We now consider an orthogonal question: whether the proposed approach is sufficiently reliable, precise, and operationally practical to be integrated into existing security solutions to strengthen detection and response capabilities against injection-based malware techniques.

We address this question along three dimensions: (i) the risk of false positives on legitimate software, (ii) the potential to reduce false negatives in widely used analysis tools, and (iii) the run-time overhead introduced by our approach.

Table 4.8. Widely used applications included in the second benign-software experiment.

Category	Programs
Browsers (3)	Chrome, Firefox ESR, Internet Explorer
Editors (3)	Notepad++, ONLYOFFICE, Sublime Text
Media (4)	Audacity, IrfanView, VLC, Windows Media Player
Microsoft Office (4)	Excel, OneNote, PowerPoint, Word
PDF readers (3)	Adobe Reader, PDF-XChange Editor, SumatraPDF
Utilities (3)	Skype, Zotero, 7-Zip

Risk of false positives on legitimate software. False positives constitute a critical concern for any technique intended for deployment in production security systems, as incorrect alerts on benign software may disrupt users and undermine trust in automated defenses.

This aspect is particularly relevant in our case because, unlike prior work (notably Starink et al. [135]), we relax strict pattern matching in favor of a behaviorally grounded detection model. While this design choice substantially reduces false negatives on real-world malware samples, it may in principle increase the risk of flagging legitimate software that performs uncommon but benign memory operations. We therefore conducted two controlled experiments on legitimate software (hereafter referred to as *goodware*) to estimate this risk.

In the first experiment, following the methodology of Starink et al. [135], we collected 425 executables from the Windows OS build (`C:\Windows\System32`) and 97 popular portable applications distributed via Chocolatey [47]. In line with recent longitudinal studies on malware behavior [100, 67], we ensured dataset integrity by excluding any binary flagged as malicious by at least one antivirus vendor on VirusTotal. This filtering step removed 55 programs exhibiting highly privileged behaviors typically abused by malware (e.g., debugging, monitoring, and security utilities for administrators). The final dataset therefore comprised 467 executables.

None of these binaries triggered injection alerts under our monitoring configuration. This result suggests a low empirical risk of false positives in common software deployment scenarios involving standard system components and widely distributed applications. Regarding the excluded security utilities, we note that such programs are generally obtained from trusted sources by experienced users, and modern AV and EDR platforms typically include explicit handling logic for these tools, indicating that their behavior is well understood within the security ecosystem.

Nevertheless, legitimate applications with complex execution models may perform operations that partially overlap with injection primitives, particularly when manipulating executable memory regions or loading objects across multiple processes. To evaluate this scenario, we conducted a second experiment involving 20 widely used applications listed in Table 4.8, spanning browsers, editors, media tools, Microsoft Office applications, PDF readers, and common utilities. Some of these programs are intrinsically harder to analyze: for instance, some may manipulate memory across multiple processes for the sake of displaying contents in a more secure way, using sandboxing mechanisms and isolation through multi-processing.

In this second experiment, we observed three false positives, all attributable to a specific flavor of secure multi-process architecture. In both Chrome and Firefox

ESR, isolated components such as rendering engines and plugins execute in separate processes. These browsers spawn a suspended copy of themselves, allocate executable memory within the new process, write code to that region, and subsequently resume execution. This sequence closely matches the low-level primitives associated with Process Hollowing. The crucial distinction, however, is that execution is resumed at the original entry point rather than redirected to attacker-controlled code. A similar behavior was observed in Skype, which is built on Electron and therefore relies on the Chromium engine underlying Chrome.

From a strictly behavioral perspective, our system correctly identifies this sequence as injection-like, since it observes the same memory allocation, write, and execution primitives commonly leveraged by malware. However, the current implementation does not incorporate a contextual validation layer capable of assessing the legitimacy of the resumption point. In operational AV and EDR systems, such low-level signals typically serve as high-sensitivity indicators that trigger enhanced monitoring rather than immediate blocking decisions, with final classification determined through additional contextual evidence.

In our present prototype, which is primarily designed for injection analysis in malware samples, we perform a range congruence check when `ResumeThread` is invoked. If the resumption address does not fall within the allocated and written memory region, the corresponding primitive is discarded; however, the behavior may still be classified as injection if the remaining primitives satisfy the detection logic. This design limitation could likely be mitigated by incorporating contextual knowledge that would eliminate these residual false positives in practice.

An additional empirical observation from this study concerns Microsoft Word, which legitimately performs hook injection via `wwlib.dll` to display license activation dialogs (`idhook=-1`). This instance does not constitute a false positive, but rather illustrates a textbook use of a documented Windows mechanism. Determining whether such a dynamically loaded DLL is malicious requires inspection of its contents and provenance, typically through integrity verification or vendor-based trust assessment. This example further highlights the distinction between detecting injection primitives and establishing malicious intent.

Overall, process injection is characterized by distinctive interactions with executable memory regions that benign applications rarely perform in the same combination and with the same control-flow semantics as malware. Nevertheless, not all executable memory manipulation is malicious, and advanced legitimate software may employ mechanisms that overlap with attacker techniques. These results therefore suggest that our method is well suited as a high-precision detector of injection primitives, provided it is embedded within a broader, context-aware decision pipeline of the type already implemented in modern AV and EDR systems. Under such integration, the residual false positive rate observed in our experiments would be expected to decrease further without compromising detection sensitivity.

Reducing false negatives in existing tools. We now evaluate whether our approach could improve the effectiveness of existing malware analysis platforms.

For closed-source solutions, we employed the VirusTotal Academic API to compare our detections against behavioral reports produced by commercial sandboxes

and other scanners available on the platform. At the time of our study, 2.8% of samples did not include behavioral reports and were therefore excluded from this analysis. Among the remaining samples, 2,172 instances (24.0%) identified by our approach were not labeled as performing injection by commercial tools. Notably, 29.4% of these missed cases were detected by our method using only two behavioral primitives. This result underscores the practical benefit of relaxing rigid pattern requirements in favor of a more flexible primitive-based detection logic.

For open-source solutions, we analyzed reports generated by the CAPE sandbox [42], which extends Cuckoo [126] and employs the *malfind* plugin [66] to identify injected code, alongside several anti-evasion mechanisms. While VirusTotal aggregates heterogeneous vendor reports and may map observed behavior to MITRE ATT&CK techniques, CAPE emphasizes memory artifacts extracted during dynamic execution. Although *malfind* can highlight suspicious memory regions, it may still fail to capture injection events that do not leave easily identifiable memory signatures. After excluding 59 samples that exceeded CAPE’s processing limits, we found that 1,694 samples (18.3%) detected by our system were not reported as injection by CAPE. In 13.7% of these cases, our detection again relied on only two primitives, reinforcing the value of our decomposed behavioral model.

Taken together, these findings reveal systematic detection gaps in both commercial and open-source tools widely used by practitioners. On average, approximately 21.2% of samples performing injection—according to our ground-truth reconstruction—were not flagged as such by existing systems. While differences in reporting semantics and analysis depth must be acknowledged, the magnitude and consistency of this gap indicate a clear opportunity for improvement.

Our results further suggest that integration of our approach into these platforms would be technically feasible. Many sandboxes already intercept a substantial subset of the API calls required by our method, implying limited additional instrumentation overhead. More broadly, this evaluation indicates that decomposing process injection into granular behavioral primitives—rather than relying on rigid patterns—can meaningfully reduce false negatives and improve the robustness of malware analysis frameworks. Such integration would enhance detection sensitivity while remaining compatible with existing analysis pipelines.

Runtime overhead and deployment feasibility. Finally, we evaluate the runtime overhead introduced by our approach, as performance constraints are critical in real-time security products such as AV and EDR systems. We adopt the experimental setup of prior work on API tracing overhead [61], using a worst-case benchmark suite designed to stress call-intensive workloads.

Specifically, we execute Wine conformance tests across 12 widely used Windows DLLs. These tests exercise a large number of APIs relevant to both benign and malicious software, including low-level operations frequently involved in process injection mechanisms. This setting represents a stress scenario in which the density of instrumented API calls is intentionally high, thereby providing an upper-bound estimate of instrumentation overhead.

To isolate the cost specific to our injection detection logic, we compare the execution time of our system against a baseline DBI tool configured with the same

Table 4.9. Wine conformance test workload and corresponding runtime overhead percentages.

Test	Workload	Baseline tool	Injection detection	Overhead
		Time(s)	Time(s)	
kernel32	atom	8.147	8.379	2.85%
advapi32	registry	8.48	8.698	2.57%
user32	edit	14.427	14.645	1.51%
shell32	shellpath	20.415	20.673	1.26%
kernelbase	sync	4.383	4.437	1.23%
oleaut32	safearray	5.498	5.563	1.18%
ole32	storage32	33.994	34.391	1.17%
iphapi	default	5.249	5.308	1.12%
ws2_32	protocol	6.804	6.872	1.00%
gdi32	metafile	22.751	22.968	0.95%
crypt32	encode	10.255	10.345	0.88%
wininet	internet	20.817	20.923	0.51%

API hooks but without any injection analysis component. The baseline therefore reflects the cost of monitoring the relevant APIs—an operation already required in many security solutions—without performing higher-level behavioral reasoning. This comparison ensures that the measured overhead captures only the additional processing introduced by our detection methodology. Table 4.9 summarizes the workload details and corresponding execution times for each test.

The largest slowdowns are observed in the `kernel32` and `advapi32` test suites, as these libraries make extensive use of allocation, memory-writing, and process-management APIs that are central to injection detection. In particular, system libraries often rely on low-level primitives even for routine functionality, resulting in a higher density of instrumented calls than in typical end-user applications. Across all benchmarks, we observe an average runtime overhead of 1.35%. Given that this figure is obtained under deliberately stress-intensive conditions, it provides strong evidence of the practical feasibility of integrating our approach into real-time security systems. We note that additional engineering optimizations could reduce this cost even further in production deployments.

RQ3. Our methodology can complement existing malware analysis solutions, which missed approximately 21.2% of the injection cases we identified. Given its low overhead and improved coverage, we argue that future security tools could benefit from adopting more principled, primitive-based analyses to better characterize real-world process injection behavior.

4.6 Discussion

In this work, we explored the elusive nature of process injection in Windows malware, drawing insights into its adoption over seven years (2017–2023). Despite the variety of process injection techniques and their implementation variants, our approach was able to analyze them by focusing on the minimal set of core operations sufficient to detect an injection attack.

From a longitudinal perspective, our findings indicate that process injection remains an impactful and favored attack vector for adversaries seeking to bypass defenses. We observed a trend toward increasingly evolved, stealth-enhancing methods. This evolution underscores the need for future detection solutions to capture the fundamental properties of process injection in order to adapt rapidly to new developments. Moreover, our results suggest that threat actor strategies may have shifted in response to what the white-hat community discovered about the attack vectors in use.

The continued popularity of process injection likely stems from its proven effectiveness, despite being a well-known problem in the security community for over two decades. Because the technique exploits legitimate operations exposed for benign programs, security vendors must exercise caution when blocking such operations to avoid interfering with or disrupting normal system functionality. However, in our goodwill experiments, we observed that our approach does not appear prone to false positives, as it considers such operations indicative of injection only under specific operating conditions characterized by spatial and temporal congruence.

The statistics reported in our study should be interpreted as a lower bound on the prevalence of process injection in the wild. A recurring challenge in malware research is the absence of an ideal analysis environment that is fully transparent and capable of accounting for environmental dependencies, such as trigger-based behavior (e.g., user interaction) or targeted execution (e.g., reliance on specific programs or files expected on the victim system). Our comparative findings between the BluePill and Detours-based implementations show that incorporating off-the-shelf anti-evasion measures can expose significantly more injection activity. An additional advantage of our analysis method is its simplicity, which makes it amenable to straightforward integration into other monitoring technologies.

Our findings indicate higher observed injection activity compared to prior studies. By moving away from strict patterns and implementation-specific rules, we were able to capture a broader range of technique variants. In the case of Process Hollowing—the most frequent technique in our observations—nearly half of the occurrences correspond to the *no-unmap* variant, which prior work did not account for. This improvement addresses a key limitation of the methodology employed by Starink et al. [135]. Furthermore, our work provides insights into the evolution of techniques and the strategic choices of threat actors that were not previously documented.

Finally, our methodology may contribute to improving malware detection and analysis systems. Our evaluation shows that, despite relying on resource-intensive techniques to detect and dump injected payloads, popular sandboxes missed approximately 20% of the injection cases detected by our lightweight approach.

We make available our code, dataset, and experimental results at: <https://github.com/Sap4Sec/process-injection/>.

Chapter 5

Beyond Single Traces: Multi-Trace Semantic Modeling of Malware Behavior

This chapter presents the second main contribution of this thesis: a structured framework that integrates multi-trace malware analysis in the presence of execution variability with controlled exploration.

Traditional dynamic analysis relies on single-execution reasoning, implicitly assuming that one runtime trace is sufficiently representative of a program’s behavior. Modern malware challenges this assumption through environmental adaptation, conditional execution, and behavioral diversity across runs. We argue that robust detection therefore requires program-level reasoning across multiple executions rather than reliance on isolated traces.

We first analyze how execution variability affects detection reliability and statistical stability. We then present an augmented API call graph that consolidates behavioral evidence across executions into a unified structural representation. Finally, we describe a structure-aware exploration mechanism that regulates trace expansion under bounded resources.

We evaluate the proposed approach through an empirical study addressing the following research questions: **(RQ1)** How does aggregating behavioral evidence across multiple program executions influence detection reliability and statistical stability? **(RQ2)** How does structural representation affect detection robustness and generalization across behavioral variants? **(RQ3)** Can structure-aware exploration reduce redundant behavioral testing while preserving or enhancing detection effectiveness? **(RQ4)** What statistical phenomena emerge under large-scale behavioral exploration, and how do they constrain scalable malware detection systems?

The findings establish foundational statistical and structural properties of multi-trace behavioral detection and should not be interpreted as a finalized production-scale system benchmark. Comprehensive large-scale validation and full system integration remain ongoing work and will be reported in a dedicated publication.

The research presented in this chapter is conducted in collaboration with Texas A&M University and co-supervised by Dr. Marcus Botacin.

5.1 Introduction

Behavioral variability has become one of the most effective—and least visible—mechanisms in modern malware design. Contemporary malicious software rarely follows a single deterministic execution path. Instead, adversaries embed anti-analysis logic and contextual dependencies that cause programs to manifest substantially different behaviors across executions [25, 97]. This variability enables detection evasion: a sample may expose benign-looking activity under analysis conditions while activating malicious payloads under different circumstances—even though both behaviors are encoded within the same underlying logic.

To cope with the thousands of new samples that appear daily, security vendors and analysts rely heavily on automated dynamic analysis sandboxes. These systems execute suspicious binaries in controlled environments and infer maliciousness from observed runtime behavior [62]. In operational settings, strict throughput requirements constrain analysis depth. Most large-scale pipelines analyze each sample only once; reports therefore reflect a single interaction observed under a specific environment at a specific point in time [106]. This single-execution paradigm rests on a strong and often implicit assumption: that one trace is sufficiently representative of the program’s behavioral space.

Modern malware directly challenges this assumption. Samples frequently exhibit “split personalities” depending on subtle contextual cues [24, 25], and may display natural behavioral variation even without explicit stimulation [34]. Some runs may reveal overtly malicious behavior, others appear benign, and many lie in a gray zone that is difficult to interpret. A single execution yields, at best, a partial view of the program’s true capabilities and, at worst, a misleading one.

In principle, one might attempt to resolve this limitation through exhaustive path exploration. In practice, this is infeasible due to path explosion [136] and the unpredictability of real-world systems [34]. Analysts must therefore assess malicious potential under intrinsic uncertainty, based on a limited and often unrepresentative subset of possible executions.

Fuzzing-inspired techniques have been proposed to expose hidden behaviors by executing samples under controlled variations [34, 38, 72]. While such strategies increase behavioral coverage, they fundamentally transform the detection problem. Instead of reasoning over a single trace, defenders must reason over a distribution of traces, each representing a partial observation of a latent behavioral space.

Crucially, this shift is not merely quantitative. Moving from “one execution per sample” to “many executions per sample” changes the representational object of analysis. The unit of reasoning is no longer a single trace, but the structural invariants and divergences that emerge across traces. Capturing these cross-execution properties requires abstractions that integrate behavioral evidence while filtering incidental runtime variability. In this sense, multi-trace analysis is inherently a semantic problem: it demands models that represent behavior at the level of structural relationships across executions rather than isolated event sequences.

Despite this representational shift, the multi-trace setting remains underexplored in machine-learning-based malware detection. A deeper issue lies in the fragmentation of the research landscape. On one side, substantial effort has been devoted to improving data acquisition—designing analysis environments [60, 85, 133]

and applying fuzzing strategies to expose evasive behavior [34, 38, 72]. On the other side, a vast literature investigates sophisticated machine-learning models for malware classification [28, 68], often assuming that the available traces are representative and semantically meaningful. These two dimensions—trace generation and classification—are typically studied in isolation.

In most existing pipelines, trace generation is treated as a fixed preprocessing stage, while classification models are developed independently of how behavioral evidence is collected. As a result, execution traces are rarely optimized for downstream learning objectives, and classifiers lack mechanisms to influence or refine the acquisition process. Behavioral variability is therefore addressed either through improved exploration without structured reasoning, or through advanced models to traces that may remain unrepresentative.

This separation exposes a fundamental gap. Robust malware classification cannot be achieved by simply collecting more executions. Repeated testing improves behavioral coverage, but also alters the statistical regime under which detection operates. Aggregating evidence across traces increases sensitivity, yet can inflate false positive rates due to repeated hypothesis testing. Multi-trace analysis introduces a reliability–stability trade-off that must be understood and controlled. The challenge is not merely to observe more behavior, but to integrate evidence across executions in a statistically controlled and structurally meaningful manner.

Our approach. We argue that the limitations of single-run analysis are not merely a matter of insufficient coverage, but fundamentally a representational and integration problem. Robust malware classification requires transitioning from isolated single-trace reasoning to structured multi-trace modeling, in which the generation and interpretation of behavioral evidence are considered jointly. By explicitly coupling exploration with representation and classification, we address the longstanding separation between trace acquisition and learning that characterizes existing detection pipelines. Simply collecting more executions is not enough: the benefit depends on how behavioral evidence is explored, abstracted, and aligned with downstream learning objectives, since repeated testing alters the statistical regime and can inflate false positive rates. The central question is therefore not whether additional executions improve detection, but how aggregation, representation, and exploration interact under statistical constraints.

To this end, we model the behavior of each sample through an *augmented API call graph* that integrates information from multiple executions into a unified structural abstraction. Rather than treating traces as independent observations, this representation merges behavioral evidence across runs while normalizing incidental variability—such as system-dependent constants, concrete memory addresses, or user-specific artifacts—that does not reflect semantic intent. The resulting graph redefines the behavioral object of analysis from an individual trace to a cross-execution representation that encodes both regularities and divergences. In this sense, it serves as a conservative approximation of the explored behavioral space.

To study this problem, we introduce a multi-trace framework that connects a sequential baseline detector operating on fixed-length API-call windows, the augmented API call graph as a structured integration layer, and a graph-aware

exploration mechanism that expands the representation selectively using classifier feedback and threshold-based stopping. This closed-loop design ensures that evidence acquisition is guided by representational structure and statistical considerations rather than by unbounded coverage.

We conduct a controlled empirical study on 958 samples, evenly split between real malware and benign software, comprising approximately 34,800 traces under multi-execution settings. Our analysis reveals several fundamental properties of multi-trace malware detection. Aggregating behavioral evidence improves sensitivity but also alters the statistical regime of detection under repeated testing, amplifying the propagation of local classification errors across executions. Structural (graph-based) representation introduces inductive bias toward connectivity and dependency, providing a principled mechanism for invariance across execution variants. Finally, structure-aware exploration enables early detection saturation and reduces redundant expansion, highlighting that scalability requires controlled evidence acquisition rather than monotonically increasing execution counts.

Taken together, these results indicate that multi-trace analysis is necessary but not sufficient. Scalable malware detection demands principled aggregation rules, structurally grounded representations, and exploration policies that operate under explicit statistical constraints. Detection quality does not scale monotonically with execution count; instead, it depends critically on how behavioral evidence is selected, structured, and integrated.

Accordingly, the evaluation in this chapter focuses on controlled characterization of the statistical and structural properties induced by multi-trace behavioral analysis rather than exhaustive corpus-scale benchmarking, since repeated executions and iterative exploration substantially increase the computational cost per analyzed sample.

5.1.1 Contributions

This work advances dynamic malware analysis along three complementary dimensions:

- We introduce an augmented API call graph representation that consolidates behavioral evidence from multiple traces into a unified structural abstraction. By normalizing incidental variability and explicitly encoding divergence across runs, this representation enables program-level reasoning under execution variability and captures structural invariants across executions.
- We provide an empirical analysis of multi-trace aggregation, characterizing how detection sensitivity and false positive rates evolve under repeated behavioral exploration. We formalize the multiple-testing effects inherent in multi-execution analysis and expose the resulting trade-offs.
- We propose a graph-aware exploration strategy that leverages structural divergence and classifier feedback to prioritize informative behavioral variants and enable early stopping. This mechanism reduces redundant exploration and mitigates false positive accumulation under repeated testing.

5.2 Problem Definition and Threat Model

In this section, we formalize the detection problem addressed in this chapter and define the threat model under which our approach operates. We first characterize the sources and implications of behavioral variability and partial observability in dynamic analysis. We then introduce a formal model of execution traces that clarifies the fundamental limitations of single-trace reasoning for reliable malware classification.

5.2.1 Behavioral Variability and Partial Observability

We consider a realistic threat model in which malware is executed inside a dynamic analysis environment that monitors a subset of Windows API calls commonly observed by antivirus products and sandbox systems. We do not assume complete observability; instead, we align monitoring capabilities with those achievable in practical deployment settings.

To mitigate common evasion techniques and increase environmental transparency, we assume the presence of anti-evasion countermeasures that reduce the visibility of the analysis environment, for example by masking virtualization artifacts or modifying observable system properties [60]. Despite these efforts, the defender’s view remains inherently incomplete: certain low-level memory manipulations, native instructions, and unmonitored APIs may remain invisible to the analysis system [97].

Within these observational boundaries, malware exhibits substantial behavioral variability across executions. Many modern malware families adapt their behavior to the execution context, incorporating checks for virtual machines, debuggers, or specific system configurations before revealing malicious functionality (see Section 2.2.1). Others rely on time-based triggers, probabilistic branching, or external conditions such as network availability, leading to distinct control-flow paths across runs. Even benign applications may display variation due to changes in system state or user environment.

An execution trace should therefore not be interpreted as a fixed behavioral fingerprint, but rather as one possible manifestation drawn from a broader behavioral space. From the defender’s perspective, this induces a condition of partial observability: each trace provides useful evidence, yet none offers a complete and reliable view of the program’s capabilities. Classification decisions may depend heavily on which execution instance is observed.

The core challenge is thus not merely to classify a single trace, but to reason about program-level behavior under limited and potentially misleading observations. This motivates representations and aggregation strategies that integrate evidence across executions while abstracting away incidental variability, forming the foundation for structured multi-trace modeling.

5.2.2 Formal Model of Execution Traces

To reason more precisely about this setting, we model a malware sample M as generating a (potentially very large) set of feasible execution traces

$$T(M) = \{t_1, t_2, \dots, t_n\}$$

under varying execution conditions, where each trace t_i is defined as a temporally ordered sequence of monitored API invocations with arguments and return values representing the observable interaction between the program and the operating system.

We define the ideal classification objective at the sample level as:

$$f(M) = \begin{cases} \text{malicious} & \text{if } \exists t_i \in T(M) \text{ revealing malicious intent,} \\ \text{benign} & \text{otherwise.} \end{cases}$$

This formulation captures the ground-truth objective: a program is malicious if it is capable of producing at least one execution trace that reveals harmful behavior. In practice, however, $T(M)$ is generally not directly accessible. Instead, the defender observes only a finite subset

$$\hat{T}(M) \subset T(M),$$

obtained through a limited number of executions under specific environmental configurations.

The asymmetry in this definition reflects operational nature of malware detection. A single revealing trace is sufficient to justify malicious classification. In contrast, benign classification implicitly requires confidence that no harmful trace exists within $T(M)$, a requirement that is difficult to satisfy under partial observation.

For modern malware, $|T(M)|$ is often large due to contextual branching and environmental dependence. Single-trace classifiers therefore capture only fragments of behavioral capability. Machine learning models typically treat traces as independent data points, even when multiple traces originate from the same sample M , obscuring the fact that traces are partial observations of the same underlying program.

This mismatch can lead to classifier instability: a sample’s training trace t_{train} may appear benign, while a different execution t_{test} reveals malicious behavior. Conversely, benign programs may sporadically exhibit anomalous traces that resemble malicious activity. As the number of observed traces increases, aggregation strategies must therefore balance improved sensitivity against the risk of statistical instability.

We thus explicitly recognize M , rather than individual traces, as the true classification unit, with traces serving as incomplete and correlated observations of an underlying behavioral distribution. The remainder of this chapter investigates how representation, aggregation, and exploration strategies can approximate $T(M)$ more faithfully while remaining statistically stable and computationally tractable.

5.3 Methodology

To address the limitations of single-trace dynamic analysis, we formulate malware detection as a structured multi-trace inference problem. Rather than assuming that a single execution is representative of a sample, we treat each malware instance M as a generator of multiple potential behaviors and aim to approximate its latent behavioral space $T(M)$ under realistic execution constraints.

5.3.1 Overview

Our methodology is grounded in two central intuitions. First, malware detection is inherently a program-level decision problem, whereas dynamic analysis produces trace-level observations. A single execution reveals only a partial view of $T(M)$; detection reliability must therefore be analyzed at the level of aggregated behavioral evidence rather than isolated trace-level predictions.

Second, multiple executions of the same program should not be treated as independent observations. They share structural components, diverge at specific decision points, and exhibit correlated behavioral patterns. Treating traces as unrelated sequences ignores this relational structure and may introduce redundancy, distort statistical estimates, and increase exposure to false-positive rates under repeated testing.

Accordingly, our approach consists of three tightly coupled components:

1. **Multi-trace collection.** Each sample is executed multiple times under controlled environmental variation and targeted API-level perturbation. This phase transforms the traditional single-trace perspective into a sample-centric view in which each trace represents a partial observation of the underlying behavioral space $T(M)$. The objective is not exhaustive coverage, but systematic exposure of behavioral variability within practical resource limits.
2. **Augmented structural representation.** Collected traces are merged into an augmented API call graph that encodes both shared and divergent structural regions across executions. This representation consolidates behavioral evidence into a unified abstraction while normalizing incidental variability such as memory addresses and environment-dependent artifacts. By preserving connectivity and explicit divergence points, the graph enables reasoning at the program level rather than at the level of isolated traces.
3. **Structure-aware exploration.** Rather than maximizing raw path coverage, exploration proceeds selectively over the augmented graph. Directed paths extracted from the graph are evaluated using a sequential malware detector with a predefined malicious-window threshold. When sufficient malicious evidence is observed, exploration terminates. When paths are classified as benign, a flip classifier identifies structurally expressive divergence regions. Traces consistent with these regions are then incorporated into the graph. This process prioritizes structurally informative variants and enables threshold-based early stopping under bounded execution budgets.

At a high level, the multi-trace collection phase provides empirical evidence about $T(M)$, the augmented graph consolidates these observations into a structured abstraction, and selective expansion governs how additional evidence is incorporated.

This design reframes dynamic malware analysis as a controlled sampling problem over a partially observable behavioral space. Rather than assuming that increased execution coverage automatically yields better detection, we explicitly study how aggregation rules, structural representation, and exploration policy interact to determine reliability, robustness, and statistical stability.

Because exhaustive enumeration of $T(M)$ is infeasible, our objective is threefold: (i) collect representative behavioral evidence under controlled variation, (ii) consolidate this evidence into a structural abstraction that captures shared and divergent behaviors, and (iii) regulate trace expansion through threshold-based stopping and structural prioritization to limit redundant evaluation and error accumulation.

5.3.2 Multi-Trace Collection

The first phase of our methodology aims to approximate the observable portion of the behavioral space $T(M)$ of a malware sample M . Under realistic monitoring constraints, this objective is pursued through systematic collection of multiple execution traces under controlled variation. As formalized earlier, each concrete execution corresponds to a single element of $T(M)$, while the program may exhibit substantially different behaviors depending on environmental context and internal decision logic. Because exhaustive exploration of $T(M)$ is computationally infeasible [136], our goal is not completeness but principled sampling. By observing multiple executions under structured perturbation, we construct a broader and more faithful empirical view of the sample’s latent behavioral capability.

Behavioral diversity arises from the inherently context-sensitive nature of modern malware. Execution paths may depend on system configuration, privilege level, network connectivity, resource availability, or values returned by system APIs. Malware frequently incorporates time-dependent logic, randomized branching, or explicit anti-analysis checks that suppress functionality in monitored environments [67, 100]. Consequently, repeated executions of the same binary may yield traces that differ both structurally and semantically, sometimes ranging from clearly malicious to apparently benign. Observing multiple executions therefore increases the probability of exposing dormant or conditionally activated functionality.

Each execution is monitored at the level of Windows API interactions commonly analyzed by sandbox and security systems, ensuring alignment with practical deployment settings. The resulting trace is a temporally ordered sequence of API invocations enriched with contextual information, including timestamp, process identity, module identity, function name, arguments, return value, and image-relative instruction offset to enable normalization across runs. This representation captures the program’s externally observable interaction with the operating system, providing a structured behavioral snapshot of a concrete execution instance. Nevertheless, each trace remains a projection conditioned on a specific runtime context and cannot, by itself, characterize the full extent of the program’s behavior.

Effective multi-trace collection requires inducing meaningful behavioral variation rather than merely repeating identical executions. Runs are therefore conducted under systematically varied environmental conditions designed to perturb decision points dependent on environmental feedback. These variations may affect system state, resource availability, configuration parameters, or selected API return values. The objective is not arbitrary mutation, but targeted stimulation of control-flow decisions that expand coverage of distinct execution paths while preserving semantic plausibility. By altering environmental signals across runs, we increase the likelihood of traversing divergent control-flow regions and activating behaviors that would otherwise remain dormant.

In particular, following prior work on fuzzing for multi-path malware tracing [34], we apply controlled return-value fuzzing to monitored APIs. By modifying return values within semantically meaningful bounds, we influence branch outcomes and expose alternative behavioral paths. This mechanism serves a dual purpose: it may defeat anti-analysis logic that suppresses malicious activity in instrumented environments, and it may reveal operational routines, fallback mechanisms, or latent capabilities not exercised during standard execution.

To further reduce adversarial sampling bias, we incorporate environmental transparency mechanisms inspired by state-of-the-art anti-evasion research [60]. Without such precautions, collected traces would systematically under-approximate $T(M)$ due to adversarial suppression of functionality. Mitigating this bias increases the likelihood that observed variability reflects genuine behavioral diversity rather than analysis-induced artifacts.

Although systematic variation broadens observable behavior, not all differences across traces correspond to semantically distinct capabilities. Some divergences arise from incidental environmental artifacts, such as memory layout differences, user-specific paths, or minor numeric fluctuations [3, 80, 97]. These variations inflate apparent diversity without contributing meaningful behavioral information. The effectiveness of multi-trace collection therefore cannot be measured solely by the number of executions performed, but by the extent to which semantically distinct execution paths are revealed. The collection process thus balances diversity and consistency, seeking to maximize informative behavioral variation while limiting redundant or superficial differences. These considerations motivate the structural alignment and semantic abstraction mechanisms introduced in the next subsection.

5.3.3 Augmented Structural Representation via API Call Graphs

The collection of multiple execution traces provides a broader empirical approximation of the behavioral space $T(M)$ of a malware sample M . However, traces remain sequential observations tied to specific runtime contexts. Treating them as independent sequences fails to capture structural relationships across executions and does not fully leverage the diversity gained through multi-trace sampling. As discussed in Section 5.2, single-trace reasoning implicitly assumes representativeness—an assumption that does not hold under behavioral variability. To overcome this limitation, we introduce the *augmented API call graph*, a structured representation that integrates multiple traces of the same sample into a unified behavioral abstraction.

Rather than viewing each execution as an isolated instance, this representation treats the malware sample as a generator of possible behaviors. The augmented graph serves as a compact structural model, encoding not only the actions observed in individual runs but also the relationships among alternative execution paths across runs. In doing so, it shifts the analytical unit from trace-level sequences to a sample-level control-flow abstraction.

As illustrated in Figure 5.1, two traces collected under different runtime conditions may share an initial segment, diverge into distinct behavioral branches, and later reconverge. By aligning semantically identical call sites and merging overlapping segments, the augmented graph integrates these executions into a unified structure that preserves both shared and alternative behaviors.

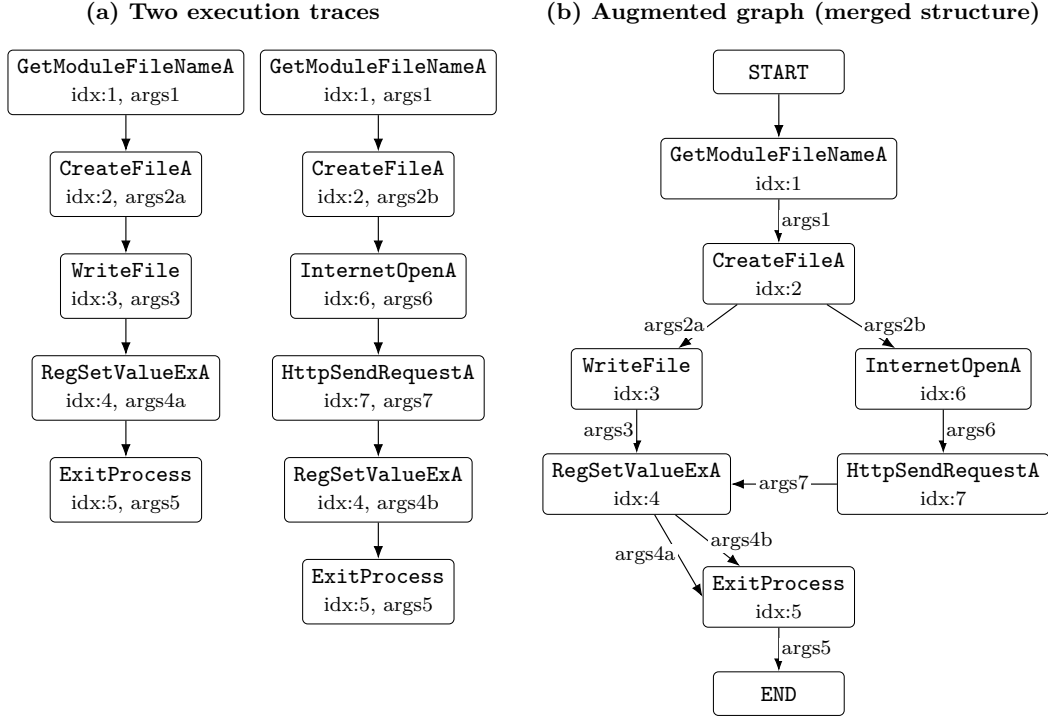


Figure 5.1. Augmented API call graph construction for a Windows malware sample.

- (a) Two execution traces collected under different runtime conditions share a common prefix, diverge into file/registry versus network behavior, and reconverge at termination.
- (b) After alignment by normalized call-site identifiers, the merged graph encodes the union of observed transitions; structural recombination across branches may imply virtual paths not explicitly observed in a single execution.

From sequential traces to structural representation. For each sample, the collected traces are transformed into a directed graph in which nodes correspond to monitored API invocations and edges represent observed control-flow transitions. Edges are enriched with argument and return-value information, allowing the graph to encode both structural dependencies and contextual semantics.

This transformation replaces purely linear temporal ordering with an explicit relational structure. While a single trace captures one realized execution path, the graph encodes multiple paths simultaneously, preserving shared prefixes and diverging suffixes across runs. The resulting structure makes explicit where executions converge, where they diverge, and how alternative behaviors are related.

Because repeated segments may arise both within and across traces, loop handling becomes a design consideration. We consider two construction strategies. In the first, loops observed in traces are preserved, maintaining cyclic structure. In the second, loops are unrolled by duplicating nodes along repeated paths, producing an acyclic representation without back edges. Both variants encode the same observed execution segments but introduce different structural biases: preserving loops emphasizes compactness and structural regularity, whereas unrolling emphasizes execution ordering and can simplify certain downstream analyses.

Trace alignment and semantic abstraction. Merging multiple executions into a single graph requires careful alignment to distinguish meaningful behavioral divergence from incidental runtime variability. Traces are aligned using normalized instruction identifiers computed at runtime, enabling consistent mapping of semantically identical call sites across executions. This alignment ensures that structural differences correspond to genuine behavioral variation rather than memory-layout artifacts [80].

Beyond alignment, we apply normalization and abstraction steps to eliminate variability that does not reflect distinct behavioral capabilities [3, 97]. Pointer-like arguments are abstracted to prevent address-level fragmentation. Environment-specific identifiers, such as file paths or user-dependent resources, are standardized while preserving meaningful distinctions. Numeric arguments subject to benign fluctuation are grouped into bounded intervals to prevent spurious structural divergence.

Semantically equivalent transitions are further consolidated to avoid combinatorial growth while preserving meaningful alternatives. The objective is not simplification for its own sake, but preservation of capability-level distinctions while filtering environmental noise. Without these steps, the augmented graph would primarily encode incidental runtime variability rather than stable behavioral structure.

The resulting graph integrates all observed executions of a sample into a single coherent representation. It is neither a simple concatenation of traces nor an averaged summary; rather, it represents the union of observed control-flow segments while preserving their relational organization. As additional traces are incorporated, the graph incrementally refines the observable approximation of $T(M)$, making explicit which behaviors are consistently exercised and which depend on specific environmental conditions.

Emergence of virtual paths. A notable property of the augmented representation is the emergence of *virtual paths*. These are sequences of API calls that do not explicitly appear in any single execution trace but arise through structural recombination of partially overlapping paths across runs. When traces share an initial segment, diverge, and later reconverge, the merged graph implicitly encodes alternative combinations of these segments.

Virtual paths do not imply guaranteed realizability. Rather, they represent behaviorally plausible sequences that are structurally consistent with the observed executions. In the context of $T(M)$, they serve as structured hypotheses about potentially reachable behavior implied by partial observation. This property is particularly valuable in adversarial settings, where malware may conceal functionality behind conditional checks that are only partially exercised during analysis.

Methodological implications for classification. The augmented API call graph does more than enrich trace representation; it redefines the unit of analysis for detection. Instead of treating executions as independent samples, the graph consolidates them into a single structural hypothesis about the behavioral capability of the program. Consequently, classification operates over an integrated approximation of $T(M)$ rather than isolated execution instances.

By explicitly encoding variability across executions, the augmented graph enables stable sample-level reasoning. Decisions are informed by structurally coherent behavioral regions rather than transient trace manifestations, thereby reducing the instability inherent in single-trace learning while preserving sensitivity to diverse execution variants.

5.3.4 Structure-Aware Exploration

Multi-trace structural modeling enables program-level reasoning, but it implicitly assumes the availability of multiple executions. In operational settings, analysts operate under constrained execution budgets: each additional run incurs computational cost and may amplify false-positive accumulation under repeated testing. The challenge is therefore not only how to collect multiple traces, but how to regulate trace expansion so that additional executions contribute informative structural evidence rather than redundant variability.

Blind re-execution is inefficient, as repeated runs frequently yield nearly identical traces. Conversely, purely random fuzzing increases variability without prioritizing structurally informative regions [34]. Rather than maximizing raw path coverage, our approach expands behavioral evidence selectively over the augmented graph constructed in the previous stage.

Exploration proceeds iteratively. Directed paths extracted from the augmented graph are segmented and evaluated using the sequential detector. If the number of windows classified as malicious exceeds a predefined threshold, the sample is labeled malicious and exploration terminates. This threshold-based stopping rule prevents unnecessary structural expansion once sufficient evidence has been accumulated, thereby limiting additional executions beyond the decision boundary.

If a path is classified as benign under the sequential detector, a secondary *flip classifier* is applied. The flip model operates on graph-derived pseudo-traces and identifies structurally expressive divergence regions—i.e., regions that differ across executions but are not associated with malicious predictions. For such regions, new executions are conducted under conditions likely to realize the corresponding structural alternatives, and the resulting traces are incorporated into the augmented graph.

Each newly incorporated trace refines the structural abstraction. The process then repeats: newly exposed paths are evaluated, malicious thresholds are checked, and structurally expressive benign regions are selectively expanded. Exploration thus resembles a best-first branch expansion guided by divergence ranking and regulated by threshold-based stopping.

This mechanism does not attempt exhaustive coverage of $T(M)$, nor does it optimize a global uncertainty objective. Instead, it controls the order in which behavioral variants are incorporated, prioritizing structurally divergent regions while enabling early termination once sufficient evidence is observed. In this sense, structure-aware exploration regulates expansion depth under bounded resources rather than maximizing execution count.

5.3.5 Implementation

This section describes the implementation of the proposed multi-trace framework, organized around three core components: (i) multi-trace collection via API-level instrumentation and controlled perturbation, (ii) structural consolidation through the augmented API call graph, and (iii) learning-based analysis using sequential and structure-derived classifiers.

API tracing module

Realizing the multi-trace collection strategy requires runtime interposition on program-environment interactions. Our approach is compatible with dynamic analysis systems that support API interception. For flexibility and practical deployment, we implement the tracer using the Intel Pin dynamic binary instrumentation (DBI) framework [99], which enables fine-grained instrumentation of user-space execution.

As discussed in Section 2.1.2, DBI-based techniques provide an effective trade-off between transparency and analytical depth. They enable detailed behavioral inspection while facilitating integration with anti-evasion countermeasures, and they are widely adopted in malware analysis research.

Malware primarily interacts with the operating system through Windows API calls. These interactions are observable via interception mechanisms commonly employed by sandbox and security systems. Our tracing module instruments a selected subset of Windows APIs to capture semantically relevant behavioral events. The monitored API set aligns with functions commonly intercepted by sandbox frameworks such as Cuckoo [127], reflecting established dynamic analysis practice. We further extend this set with APIs associated with evasion techniques identified in prior work [38].

In total, we instrument 308 distinct API functions (including A/W variants). These functions are selected based on their frequent association with malicious capability execution and environmental awareness. They span core operating system interfaces, including filesystem operations, process and thread management, registry modification, service control, networking, cryptography, system fingerprinting, synchronization primitives, and timing-based evasion. The selected set balances broad behavioral coverage with manageable instrumentation overhead.

For each intercepted API invocation, the handler records the timestamp, process and thread identifiers, function name and module, argument values, return value, and an image-relative instruction offset. The image-relative offset ensures consistent identification of call sites across executions despite address space layout randomization or loader-induced relocation. Table 5.1 provides representative examples of monitored APIs across behavioral categories, illustrating the diversity of capability-relevant functions included in the tracing configuration.

To support controlled behavioral perturbation, we implement mutation operators directly at the API interception layer. By default, return values remain unmodified to reflect natural execution. When perturbation is enabled, the handler substitutes semantically plausible failure codes or alternative values consistent with realistic system responses. For example, a call to `GetUserNameA` may either return the actual username or be forced to return a failure code. Such constrained perturbations

Table 5.1. Representative examples of Windows APIs instrumented by the tracing module, grouped by behavioral category.

Group	API Name	Description
Cert Store	CertControlStore	Allows an application to be notified when there is a difference between the contents of a cached store in use and the contents of that store as it is persisted to storage
	CertOpenStore	Opens a certificate store by using a specified store provider type
	CertCreateCertificateContext	Creates a certificate context from an encoded certificate
Crypto	CryptEncrypt	Encrypts data
	CryptDecrypt	Decrypts data previously encrypted
	CryptGenKey	Generates a random cryptographic session key or a public/private key pair
Filesystem	CreateDirectoryA	Creates a new directory
	MapViewOfFile	Maps a view of a file mapping into the address space of a calling process
	WriteFile	Writes data to the specified file or input/output (I/O) device
Misc	ShellExecuteExA	Performs an operation on a specified file
	FindResourceA	Determines the location of a resource with the specified type and name in the specified module
	ExitWindowsEx	Logs off the interactive user, shuts down the system, or shuts down and restarts the system
Network	InternetOpenA	Initializes an application's use of the WinINet functions
	HttpSendRequestA	Sends the specified request to the HTTP server
	WinHttpOpen	Initializes, for an application, the use of WinHTTP functions and returns a WinHTTP-session handle
Process	VirtualAllocEx	Reserves, commits, or changes the state of a region of memory within the virtual address space of a specified process
	WriteProcessMemory	Writes data to an area of memory in a specified process
	CreateProcessA	Creates a new process and its primary thread
Registry	RegCreateKeyA	Creates the specified registry key
	RegQueryInfoKeyA	Retrieves information about the specified registry key
	RegSetValueExA	Sets the data and type of a specified value under a registry key
Services	ControlService	Sends a control code to a service
	EnumServicesStatusA	Enumerates services in the specified service control manager database
	OpenSCManagerA	Establishes a connection to the service control manager on the specified computer and opens the specified service control manager database
Synchronization	CreateMutexA	Creates or opens a named or unnamed mutex object
	CreateSemaphoreA	Creates or opens a named or unnamed semaphore object
	EnterCriticalSection	Waits for ownership of the specified critical section object
System	GetUserNameA	Retrieves the name of the user associated with the current thread
	GetTimeZoneInformation	Retrieves the current time zone settings
	IsProcessorFeaturePresent	Determines whether the specified processor feature is supported by the current computer

stimulate conditional branches dependent on environmental feedback while preserving execution plausibility and avoiding unrealistic system states.

To ensure comparability across multiple runs, API invocations are normalized using their image-relative instruction offsets. Absolute memory addresses vary across executions due to randomization and loader effects; offset normalization guarantees consistent alignment of semantically identical call sites across traces.

Malware frequently incorporates mechanisms to detect analysis environments and suppress malicious behavior. To mitigate this risk, we integrate established anti-evasion techniques [60] that reduce observable artifacts of instrumentation and virtualization. These countermeasures limit adversarial suppression of functionality and increase the likelihood that observed trace variability reflects genuine program behavior rather than artifacts of the analysis environment.

Augmented API Call Graph

The augmented API call graph is implemented as a directed multigraph that integrates multiple execution traces into a unified structural representation. The abstraction consists of vertices representing API invocation sites and edges representing observed control-flow transitions enriched with contextual information.

Each vertex corresponds to a unique API call site and is identified by three attributes: the API name, the module, and the normalized instruction offset. This design guarantees alignment of semantically identical call sites across executions and directly supports the trace-alignment strategy described in the methodology.

Edges encode directed transitions between successive API invocations within each trace. In addition to source and destination vertices, each edge stores contextual attributes extracted from traces, including normalized argument values, return values, and fuzzing annotations. The resulting graph therefore captures both structural dependencies and execution-specific semantic context.

During ingestion, each API call is mapped to a vertex and consecutive calls are connected by directed edges. Prior to insertion into the global graph, normalization procedures are applied to reduce environment-induced variability and prevent artificial structural fragmentation.

Pointer normalization replaces concrete memory addresses with symbolic placeholders (e.g., `ptr_<id>`), eliminating address-level fragmentation across executions. File paths and resource identifiers are standardized to remove user- and environment-specific artifacts (e.g., Windows version differences, user profiles, or directory layouts) while preserving semantically meaningful structure. For example, paths rooted in `C:\` retain the drive and top-level directory, whereas file names are replaced with deterministic placeholders scoped to source-destination pairs. Numeric attributes identified as mergeable are abstracted into bounded intervals during consolidation. This allows multiple concrete values to collapse into a single abstract edge when differences fall within semantically acceptable bounds.

After integrating multiple traces, the graph undergoes a consolidation phase that merges semantically equivalent transitions. Three complementary strategies are applied:

- **Structural merging:** redundant edges with identical attributes and destinations are removed.

- **Pointer-based merging:** transitions that differ only in normalized pointer identifiers are unified.
- **Numeric interval merging:** transitions that differ only in mergeable numeric attributes are abstracted into interval-based representations.

Merging is performed locally at the level of outgoing edges from each vertex. This preserves global behavioral distinctions while reducing redundant structural replication and limiting graph growth.

To manage multi-path phenomena—where multiple edges connect the same source and destination with distinct contextual annotations—the implementation includes a node-squashing mechanism. When such patterns are detected, synthetic intermediate vertices are introduced to encode abstracted behavioral alternatives between the original source and destination. This process converts parallel multi-edges into explicit branching structures, preventing combinatorial edge proliferation while preserving semantically distinct behavioral options.

Finally, the graph maintains explicit **START** and **END** sentinel nodes to embed all traces within a common entry and exit structure. This guarantees that each execution contributes a connected subgraph and simplifies traversal, merging, and structural analysis operations across the unified representation.

Sequential and Structural-Derived Classifiers

To study the effect of behavioral representation and aggregation on malware detection, we implement two complementary learning models: (i) a sequential trace-based classifier and (ii) a structure-derived flip classifier operating on paths extracted from the augmented multi-trace graph.

Both models share a common downstream learning algorithm based on Random Forests. Using a consistent classifier architecture isolates the effect of representation and data preparation, ensuring that observed performance differences arise from behavioral abstraction rather than variations in model complexity. Detailed hyperparameter configurations and dataset splits are provided in the evaluation section.

Random Forests are selected for their robustness in high-dimensional sparse feature spaces, their ability to capture nonlinear decision boundaries, and their stability under heterogeneous behavioral distributions—properties well suited to API-level behavioral data.

Sequential trace-based classifier. The sequential classifier operates directly on execution traces and serves as the primary baseline for malware detection. Each trace is segmented into fixed-length, non-overlapping windows of API calls, where each window captures a localized behavioral context. The window size is fixed across experiments; its empirical justification and sensitivity analysis are discussed in the evaluation section.

Each window is tokenized into a textual representation encoding API names together with selected contextual attributes (e.g., normalized arguments and return values). Because API calls vary in parameter structure, token sequences may differ in length across windows. This variability reflects genuine invocation-level differences and is preserved to avoid artificial truncation.

The tokenized representation is transformed into a sparse numerical feature vector using TF-IDF encoding. Classification is performed at the window level using a binary Random Forest model distinguishing malicious from benign behavior. Sample-level decisions are subsequently derived through aggregation of window-level predictions, as formalized in the evaluation protocol.

This model represents a standard sequential behavioral detector and provides a controlled baseline for studying the impact of multi-trace aggregation and structural abstraction.

Structure-derived flip classifier. The flip classifier extends the sequential baseline by operating on structural information derived from the augmented multi-trace graph. Rather than analyzing raw trace segments, we extract directed execution paths from the graph that satisfy a minimum structural length constraint. This filtering step removes trivial fragments and focuses the model on behaviorally meaningful regions of the graph.

Extracted paths are converted into pseudo-traces and segmented using a sliding window strategy to capture localized structural patterns. Feature extraction mirrors the sequential model via TF-IDF encoding, ensuring comparability at the feature level and isolating the effect of structural derivation from feature engineering differences.

To assign labels for flip training, we employ a capability-based semantic criterion derived from CAPA rule matching [101]. For each trace, CAPA rules are evaluated over the corresponding dynamic behavior report, producing a set of matched capability identifiers. Structural segments (i.e., extracted paths or windows) are labeled as `flip` when they are associated with traces exhibiting non-trivial capability activation, and as `not_flip` otherwise. CAPA is used here as a weak semantic signal to identify capability-relevant behavioral regions, not as a ground-truth oracle for malware labeling.

In this formulation, the flip label does not denote malware versus benign classification; instead, it indicates the presence of semantically expressive or behaviorally divergent regions within the structural representation. Such regions frequently correspond to decision points where alternative execution branches activate distinct behavioral routines.

This design separates structural divergence detection from primary malware classification. Whereas the sequential classifier predicts malicious versus benign behavior at the window level, the flip classifier identifies graph-derived regions that correspond to meaningful capability-level variation across executions. Such regions often correspond to decision points where alternative execution branches activate distinct behavioral routines.

Class imbalance is addressed through frequency-proportional class weighting. The precise thresholds used for path length constraints, capability activation, and trace selection are specified in the evaluation section.

Conceptually, the flip classifier serves as a structural probe over the augmented representation. It does not replace the primary detection model; rather, it highlights behaviorally expressive or divergent regions within the multi-trace graph. This decoupling clarifies the distinct roles of representation and decision-making: the sequential model performs malware detection, while the flip model characterizes structural variability within the observed behavioral space.

5.4 Experimental Evaluation

This section presents an empirical evaluation of the proposed multi-trace malware analysis framework. The objective extends beyond reporting classification accuracy to characterizing how detection reliability, robustness, and statistical stability evolve under execution variability and large-scale behavioral exploration.

We study how detection behaves when behavioral evidence is accumulated across multiple executions, how structural abstractions influence generalization across variants, and how exploration policies interact with statistical scaling effects.

Rather than benchmarking a wide range of models, we isolate four fundamental dimensions: (i) behavioral aggregation across executions, (ii) structural representation of execution behavior, (iii) structure-aware exploration strategies, and (iv) statistical phenomena emerging under large-scale fuzzing.

The evaluation is organized around the following research questions:

RQ1: How does aggregating behavioral evidence across multiple program executions influence detection reliability and statistical stability?

RQ2: How does structural representation affect detection robustness and generalization across behavioral variants?

RQ3: Can structure-aware exploration reduce redundant behavioral testing while preserving or enhancing detection effectiveness?

RQ4: What statistical phenomena emerge under large-scale behavioral exploration, and how do they constrain scalable malware detection systems?

5.4.1 Experimental Setup

The experiments are designed as controlled studies that isolate specific statistical and representational effects, validating the proposed analytical perspective and demonstrating the practical implications of structural abstraction under bounded execution settings.

Dataset

The dataset comprises approximately 34,800 dynamically collected execution traces derived from 958 samples, evenly split between real malware and benign software, under the instrumentation framework described in Section 5.3.2.

Malware samples were collected from two complementary sources: (i) a training set of unique Brazilian banking Windows desktop malware [35], and (ii) a testing set of samples exhibiting environment-sensitive behavior [38]. Together, these datasets provide curated ground truth for analysis of real-world malware. Benign software was collected from popular online repositories following established methodologies in software security research [36]. Each sample was executed multiple times under controlled environmental variation and return-value perturbation.

All experiments employ a random 90/10 train-test split at the sample level to prevent cross-trace leakage between training and testing. Multi-trace aggregation is performed exclusively within samples belonging to the same split.

To study the effect of behavioral richness and structural variability, we evaluate two dataset configurations:

- **Naive configuration:** all collected traces are included without structural or capability-based filtering.
- **Capability-filtered configuration:** only samples with at least five execution traces are retained, and only traces exhibiting a minimum level of semantic expressiveness are considered.

Semantic expressiveness is determined using CAPA rule matching over dynamic behavior reports. For each trace, matched CAPA capabilities are computed, and traces whose capability activations fall below a predefined threshold are excluded. This filtering enforces a minimum behavioral signal requirement and reduces shallow executions that do not meaningfully contribute to structural modeling.

The filtered configuration enables analysis of distributional shifts induced by capability-based selection and allows us to examine how aggregation behaves when low-information traces are removed.

5.4.2 Feature Extraction

Each execution trace is segmented into non-overlapping windows of fixed size $w = 44$ API calls. This window size was selected empirically as the smallest context length that consistently yielded a window-level false positive rate below 1% under the sequential baseline on the training data. It therefore represents a sufficient semantic context for stable local discrimination.

Each window is tokenized into a textual representation capturing API names together with normalized contextual attributes. Because API calls vary in parameter structure and argument count, token sequences differ in length across windows. This variability reflects intrinsic semantic diversity and is preserved rather than artificially truncated.

Tokenized windows are embedded using TF-IDF encoding to produce sparse feature vectors suitable for tree-based classifiers. This representation is shared across models to ensure comparability and isolate the effect of structural derivation rather than feature engineering differences.

5.4.3 Models

Sequential Baseline

The primary detection model is a Random Forest classifier operating at the window level. For each fixed-length window, the model independently predicts malicious versus benign behavior. Window-level predictions are aggregated hierarchically: first within each trace, and subsequently across traces belonging to the same sample to obtain a sample-level decision.

Default hyperparameters are used unless otherwise stated to avoid overfitting to specific experimental conditions. Hyperparameter optimization is not the focus of this study; rather, the objective is to isolate the effects of aggregation and representation.

Table 5.2. Linear detector performance across aggregation levels under naive and filtered configurations.

Config	Level	TPR	FPR	Precision
Naive	Window	0.418	0.004	0.991
	Trace	0.777	0.007	0.991
	Sample	0.818	0.253	0.764
Filtered	Window	0.439	0.006	0.987
	Trace	0.538	0.007	0.987
	Sample	0.716	0.236	0.752

Importantly, the sequential baseline does not incorporate structural information. Each trace is treated as an independent observation, and aggregation across executions is purely statistical, without structural coupling. This model therefore serves as a controlled baseline for studying the impact of multi-trace aggregation (RQ1) and statistical scaling effects (RQ4).

Structure-Derived Flip Classifier

The flip classifier operates on paths extracted from the augmented API call graph. For each sample, the consolidated graph representation is constructed. Directed paths satisfying a minimum structural length constraint are then extracted to focus on behaviorally meaningful regions. These paths are converted into pseudo-traces and segmented using a sliding window strategy to capture localized structural divergence.

Flip labels are assigned using CAPA-based capability activation with a predefined threshold. Structural segments associated with traces exhibiting non-trivial capability matches are labeled as `flip`, while others are labeled as `not_flip`. The flip classifier therefore does not predict malware versus benign behavior; instead, it identifies structurally expressive or divergent regions within the augmented representation.

Class imbalance is handled using frequency-proportional class weighting.

Conceptually, the flip model functions as a structural probe that highlights decision-relevant divergence points across executions. It is evaluated jointly with the sequential detector to quantify structure-aware exploration effects (RQ3), rather than as a standalone malware classifier.

5.4.4 Behavioral Aggregation and Detection Stability

We begin by quantifying how detection behavior evolves as evidence is aggregated hierarchically: from windows to traces, and from traces to samples. This analysis isolates the statistical effect of repeated behavioral testing under a purely sequential model, without incorporating structural constraints.

Table 5.2 reports performance metrics at three granularities. Window-level predictions are aggregated to traces using an OR-based rule (a trace is malicious if at least one window is malicious), and trace-level predictions are further aggregated to samples using the same rule across executions.

Window-level decisions. At the window level, false positive rates remain below 1% in both configurations. This operating point is deliberate rather than incidental:

the classifier was trained using a cost-sensitive objective that assigns a higher penalty to misclassifying benign (goodware) windows than malware windows. This reflects operational realities in endpoint security, where even small false positive rates become impractical at scale.

Under this design, overall accuracy is not an appropriate evaluation metric, as it assumes symmetric misclassification costs and can mask undesirable operating points. Instead, the model is calibrated toward high precision ($\approx 99\%$ at the window level) at the expense of recall. In other words, when the detector flags a window as malicious, it is almost always correct, but detection coverage is intentionally sacrificed to maintain strict false positive control.

This distinction is critical for realistic evaluation. Prior work has often reported near-perfect performance (e.g., $> 99\%$ accuracy) in sandbox-based settings without accounting for deployment conditions or the effects of distribution shift between sandbox and endpoint environments. Recent evidence demonstrates that such laboratory measurements can substantially overestimate real-world effectiveness [82]. Our enforced low-FPR operating point therefore prioritizes deployment realism over benchmark optimization.

While false positives remain tightly controlled, window-level true positive rates are moderate (0.418–0.439), indicating that a single short 44-API-call segment often lacks sufficient discriminative evidence. This motivates aggregation across windows and executions as a mechanism for recovering sensitivity without relaxing the false positive constraint.

Trace-level aggregation. Aggregating window-level predictions within a single execution substantially improves sensitivity in the naive configuration (TPR increases from 0.418 to 0.777). A trace is classified as malicious if at least one window is predicted as malicious. Detection therefore no longer requires every window to exhibit strong evidence; identifying a single discriminative segment suffices.

However, this asymmetry imposes a stricter robustness requirement on benign traces. For a benign execution to be classified correctly, every window must be labeled benign. A single erroneous window contaminates the entire trace. Thus, aggregation boosts malware detection while simultaneously increasing vulnerability to false positives.

This reveals a deeper difficulty: achieving low error rates at the window or single-trace level is relatively easy, but maintaining correctness across multiple windows and executions is substantially more demanding. Trace-level evaluation therefore better captures stability under environmental variation than isolated window-level metrics.

In the capability-filtered configuration, trace-level TPR decreases (0.538), reflecting the impact of increased behavioral diversity. Enforcing minimum behavioral richness exposes additional execution paths and increases variability across traces. While this diversity can reveal latent malicious behavior, it also amplifies false positives on benign executions. This demonstrates limited robustness to small behavioral perturbations and motivates further analysis under RQ2 (Section 5.4.5).

Sample-level aggregation. At the sample level, sensitivity increases only modestly relative to the trace level, whereas the false positive rate rises sharply (e.g.,

Table 5.3. Sample-level performance as a function of aggregated trace count (n) under two threshold calibrations: a conservative operating point (low per-trace FPR) and an aggressive operating point (higher per-trace FPR).

Operating Point	n	TPR	FPR
Conservative	1	0.495	0.006
	3	0.610	0.023
	5	0.662	0.031
	10	0.688	0.040
	max	0.696	0.128
Aggressive	1	0.813	0.111
	3	0.866	0.273
	5	0.868	0.367
	10	0.868	0.494
	max	0.868	0.767

0.253 in the naive setting). This behavior reflects the compounding asymmetry of OR-based aggregation: a single positive trace suffices to classify a sample as malicious. For malware samples, additional executions probabilistically reduce false negatives. For benign samples, however, the probability that at least one trace triggers a false positive increases with the number of executions evaluated. Aggregation therefore increases sensitivity while amplifying instability.

False positive growth under repeated execution. To isolate the effect of repeated execution, we evaluate sample-level performance as a function of the number of aggregated traces (n). Table 5.3 reports results under two threshold calibrations: a conservative operating point with low per-trace FPR, and an aggressive operating point with higher per-trace FPR.

Under the conservative operating point, increasing n improves TPR (0.495 $\rightarrow$ 0.688 by $n = 10$), demonstrating the benefit of multi-trace aggregation. However, FPR increases steadily even when per-trace error rates are low.

Under the aggressive operating point, TPR saturates rapidly while FPR grows sharply and eventually dominates the decision process. Beyond a certain number of executions, additional traces yield negligible sensitivity gains but substantially degrade specificity.

From a statistical perspective, let p denote the per-trace false positive probability under OR-based aggregation. If trace-level decisions were independent, the probability that a benign sample is misclassified after aggregating n traces is:

$$P(\text{FP}_n) = 1 - (1 - p)^n.$$

This function increases monotonically with n and converges to 1 as $n \rightarrow \infty$. Although strict independence does not hold due to trace correlation, empirical results exhibit the same qualitative trend: repeated behavioral testing increases the probability that at least one local error propagates to the sample level. Figure 5.2 visualizes this growth empirically.

More importantly, this phenomenon reveals that multi-trace malware detection differs qualitatively from single-execution analysis. In conventional classification

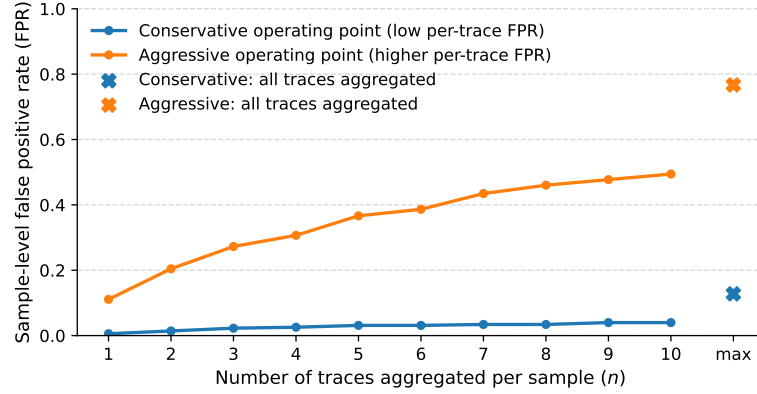


Figure 5.2. Growth of sample-level false positive rate as a function of the number of aggregated traces under conservative and aggressive threshold calibrations.

settings, additional observations are generally expected to improve confidence through evidence accumulation. Under repeated behavioral exploration, however, each additional execution simultaneously increases both the opportunity to observe latent malicious behavior and the opportunity for local classification errors to propagate through aggregation. Consequently, detection performance depends not only on classifier quality, but also on how behavioral evidence is accumulated as exploration depth increases.

RQ1. Multi-trace behavioral detection operates under a fundamentally different regime than single-execution analysis. Aggregating evidence across executions improves sensitivity, but repeated exploration also amplifies the propagation of local classification errors to the sample level. Detection stability is therefore constrained not only by classifier quality, but by the interaction between aggregation policy and exploration depth.

5.4.5 Structural Representation and Robustness

The previous analysis showed that naive multi-trace aggregation improves sensitivity but also introduces statistical instability under repeated behavioral testing. The second research question examines how behavioral representation influences robustness and generalization under execution variability.

In our framework, the sequential baseline encodes behavior as local API windows aggregated statistically, whereas the augmented representation introduces structural bias by consolidating multiple executions into a unified call-graph abstraction. Although a dedicated graph-based malware classifier is not evaluated here, comparing sequential aggregation and structure-derived modeling allows us to analyze how representational assumptions affect robustness.

Sequential encoding and locality assumptions. The sequential model encodes behavior locally by classifying fixed-length windows and aggregating predictions

temporally. This representation captures short-range syntactic regularities and relies on repeated local evidence to recover higher-level semantics.

Empirically, aggregation within a single trace substantially improves sensitivity in the unfiltered configuration (window-level TPR 0.418 $\rightarrow$ trace-level TPR 0.777 in Table 5.2). This suggests that malicious semantics are often distributed across multiple local contexts and become detectable when evidence is pooled over time.

However, robustness deteriorates under capability-based filtering. In the filtered configuration, trace-level TPR decreases from 0.777 to 0.538. Because filtering enforces minimum behavioral richness while increasing diversity and reducing redundancy, this degradation is unlikely to be fully explained by reduced signal strength alone. Instead, it suggests a shift in the behavioral distribution toward greater structural heterogeneity.

This sensitivity indicates that the sequential representation is partially dependent on surface-level regularities. When behavioral diversity increases and repetitive patterns become less dominant, detection performance degrades. This representational limitation provides a structural explanation for the instability and false-positive accumulation observed under multi-trace aggregation in RQ1 (Section 5.4.4).

Filtering as controlled distribution shift. The capability-filtered configuration acts as a controlled distribution shift. By retaining only samples with at least five traces and traces exhibiting non-trivial capability activation, the dataset emphasizes structural diversity across executions.

Under this shift, the sequential n -gram representation exhibits reduced sensitivity, suggesting limited invariance to execution variants. Local token-based encodings may struggle to generalize when malicious behavior is expressed through structurally different yet semantically equivalent control-flow realizations.

This observation highlights a limitation of purely sequential modeling: temporal chunking captures co-occurrence patterns but discards inter-procedural connectivity and global dependency structure.

Structural abstraction and inductive bias. The augmented API call graph introduces a different inductive bias. By merging multiple executions into a unified structural representation, it preserves connectivity, branching structure, and shared behavioral prefixes across runs. Divergence points are encoded explicitly rather than implicitly through statistical aggregation.

Although the flip classifier is not a malware detector, it operates over graph-derived paths and therefore inherits this structural bias. Its construction illustrates how behavior can be represented in terms of connectivity and dependency rather than purely local temporal context.

Sequential segmentation discards inter-procedural connectivity, long-range dependencies across execution paths, and structural relationships between divergent behaviors. Graph-derived representations retain these properties, making them conceptually better positioned to tolerate superficial perturbations such as reordered calls, injected noise, or exploration-induced path variation.

Table 5.4. Guided exploration summary using the flip classifier.

Total Samples	43
Detected	39
Undetected	4
Detection Rate	90.7%

Implications for robustness. The observed sensitivity of the sequential model under distribution shift demonstrates that representation choice materially affects robustness under behavioral diversity. Structural abstraction offers a principled mechanism to encode invariances across execution variants by emphasizing dependency structure over incidental ordering. While a full graph-based malware classifier remains future work, the empirical degradation under controlled filtering supports the hypothesis that structural inductive bias is essential for stable generalization across behavioral variants.

RQ2. Sequential (n -gram) representations exhibit sensitivity to distribution shifts induced by increased behavioral diversity, indicating limited invariance to execution variants. Structural abstraction through augmented call graphs introduces an inductive bias toward connectivity and dependency, providing a principled pathway toward improved robustness and generalization across behavioral variants.

5.4.6 Structure-Aware Exploration Efficiency

The third research question evaluates whether structurally guided trace expansion can reduce redundant behavioral exploration while preserving detection effectiveness under bounded execution budgets.

Unlike naive aggregation—which evaluates all traces independently and without prioritization—the proposed mechanism expands the augmented graph selectively. Starting from an initial trace, graph-derived paths are evaluated using the sequential detector. Exploration terminates once malicious evidence exceeds a predefined threshold. When paths appear benign, the flip classifier identifies structurally divergent regions likely to correspond to alternative behavioral realizations. Traces consistent with these divergence points are prioritized and incorporated into the graph.

This strategy does not modify the malware classifier itself. Rather, it alters the order and selection of trace evaluation. The central question is whether such structural prioritization enables earlier detection decisions and reduces unnecessary expansion. Table 5.4 summarizes sample-level detection outcomes under guided exploration.

Early detection saturation. A key empirical observation is the concentration of successful detections in early exploration stages. For many detected samples, malicious behavior is identified in the first evaluated trace, and additional traces provide diminishing marginal benefit.

This contrasts with naive multi-trace aggregation, where traces are evaluated sequentially without prioritization and each additional execution increases both

computational cost and exposure to false positives. In the guided setting, trace selection is not uniform: paths identified as structurally expressive by the flip model are evaluated first, increasing the probability that early executions contain discriminative evidence.

The empirical result is early detection saturation: decisions are frequently reached before exhaustive exploration of available traces.

Exploration efficiency and statistical stability. The efficiency gains have direct statistical implications. Under OR-based aggregation, each additional trace constitutes an additional statistical trial (Section 5.4.4). As previously shown, repeated trials increase the probability that at least one local error propagates to the sample level. By reducing the number of evaluated traces required to reach a decision, structure-aware exploration effectively limits the number of trials.

Guided exploration therefore improves not only computational efficiency but also statistical stability. Early stopping reduces exposure to the error accumulation inherent in large-scale multi-trace aggregation.

Scope and limitations. This experiment is conducted on a limited subset of 43 samples and should be interpreted as a controlled study of exploration dynamics rather than as a comprehensive benchmark. The flip classifier functions as a prioritization mechanism rather than as a full graph-based malware detector. Accordingly, the results demonstrate efficiency and stability effects under structural guidance, not standalone superiority in classification accuracy.

RQ3. Structure-aware trace prioritization enables early detection saturation under bounded execution budgets. By concentrating evaluation on structurally informative behaviors and enabling early stopping, guided exploration reduces redundant executions and mitigates the false positive accumulation inherent in large-scale multi-trace aggregation.

5.4.7 Statistical Limits of Large-Scale Behavioral Exploration

The final research question synthesizes the preceding observations to examine the statistical phenomena that emerge under large-scale behavioral exploration and how they constrain scalable malware detection.

Multiple-testing and aggregation asymmetry. Under window- and trace-level OR-based aggregation, each additional execution introduces implicit hypothesis tests. Even when per-window false positive rates are below 1% (Table 5.2), repeated evaluation increases the probability that at least one unit is misclassified. As shown in Table 5.3, sample-level FPR increases monotonically with the number of aggregated traces.

This reflects a structural asymmetry: for malicious samples, additional traces increase the probability of observing revealing behavior; for benign samples, they increase the probability that at least one spurious activation occurs. Under OR-based aggregation, a single false positive dominates the sample-level decision.

Multi-trace detection therefore behaves analogously to a sequential multiple-testing procedure in which the effective family-wise error rate increases with the number of examined executions.

Extreme-value amplification under active search. Fuzzing differs from passive sampling because it actively explores behavioral space. As exploration depth increases, the probability of encountering rare behavioral variants grows. Imperfectly calibrated models may assign extreme confidence scores to such outliers.

Under OR-based aggregation, extreme outlier scores are particularly influential: a single high-confidence false activation can determine the sample-level outcome. As the number of evaluated windows or traces increases, the decision increasingly depends on the maximum prediction score across many trials. The system thus transitions from average-case behavior to an extreme-value regime. This shift helps explain why FPR growth can accelerate even when local error rates remain small.

Correlation and effective sample size. Trace-level observations are not independent. Executions of the same sample share code base, execution prefixes, and environmental context, leading to correlated evidence. Correlation limits the marginal sensitivity gains obtained from additional traces, while still permitting rare error events to accumulate.

Empirically, this manifests as early saturation of TPR alongside continued growth in FPR (Table 5.3). Additional traces provide diminishing sensitivity gains but continue to introduce new opportunities for false activation.

Scalability constraint. Taken together, these phenomena imply a practical scalability constraint. Increasing behavioral coverage through unbounded exploration does not guarantee improved detection quality. Beyond a certain point, additional executions primarily increase exposure to multiple-testing and extreme-value effects.

Coverage expansion alone is therefore insufficient; exploration must be statistically controlled. Scalable behavioral malware detection requires false-positive-aware aggregation mechanisms, exploration policies that prioritize informative traces, and principled stopping criteria to limit the effective trial count.

RQ4. Large-scale behavioral exploration induces multiple-testing, extreme-value, and correlation effects that constrain scalability. As the number of evaluated traces increases, sample-level false positive rates can grow sharply even when local decisions are conservative. Scalable malware detection therefore requires statistically aware aggregation and controlled exploration rather than unbounded execution coverage.

5.4.8 Discussion

The experimental results establish several structural and statistical properties of trace-based malware detection under execution variability.

First, single-trace classification is inherently limited because each execution represents only a partial observation of a stochastic behavioral space. Second, aggregating

evidence across executions improves recall but introduces a reliability–stability trade-off at the sample level. Third, dataset filtering that increases behavioral diversity exposes the sensitivity of purely sequential encodings to distributional shifts. Fourth, exploration exhibits diminishing returns: beyond a limited number of informative executions, additional traces contribute marginal sensitivity gains while disproportionately increasing exposure to error accumulation.

Taken together, the findings support four central conclusions:

- **Multi-trace aggregation changes the statistical regime of detection.** Multi-trace evidence substantially increases TPR; however, repeated behavioral exploration also amplifies the propagation of local classification errors to the sample level. As the number of evaluated traces grows, detection quality becomes governed not only by classifier accuracy, but by the interaction between aggregation policy, exploration depth, and error accumulation. Unbounded execution scaling therefore degrades specificity unless aggregation explicitly controls statistical instability (**RQ1**, **RQ4**).
- **Representation determines robustness under behavioral shift.** Performance degradation under capability-based filtering indicates that purely local n -gram encodings rely partly on surface-level regularities. Structural abstractions that preserve connectivity and divergence points provide a principled mechanism for improved invariance across execution variants (**RQ2**).
- **Exploration efficiency governs both cost and stability.** Structure-aware trace prioritization concentrates evaluation on informative behavioral regions, enabling early detection saturation. By reducing the effective number of evaluated traces, guided exploration simultaneously lowers computational overhead and limits multiple-testing exposure (**RQ3**).
- **Statistical scaling is a first-class design constraint.** Large-scale fuzzing induces multiple-testing, extreme-value amplification, and correlation effects. Detection systems must therefore be designed jointly with aggregation and exploration policies; local accuracy does not guarantee stability at scale (**RQ4**).

Collectively, these results demonstrate that multi-trace analysis is necessary but not sufficient for robust behavioral detection. Repeated behavioral exploration fundamentally changes the statistical regime under which detection operates: additional executions simultaneously increase behavioral coverage and the opportunity for error propagation. Naive aggregation of increasingly many executions therefore cannot, by itself, resolve structural limitations inherent in flat, statistically independent trace modeling. Detection performance depends critically on how traces are selected, represented, and integrated under explicit statistical constraints.

This synthesis motivates a shift from flat trace aggregation toward structured behavioral modeling combined with controlled exploration. Rather than treating additional executions as uniformly beneficial, scalable malware detection must explicitly manage evidence acquisition, structural abstraction, and statistical aggregation as coupled components of a unified system.

Scope and future directions. The results presented in this chapter constitute a controlled empirical analysis of multi-trace behavioral detection under execution variability. They establish the underlying phenomena rather than a finalized production-scale system evaluation.

A larger-scale validation and integration of the proposed framework are part of ongoing joint work and will be reported separately. The implementation and experimental framework developed for this study will be released following that work. A broader discussion of implications and future research directions is provided in Chapter 6.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis investigated how behavioral abstraction can strengthen malware detection under adversarial pressure and execution variability. Across two complementary contributions, it examined the limitations of technique-specific detection and single-trace reasoning, and proposed principled mechanisms to improve robustness, interpretability, and statistical stability in dynamic malware analysis.

The first contribution addressed process injection, an effective attack technique in modern Windows malware. Rather than relying on narrowly scoped signatures or recognizing specific injection variants, we identified three fundamental operations shared across injection strategies—memory allocation, payload writing, and execution triggering—and proposed detecting their minimal co-occurrence. This abstraction shifts detection from fragile pattern matching toward behaviorally grounded reasoning. Empirical evaluation on real-world malware confirmed that process injection remains dominant and evolving, with threat actors continuously adapting implementation choices in response to defensive pressure and public knowledge. The analysis further revealed that mainstream sandboxes underestimate its prevalence due to overly strict heuristics. By raising detection from technique-specific artifacts to operationally grounded behavioral semantics, the proposed methodology increases robustness against variant proliferation and implementation shifts.

The second contribution extended the abstraction principle beyond a specific technique to the broader challenge of execution variability in dynamic malware analysis. Modern malware frequently exhibits environment-dependent behavior and conditional logic, challenging the implicit assumption in large-scale pipelines that a single runtime trace sufficiently represents program intent. We showed that robust behavioral analysis increasingly requires multi-trace reasoning to approximate a program’s behavioral potential more faithfully. However, naive aggregation introduces statistical risks: repeated execution increases exposure to false-positive accumulation and reliability–stability trade-offs. Through controlled empirical analysis, this work characterized key structural and statistical properties of multi-trace detection. Structural abstraction introduces inductive bias toward connectivity and dependency, supporting robustness under behavioral diversity. Structure-aware exploration can reduce redundant trace evaluation and limit exposure to multiple-testing effects

under bounded execution budgets. More broadly, scalable malware detection requires the joint design of aggregation rules, representation choices, and exploration policies. Repeated behavioral exploration fundamentally alters the statistical regime under which detection operates: detection quality does not scale monotonically with execution count, but instead depends critically on how behavioral evidence is selected, structured, and integrated.

Taken together, these contributions argue for a shift in perspective: from flat, single-trace detection pipelines toward behaviorally grounded, structurally aware, and statistically controlled reasoning frameworks. Robust detection requires not only improved heuristics, but principled abstraction mechanisms that connect low-level behavioral evidence to higher-level semantic interpretation while explicitly managing exploration and aggregation effects.

By unifying behavioral abstraction, multi-trace structural representation, and statistical analysis, this thesis advances both the conceptual foundations and practical design of dynamic malware detection systems.

6.2 Future Works

The results presented in this thesis open several promising research directions centered on principled behavioral abstraction for malware detection. Across both contributions, a common theme emerges: robustness does not arise from accumulating more low-level evidence, but from structuring that evidence through appropriate semantic abstraction and statistical control. Future work can therefore build upon this abstraction-centered perspective by extending, scaling, and formalizing the mechanisms introduced here.

A first direction concerns the generalization of primitive-based behavioral modeling. The minimal co-occurrence principle introduced for process injection demonstrates that technique-specific detection can be reframed in terms of underlying operational semantics. Extending this principle beyond Windows environments and beyond classical injection techniques would contribute toward a broader abstraction framework for memory-resident and execution-hiding strategies. In particular, systematically analyzing whether other stealth mechanisms—such as process ghosting [90]—can be expressed through analogous primitive compositions would support cross-technique semantic modeling grounded in observable system behaviors.

A second direction involves scaling structured multi-trace reasoning. While this thesis established foundational structural and statistical properties of multi-trace detection, comprehensive validation across larger datasets remains necessary. In parallel, integrated models trained directly on consolidated multi-trace representations could explore graph-based neural architectures to learn invariant structural patterns across execution variants. Such models should be evaluated not only for classification accuracy, but also for robustness under distributional shift and adversarial adaptation.

An additional extension lies in the study of implicit behavior modeling. A key open question concerns the conditions under which “virtual paths”—execution paths that do not explicitly occur in any single trace but emerge from consolidation across runs—enhances generalization versus amplifying structural noise.

Closely related is the development of statistically controlled exploration mechanisms. Naive multi-trace analysis assumes unrestricted execution budgets, whereas practical analysis settings impose strict time and resource constraints. Guided exploration strategies informed by structural divergence and classifier feedback offer a principled alternative. This direction can be formalized through adaptive stopping criteria, trace comparison methods grounded in graph traversal, and confidence-aware mutation prioritization. Such developments would position malware analysis within a broader framework of sequential decision theory, enabling exploration strategies that allocate computational resources based on measurable evidence gain.

Finally, abstraction-oriented malware detection should incorporate interpretability and analyst-centered reasoning. Structural representations create opportunities for extracting path-based explanations, identifying discriminative subgraphs, and quantifying behavioral uncertainty. Integrating explanation mechanisms with multi-trace reasoning would support not only automated classification, but also forensic validation and human-in-the-loop investigation.

Bibliography

- [1] Monnappa K A. Detecting Deceptive Process Hollowing Techniques Using Hollowfind Volatility Plugin. <https://cysinfo.com/detecting-deceptive-hollowing-techniques/>, 2016.
- [2] Monnappa K A. HollowFind. <https://github.com/monnappa22/HollowFind>, 2016.
- [3] Vitor Afonso, Anatoli Kalysch, Tilo Müller, Daniela Oliveira, André Grégio, and Paulo Lício de Geus. Lumus: Dynamically Uncovering Evasive Android Applications. In *Proceedings of the 21st International Conference on Information Security (ISC '18)*, pages 47–66. Springer International Publishing, 2018.
- [4] Asad Afreen, Moosa Aslam, and Saad Ahmed. Analysis of Fileless Malware and its Evasive Behavior. In *Proceedings of the 15th International Conference on Cyber Warfare and Security (ICCWS '20)*, pages 1–8, 2020.
- [5] Ross Anderson, Chris Barton, Rainer Bölme, Richard Clayton, Carlos Ganán, Tom Grasso, Michael Levi, Tyler Moore, and Marie Vasek. Measuring the Changing Cost of Cybercrime. In *Proceedings of the 18th Workshop on the Economics of Information Security (WEIS '19)*, 2019.
- [6] Ömer Aslan Aslan and Refik Samet. A Comprehensive Review on Malware Detection Approaches. *IEEE Access*, 8:6249–6271, 2020.
- [7] Cristian Assaiante, Simone Nicchi, Daniele Cono D’Elia, and Leonardo Querzoni. Evading Userland API Hooking, Again: Novel Attacks and a Principled Defense Method. In *Proceedings of the 21st Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '24)*, pages 150–173. Springer Nature Switzerland, 2024.
- [8] MITRE ATT&CK. Event Triggered Execution. <https://attack.mitre.org/techniques/T1546/>, 2025.
- [9] MITRE ATT&CK. Event Triggered Execution: AppCert DLLs. <https://attack.mitre.org/techniques/T1546/009/>, 2025.
- [10] MITRE ATT&CK. Event Triggered Execution: AppInit DLLs. <https://attack.mitre.org/techniques/T1546/010/>, 2025.

- [11] MITRE ATT&CK. Event Triggered Execution: Application Shimming. <https://attack.mitre.org/techniques/T1546/011/>, 2025.
- [12] MITRE ATT&CK. Event Triggered Execution: Component Object Model Hijacking. <https://attack.mitre.org/techniques/T1546/015/>, 2025.
- [13] MITRE ATT&CK. Event Triggered Execution: Image File Execution Options Injection. <https://attack.mitre.org/techniques/T1546/012/>, 2025.
- [14] MITRE ATT&CK. ListPlanting. <https://attack.mitre.org/techniques/T1055/015/>, 2025.
- [15] MITRE ATT&CK. Process Injection. <https://attack.mitre.org/techniques/T1055/>, 2025.
- [16] MITRE ATT&CK. Process Injection: Asynchronous Procedure Call. <https://attack.mitre.org/techniques/T1055/004/>, 2025.
- [17] MITRE ATT&CK. Process Injection: Dynamic-link Library Injection. <https://attack.mitre.org/techniques/T1055/001/>, 2025.
- [18] MITRE ATT&CK. Process Injection: Extra Window Memory Injection. <https://attack.mitre.org/techniques/T1055/011/>, 2025.
- [19] MITRE ATT&CK. Process Injection: Portable Executable Injection. <https://attack.mitre.org/techniques/T1055/002/>, 2025.
- [20] MITRE ATT&CK. Process Injection: Process Doppelganging. <https://attack.mitre.org/techniques/T1055/013/>, 2025.
- [21] MITRE ATT&CK. Process Injection: Process Hollowing. <https://attack.mitre.org/techniques/T1055/012/>, 2025.
- [22] MITRE ATT&CK. Process Injection: Thread Execution Hijacking. <https://attack.mitre.org/techniques/T1055/003/>, 2025.
- [23] Mitre ATT&CK. Virtualization/Sandbox Evasion. <https://attack.mitre.org/techniques/T1497/>, 2025.
- [24] Erin Avllazagaj, Ziyun Zhu, Leyla Bilge, Davide Balzarotti, and Tudor Dumitras. When Malware Changed Its Mind: An Empirical Study of Variable Program Behaviors in the Real World. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. USENIX Association, 2021.
- [25] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS '10)*, 2010.
- [26] Thomas Barabosch, Niklas Bergmann, Adrian Dombek, and Elmar Padilla. Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps. In *Proceedings of the 14th Conference on Detection of Intrusions and Malware, and*

- Vulnerability Assessment (DIMVA '17)*, pages 209–229. Springer International Publishing, 2017.
- [27] Thomas Barabosch, Sebastian Eschweiler, and Elmar Gerhards-Padilla. Bee Master: Detecting Host-Based Code Injection Attacks. In *Proceedings of the 11th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '14)*, pages 235–254. Springer International Publishing, 2014.
- [28] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS '09)*, 2009.
- [29] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, 2006.
- [30] Tristan Bilot, Nour El Madhoun, Khaldoun Al Agha, and Anis Zouaoui. A Survey on Malware Detection with Graph Representation Learning. *ACM Computing Surveys (CSUR)*, 56(11), 2024.
- [31] Bitdefender. The Differences Between Static and Dynamic Malware Analysis. <https://www.bitdefender.com/en-us/blog/businessinsights/the-differences-between-static-malware-analysis-and-dynamic-malware-analysis>, 2023.
- [32] Frank Block. Some Experiments With Process Hollowing. <https://insinuator.net/2022/09/some-experiments-with-process-hollowing/>, 2022.
- [33] Pietro Borrello, Emilio Coppa, Daniele Cono D’Elia, and Camil Demetrescu. The ROP Needle: Hiding Trigger-based Injection Vectors via Code Reuse. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*, pages 1962–1970. Association for Computing Machinery, 2019.
- [34] Marcus Botacin. Fuzzing and Symbolic Execution for Multipath Malware Tracing: Bridging Theory and Practice via Survey and Experiments. *Digital Threats*, 5(4), 2024.
- [35] Marcus Botacin, Hojjat Aghakhani, Stefano Ortolani, Christopher Kruegel, Giovanni Vigna, Daniela Oliveira, Paulo Lício De Geus, and André Grégio. One Size Does Not Fit All: A Longitudinal Analysis of Brazilian Financial Malware. *ACM Transactions on Privacy and Security (TOPS)*, 24(2), 2021.
- [36] Marcus Botacin, Giovanni Bertão, Paulo de Geus, André Grégio, Christopher Kruegel, and Giovanni Vigna. On the Security of Application Installers and Online Software Repositories. In *Proceedings of the 17th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '20)*, pages 192–214. Springer International Publishing, 2020.

- [37] Nicola Bottura, Giorgia Di Pietro, Yuya Yamada, Daniele Cono D’Elia, and Leonardo Querzoni. Poster: All Right Then, (Don’t) Keep Your Secrets: Exposing API Hashing in Malware. In *Proceedings of the 22nd Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA ’25)*, pages 287–293. Springer Nature Switzerland, 2025.
- [38] Nicola Bottura, Daniele Cono D’Elia, and Leonardo Querzoni. Pfuzzer: Practical, Sound, and Effective Multi-path Analysis of Environment-sensitive Malware with Coverage-guided Fuzzing. In *Proceedings of the IEEE 10th European Symposium on Security and Privacy (EuroS&P ’25)*, pages 1121–1139, 2025.
- [39] BreakingMalwareResearch. Atom-bombing. <https://github.com/BreakingMalwareResearch/atom-bombing>, 2016.
- [40] Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, 1999.
- [41] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. *Automatically Identifying Trigger-based Behavior in Malware*, pages 65–88. Springer US, 2008.
- [42] CAPE. CAPE Sandbox. <https://capev2.readthedocs.io/en/latest/>.
- [43] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA ’08)*, pages 143–163. Springer Berlin Heidelberg, 2008.
- [44] Fabrício Ceschin, Marcus Botacin, Albert Bifet, Bernhard Pfahringer, Luiz S. Oliveira, Heitor Murilo Gomes, and André Grégio. Machine Learning (In) Security: A Stream of Problems. *Digital Threats*, 5(1), 2024.
- [45] Fabrício Ceschin, Marcus Botacin, Heitor Murilo Gomes, Felipe Pinagé, Luiz S. Oliveira, and André Grégio. Fast & Furious: On the Modelling of Malware Detection as an Evolving Data Stream. *Expert Systems with Applications*, 212:118590, 2023.
- [46] Checkpoint. Evasion techniques. <https://evasions.checkpoint.com/>.
- [47] Chocolatey. The Package Manager for Windows. <https://chocolatey.org/>.
- [48] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security ’03)*, 2003.
- [49] Cyberbit. New ‘Early Bird’ Code Injection Technique Discovered. <https://www.cyberbit.com/endpoint-security/new-early-bird-code-injection-technique-discovered/>, 2018.
- [50] Angelo Schranko De Oliveira and Renato José Sassi. Behavioral Malware Detection Using Deep Graph Convolutional Neural Networks. *TechRxiv*, 2019.

- [51] Daniele Cono D’Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 14th ACM Asia Conference on Computer and Communications Security (ASIACCS ’19)*, pages 15–27. Association for Computing Machinery, 2019.
- [52] Daniele Cono D’Elia, Lorenzo Invidia, and Leonardo Querzoni. Rope: Covert Multi-process Malware Execution with Return-Oriented Programming. In *Proceedings of the 26th European Symposium on Research in Computer Security (ESORICS ’21)*, pages 197–217. Springer International Publishing, 2021.
- [53] Giorgia Di Pietro, Daniele Cono D’Elia, and Leonardo Querzoni. Can You Run My Code? A Close Look at Process Injection in Windows Malware. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security (ASIACCS ’25)*, pages 1600–1616. Association for Computing Machinery, 2025.
- [54] Microsoft Windows Documentation. About Transactional NTFS. <https://learn.microsoft.com/en-us/windows/win32/fileio/about-transactional-ntfs>, 2021.
- [55] Microsoft Windows Documentation. Asynchronous Procedure Calls. <https://learn.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls?redirectedfrom=MSDN>, 2021.
- [56] Microsoft Windows Documentation. Event Tracing for Windows (ETW). <https://learn.microsoft.com/en-us/windows-hardware/drivers/devtest/event-tracing-for-windows--etw->, 2021.
- [57] Microsoft Windows Documentation. How User Account Control Works. <https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/how-it-works>, 2025.
- [58] Microsoft Windows Documentation. Section Objects and Views. <https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/section-objects-and-views>, 2025.
- [59] VirusTotal Documentation. Files. <https://docs.virustotal.com/reference/files>, 2025.
- [60] Daniele Cono D’Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. On the Dissection of Evasive Malware. *IEEE Transactions on Information Forensics and Security (TIFS)*, 15:2750–2765, 2020.
- [61] Daniele Cono D’Elia, Simone Nicchi, Matteo Mariani, Matteo Marini, and Federico Palmaro. Designing Robust API Monitoring Solutions. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 20(1):392–406, 2023.
- [62] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Computing Surveys (CSUR)*, 44(2), 2008.

- [63] Elastic. Detection Rules. <https://github.com/elastic/detection-rules>, 2026.
- [64] Csaba Fitzl. Injection. <https://github.com/theevilbit/injection>, 2019.
- [65] The Volatility Foundation. Volatility. <https://volatilityfoundation.org/>.
- [66] The Volatility Foundation. Malfind. <https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#malfind>, 2025.
- [67] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. A Systematical and Longitudinal Study of Evasive Behaviors in Windows Malware. *Computers & Security*, 113:102550, 2022.
- [68] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware Analysis and Classification: A Survey. *Journal of Information Security*, 05:56–64, 2014.
- [69] Daniel Gibert, Carles Mateu, and Jordi Planes. The Rise of Machine Learning for Detection and Classification of Malware: Research Developments, Trends and Challenges. *Journal of Network and Computer Applications*, 153:102526, 2020.
- [70] Statcounter GlobalStats. Desktop Operating System Market Share Worldwide. <https://gs.statcounter.com/os-market-share/desktop/worldwide>, 2026.
- [71] Google. AFL. <https://github.com/google/AFL>, 2021.
- [72] Floris Gorter, Cristiano Giuffrida, and Erik Van Der Kouwe. Enviral: Fuzzing the Environment for Evasive Malware Analysis. In *Proceedings of the 16th European Workshop on System Security (EUROSEC '23)*, pages 8–14. Association for Computing Machinery, 2023.
- [73] Hasherezade. Process Doppelganging. https://github.com/hasherezade/process_doppelganging, 2022.
- [74] Ashkan Hosseini. Ten Process Injection Techniques: A Technical Survey of Common and Trending Process Injection Techniques. <https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process>, 2017.
- [75] Marcus Hutchins. Portable Executable Injection For Beginners. <https://www.malwaretech.com/2013/11/portable-executable-injection-for.html>, 2013.
- [76] AV-TEST: The Independent IT-Security Institute. Malware Statistics. <https://www.av-test.org/en/statistics/malware/>.
- [77] Mossé Cyber Security Institute. Malware Injection Techniques: APC Injection. <https://library.mosse-institute.com/articles/2022/05/malware-injection-techniques-apc-injection/malware-injection-techniques-apc-injection.html>, 2022.

- [78] Kyriakos K. Ispoglou and Mathias Payer. malWASH: Washing Malware to Evade Dynamic Analysis. In *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT '16)*. USENIX Association, 2016.
- [79] Haodi Jiang, Turki Turki, and Jason T. L. Wang. DLGraph: Malware Detection Using Deep Learning and Graph Embedding. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA '18)*, pages 1029–1033, 2018.
- [80] Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (SP '11)*, pages 347–362, 2011.
- [81] Ilker Kara. Fileless Malware Threats: Recent Advances, Analysis Approach through Memory Forensics and Research Challenges. *Expert Systems with Applications*, 214:119133, 2023.
- [82] Yigitcan Kaya, Yizheng Chen, Marcus Botacin, Shoumik Saha, Fabio Pierazzi, Lorenzo Cavallaro, David Wagner, and Tudor Dumitras. ML-Based Behavioral Malware Detection Is Far From a Solved Problem. In *Proceedings of the 3rd IEEE Conference on Secure and Trustworthy Machine Learning (SaTML '25)*, pages 921–940, 2025.
- [83] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*, pages 121–132. Association for Computing Machinery, 2012.
- [84] Thomas N Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907*, 2017.
- [85] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, pages 287–301. USENIX Association, 2014.
- [86] Amit Klein and Itzik Kotler. Windows Process Injection in 2019. *Black Hat USA*, 2019.
- [87] David Korczynski and Heng Yin. Capturing Malware Propagations with Code Injections and Code-Reuse Attacks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*, pages 1691–1708. Association for Computing Machinery, 2017.
- [88] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *Proceedings of the 14th USENIX Security Symposium (USENIX Security '05)*. USENIX Association, 2005.

- [89] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS '21)*, 2021.
- [90] Gabriel Landau. What You Need to Know about Process Ghosting, a New Executable Image Tampering Attack. <https://www.elastic.co/blog/process-ghosting-a-new-executable-image-tampering-attack>, 2021.
- [91] Martti Lehto. *Cyber-Attacks Against Critical Infrastructure*, pages 3–42. Springer International Publishing, 2022.
- [92] Ce Li, Zijun Cheng, He Zhu, Leiqi Wang, Qiujuan Lv, Yan Wang, Ning Li, and Degang Sun. DMalNet: Dynamic Malware Analysis Based on API Feature Engineering and Graph Learning. *Computers & Security*, 122:102872, 2022.
- [93] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: A Survey. *Cybersecurity*, 1, 2018.
- [94] Peng Li, Limin Liu, Debin Gao, and Michael K. Reiter. On Challenges in Evaluating Malware Clustering. In *Proceedings of the 13th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '10)*, pages 238–255. Springer Berlin Heidelberg, 2010.
- [95] Shanxi Li, Qingguo Zhou, Rui Zhou, and Qingquan Lv. Intelligent Malware Detection Based on Graph Convolutional Network. *The Journal of Supercomputing*, 78(3):4182–4198, 2022.
- [96] Tal Liberman and Eugene Kogan. Lost in Transaction: Process Doppelgänger. *Black Hat EU*, 2017.
- [97] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting Environment-Sensitive Malware. In *Proceedings of the 14th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '11)*, pages 338–357. Springer Berlin Heidelberg, 2011.
- [98] Side Liu, Guojun Peng, Haitao Zeng, and Jianming Fu. A Survey on the Evolution of Fileless Attacks and Detection Techniques. *Computers & Security*, 137:103653, 2024.
- [99] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200. Association for Computing Machinery, 2005.
- [100] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. Longitudinal Study of the Prevalence of Malware Evasive Techniques. *arXiv preprint arXiv:2112.11289*, 2021.

- [101] Mandiant. Capa. <https://mandiant.github.io/capa/>.
- [102] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The Art, Science, and Engineering of Fuzzing: A Survey. *arXiv preprint arXiv:1812.00140*, 2019.
- [103] Nuno Martins, José Magalhães Cruz, Tiago Cruz, and Pedro Henriques Abreu. Adversarial Machine Learning Applied to Intrusion and Malware Scenarios: A Systematic Review. *IEEE Access*, 8:35403–35419, 2020.
- [104] Medium. Utilizing Image File Execution Options (IFEO) For Stealthy Persistence. <https://securityblueteam.medium.com/utilizing-image-file-execution-options-ifeo-for-stealthy-persistence-331bc972554e>, 2023.
- [105] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP '17)*, pages 1009–1024, 2017.
- [106] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (SP '07)*, pages 231–245, 2007.
- [107] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC '07)*, pages 421–430, 2007.
- [108] Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [109] Mohammed Nasereddin and Raad Al-Qassas. A New Approach for Detecting Process Injection Attacks Using Memory Analysis. *International Journal of Information Security*, 23(3), 2024.
- [110] Emeric Nasi. PE Injection Explained. <https://blog.sevagas.com/PE-injection-explained>, 2014.
- [111] Meisam Navaki, Geoffrey Alexander, Hooman Rokham, Aokun Chen, Michalis Faloutsos, Xuetao Wei, Daniela Seabra Oliveira, and Jedidiah R. Crandall. FAROS: Illuminating In-memory Injection Attacks via Provenance-Based Whole-System Dynamic Information Flow Tracking. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '18)*, pages 231–242, 2018.
- [112] Red Team Notes. Code & Process Injection. <https://www.ired.team/offensive-security/code-injection-process-injection>, 2019.
- [113] Red Team Notes. CreateRemoteThread Shellcode Injection. <https://www.ired.team/offensive-security/code-injection-process-injection/process-injection>, 2019.

- [114] Red Team Notes. Early Bird APC Queue Code Injection. <https://www.ired.team/offensive-security/code-injection-process-injection/early-bird-apc-queue-code-injection>, 2019.
- [115] Red Team Notes. Early Bird APC Queue Code Injection. <https://www.ired.team/offensive-security/code-injection-process-injection/early-bird-apc-queue-code-injection>, 2020.
- [116] Red Team Notes. NtCreateSection + NtMapViewOfSection Code Injection. <https://www.ired.team/offensive-security/code-injection-process-injection/ntcreatesection-+-ntmapviewofsection-code-injection>, 2020.
- [117] Red Team Notes. SetWindowHookEx Code Injection. <https://www.ired.team/offensive-security/code-injection-process-injection/setwindowhookex-code-injection>, 2020.
- [118] Red Team Notes. PE Injection: Executing PEs inside Remote Processes. <https://www.ired.team/offensive-security/code-injection-process-injection/pe-injection-executing-pes-inside-remote-processes>, 2021.
- [119] Red Team Notes. Reflective DLL Injection. <https://www.ired.team/offensive-security/code-injection-process-injection/reflective-dll-injection>, 2021.
- [120] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhen-dong Su. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, pages 829–844. USENIX Association, 2014.
- [121] Unprotect Project. Unprotect Project. <https://unprotect.it/>.
- [122] Unprotect Project. Extra Window Memory Injection. <https://unprotect.it/technique/extra-window-memory-injection/>, 2019.
- [123] FortiGuard Labs Threat Research. AtomBombing - A Brand New Code Injection Technique for Windows. <https://www.fortinet.com/blog/threat-research/atombombing-brand-new-code-injection-technique-for-windows>, 2016.
- [124] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and Classification of Malware Behavior. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08)*, pages 108–125. Springer Berlin Heidelberg, 2008.
- [125] Rinse and REpeat analysis. Early Bird Injection - APC Abuse. <https://rinseandrepeatanalysis.blogspot.com/2019/04/early-bird-injection-apc-abuse.html?m=1>, 2019.
- [126] Cuckoo Sandbox. Cuckoo Sandbox. <https://github.com/cuckoosandbox>.

- [127] Cuckoo Sandbox. Hooked APIs and Categories. <https://github.com/cuckoosandbox/cuckoo/wiki/Hooked-APIs-and-Categories>, 2015.
- [128] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. AV-class: A Tool for Massive Malware Labeling. In *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '16)*, pages 230–253. Springer International Publishing, 2016.
- [129] SigmaHQ. Sigma. <https://github.com/SigmaHQ/sigma>, 2026.
- [130] Rami Sihwail, Khairuddin Omar, and KA Zainol Ariffin. A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. *International Journal on Advanced Science Engineering and Information Technology (IJASEIT)*, 8(4–2):1662–1671, 2018.
- [131] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*. No Starch Press, 2012.
- [132] Kevin Z Snow, Srinivas Krishnan, Fabian Monroe, and Niels Provos. SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security '11)*. USENIX Association, 2011.
- [133] Chad Spensky, Hongyi Hu, and Kevin Leach. LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS '16)*, 2016.
- [134] Splunk. Security Content. https://github.com/splunk/security_content, 2026.
- [135] Jerre Starink, Marieke Huisman, Andreas Peter, and Andrea Continella. Understanding and Measuring Inter-process Code Injection in Windows Malware. In *Proceedings of the 21st Security and Privacy in Communication Networks (SecureComm '25)*, pages 490–514. Springer Nature Switzerland, 2025.
- [136] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS '16)*, 2016.
- [137] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, Detection and Analysis of Malware for Smart Devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, 2014.
- [138] Microsoft Defender Security Research Team. Uncovering Cross-Process Injection with Windows Defender ATP. <https://www.microsoft.com/en-us/security/blog/2017/03/08/uncovering-cross-process-injection-with-windows-defender-atp/?source=mmmpc>, 2017.

- [139] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of Machine Learning Techniques for Malware Analysis. *Computers & Security*, 81:123–147, 2019.
- [140] Vx Underground. Bazaar Collection. <https://vx-underground.org/Samples/Bazaar%20Collection>.
- [141] UNPACME. Malware Trends: Yearly 2023. <https://blog.unpac.me/2024/01/30/malware-trends-yearly/>, 2024.
- [142] Kevin van Liebergen, Juan Caballero, Platon Kotzias, and Chris Gates. A Deep Dive into the VirusTotal File Feed. In *Proceedings of the 20th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '23)*, pages 155–176. Springer Nature Switzerland, 2023.
- [143] VirusTotal. VirusTotal. <https://www.virustotal.com/>.
- [144] Juan Wang, Chenjun Ma, Ziang Li, Huanyu Yuan, and Jie Wang. ProcGuard: Process Injection Behaviours Detection Using Fine-grained Analysis of API Call Chain with Deep Learning. In *Proceedings of the 21th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom '22)*, pages 778–785, 2022.
- [145] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A Gunter, et al. You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS '20)*, 2020.
- [146] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. GoldenEye: Efficiently and Effectively Unveiling Malware’s Targeted Environment. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '14)*, pages 22–45. Springer International Publishing, 2014.
- [147] YARA. YARA’s Documentation. <https://yara.readthedocs.io/en/stable/index.html>.
- [148] Ilsun You and Kangbin Yim. Malware Obfuscation Techniques: A Brief Survey. In *Proceedings of the 5th International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA '10)*, pages 297–300, 2010.
- [149] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (SP '20)*, pages 1121–1138, 2020.
- [150] Google Project Zero. WinAFL. <https://github.com/googleprojectzero/win afl>, 2025.

-
- [151] Zikai Zhang, Yidong Li, Hairong Dong, Honghao Gao, Yi Jin, and Wei Wang. Spectral-Based Directed Graph Network for Malware Detection. *IEEE Transactions on Network Science and Engineering*, 8(2):957–970, 2021.