SAPIENZA
UNIVERSITÀ DI ROMA

Engaging with Malware: Coverage-guided Fuzzing and Dynamic Introspection to Analyze Environment-Sensitive and Obfuscated Windows Malware

Faculty of Information Engineering, Informatics, and Statistics
Ph.D. Program in Engineering in Computer Science (XXXVII cycle)

Nicola Bottura
ID number 1904944

Advisor
Prof. Leonardo Querzoni

Co-Advisor
Prof. Daniele Cono D'Elia

Academic Year 2025/2026

Thesis reviewed by:

Prof. Alessio Merlo

Dr. Marcus Botacin

Engaging with Malware: Coverage-guided Fuzzing and Dynamic Introspection to Analyze Environment-Sensitive and Obfuscated Windows Malware

PhD thesis. Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: bottura@diag.uniroma1.it

And To Those I Love, Thanks For Sticking Around

Abstract

This thesis develops automated techniques for the analysis of Windows malware that employs anti-analysis defenses and environment-dependent, multi-path behavior. It introduces two complementary systems that transform adversarial logic from an obstacle to a systematic source of insight.

Rather than pursuing full transparency of execution, the dissertation treats evasive behavior as an observable signal. This engagement-based perspective contrasts with traditional approaches that seek to conceal analysis artifacts and forms the conceptual basis for both contributions.

The first contribution is an automated deobfuscation pipeline for API hashing, a widely used technique that conceals dependencies by replacing plaintext API names with digests. A slice-guided static and dynamic analysis identifies comparison sites and state-update instructions, reconstructing hashing logic and hash-to-API mappings. Each specimen is then repurposed as a *hash oracle*: by injecting chosen strings and emulating only the recovered slice, the pipeline derives ground-truth digests using the malware's own code, avoiding manual re-implementation and improving robustness to routine variants. On a labeled set of eight Windows samples, the pipeline recovers hashing routines in 87.5% of cases and precisely identifies the hashing function in 62.5%. Applied to 21,991 samples exhibiting indicators of API hiding, it identifies 1,686 strong candidates for API hashing, demonstrating that the technique is both prevalent in the wild and amenable to automated deobfuscation.

The second and main contribution of this thesis is PFuzzer, a coverage-guided fuzzing framework for systematically exploring environment-sensitive execution paths. PFuzzer models environmental conditions as *fuzzable inputs*, automatically mutating system responses to induce latent behaviors. Evaluated on a curated corpus of 1,078 Windows malware samples, PFuzzer exposes additional conspicuous behavior in 42.39% of specimens and consistently outperforms state-of-the-art systems such as BluePill and Enviral in head-to-head comparisons.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Malware remains one of the most persistent and rapidly evolving threats in modern computing [121, 31]. As defensive technologies advance, adversaries continuously adapt, developing increasingly sophisticated mechanisms to conceal malicious intent and evade detection. This dynamic has created an arms race between attackers seeking to hide and analysts striving to reveal, as repeatedly documented in threat-landscape and malware-trend reports over the last decade [85, 50]. Within this race, a growing number of malware families employ *anti-analysis techniques* [81, 41, 24, 52] to detect, exploit, or manipulate the analyst's environment or complicating aspects of the code itself to mislead both human investigators and automated systems [63, 133].

The scale of this threat continues to expand at a remarkable pace [39, 125, 38]. According to the AV-TEST Institute, over 390,000 new malicious programs are registered every day, amounting to more than 12 million new variants per month [72]. Microsoft's threat intelligence reports indicate that its customers face over 600 million attempted cyberattacks daily, encompassing ransomware, credential theft, and spyware campaigns [97]. Meanwhile, IBM's Cost of a Data Breach Report 2024 notes that the global average cost of a data breach has reached 4.88 million USD—an all-time high [32]. These figures underscore a fundamental reality: malware is not only pervasive but also increasingly adept at evading both signature-based and behavioral detection, often flying under the radar of anti-malware systems. As a result, understanding how malware hides, reacts, and adapts has become central to advancing defensive capabilities. This limitation is not only conceptual: the evaluation in Chapter 5 shows that, on a curated corpus of 1,078 Windows malware samples, single-run sandbox executions miss additional conspicuous behaviors that our multi-path exploration uncovers for 42.39% of the dataset.

Traditional malware analysis approaches—whether static or dynamic—have largely operated under the assumption that successful analysis depends on trans-

parency: the ability to observe execution without being observed [70, 41]. Yet complete transparency has proven elusive. Modern malware can detect virtualization artifacts, recognize instrumentation frameworks, and adapt its behavior based on environmental cues. As a result, many automated systems capture only a subset of a sample's behavior or miss critical functionality altogether.

This thesis challenges that assumption. Rather than striving for invisibility, it explores the idea of engaging with malware's evasive logic—treating anti-analysis behavior itself as a meaningful signal for the analysis.

To that end, this research introduces two complementary approaches that embody this principle, each addressing a distinct dimension of concealment in Windows malware:

- **Semantic concealment** through API obfuscation, where malware hides the intent of its operations by replacing readable imports with opaque hash values to thwart static analysis.
- **Behavioral concealment** through environmental sensitivity, where malware adapts its execution based on properties of the underlying system or contextual triggers;

These two dimensions are complementary, as each targets one of the two primary methods in the malware analyst's toolbox—static and dynamic analysis—further complicating efforts to achieve comprehensive insight into a sample's behavior.

The first contribution focuses on API hashing, an anti-static technique used to obscure a binary's API dependencies. We propose a dynamic-introspection method that isolates and interprets hashing logic within a malware sample, effectively transforming the sample into a *hash oracle*. This enables the recovery of mappings between obfuscated hash constants and their corresponding API names without reimplementing or guessing the algorithm, greatly reducing manual reverse-engineering effort. Large-scale analysis reveals how real-world samples implement API-hiding techniques, emphasizing the need for automated deobfuscation tools to extend the analyst's capabilities.

The second contribution introduces PFuzzer, a coverage-guided fuzzing framework that automatically perturbs environmental characteristics to uncover multiple behavioral states in environment-sensitive malware. The challenge posed by such malware has been actively investigated for over nearly two decades [81, 15, 98]. In this research we propose a fundamentally different strategy compared to prior research: rather than minimizing its footprint striving for transparency, PFuzzer deliberately alters features such as system configuration, registry contents, and timing information to provoke divergent execution paths—including those in which

the sample attempts to evade analysis (for example, by terminating early or exhibiting only benign actions to conceal its true intent). This *environmental fuzzing* perspective transforms evasion into an exploration mechanism, revealing hidden behaviors that conventional single-run methods (*e.g.,* sandbox systems) often miss. Our experiments show that PFuzzer exposes additional conspicuous behaviors in 42.39% of the dataset and clearly outperforms state-of-the-art systems such as BluePill and Enviral in direct comparisons.

Taken together, these two contributions share a common vision: restoring analytical visibility not by hiding from malware, but by actively engaging with it. By exploiting the very mechanisms designed to thwart observation, our approaches open new avenues to study, interpret, and counter advanced anti-analysis behaviors.

Contributions

This thesis makes two principal contributions to the field of malware analysis, both grounded in the idea of dynamic engagement rather than static transparency:

- **Automated Deobfuscation of API Hashing via Dynamic Introspection.** The first contribution addresses the complementary challenge of code-level concealment. By dynamically tracing and slicing the hashing routines embedded within malware, we recover the logic responsible for computing and comparing API hashes. We then *repurpose* each sample as a self-contained hash oracle, enabling analysts to reconstruct hash→name mappings and infer underlying hashing algorithms at scale. This approach was first validated on a training dataset of eight real-world samples to demonstrate its effectiveness, and subsequently applied to a large corpus of 21,991 samples exhibiting API-hiding behavior to characterize API hashing in the wild, leading to 1,686 candidates. This contribution reduces manual reverse-engineering effort and provides a reproducible, automated method for analyzing a pervasive obfuscation technique.

- **Fuzzing the Windows Environment to Discover Multiple Behaviors in Malware.** Our second contribution, PFuzzer, introduces a systematic approach to fuzz the environmental features of Windows malware in a coverage-guided manner. It demonstrates that mutating environmental properties—rather than program inputs, as is common in traditional fuzz testing—can expose additional behaviors that single-run systems and state-of-the-art solutions fail to capture. Through extensive experimentation, PFuzzer reveals that many samples contain latent logic activated only under specific environmental conditions, which can be automatically discovered through envi-

ronmental exploration. This contribution was evaluated on a curated dataset of 1078 environment-sensitive, real-world malware, bridging dynamic analysis and fuzz testing to establish a new methodology for exploring evasive malware behavior.

While the two problems studied in this thesis may initially appear as independent choices, their selection is grounded in three criteria. First, empirical prevalence: large-scale analyses and industry reports consistently show that environment fingerprinting and multi-path execution are among the most common anti-dynamic-analysis strategies, and that API hashing is a standard mechanism to conceal imports in modern malware [81, 15, 130, 4]. Second, a clear methodological gap: environment-sensitive behavior is typically explored through a single sandbox run that may miss additional relevant behavior [70, 41], and API hashing is almost always handled manually by analysts, with no widely adopted automated workflow [114, 60]. Third, complementarity: the first contribution addresses semantic concealment in static code, while the second targets behavioral concealment at runtime, together covering the two primary axes along which malware hinders analysis.

Although these two contributions operate at different stages of an analyst's workflow, they are complementary. In a realistic pipeline, automated recovery of hidden API dependencies can guide dynamic exploration by indicating which subsystems and environmental resources a sample is likely to query, while multi-path exploration exposes additional code paths whose API usage can, in turn, be fed back into static deobfuscation. Exploring such integration is an interesting direction for future work.

1.1 Thesis Outline

The remainder of this thesis is organized as follows:

- **Chapter 2—Background:** Introduces the foundational concepts and technical context underlying both contributions, highlighting the challenges posed by anti-analysis techniques. This chapter is further structured into sections that cover malware analysis, anti-analysis methods in malicious software, program analysis and dynamic binary instrumentation, and fuzz testing techniques.
- **Chapter 3—State of the Art:** Surveys prior work on environment-sensitive malware and multi-path exploration; reviews approaches complementary to PFuzzer; and examines API-hiding mechanisms relevant to our first contribution. This chapter also analyzes the limitations of transparency-driven

methods, motivates our choice of *active engagement* as a guiding principle, and explains why related approaches fall short as reliable analysis frameworks.

- **Chapter 4—Automated Deobfuscation of API Hashing via Dynamic Introspection:** Describes our first contribution, an automated approach for deobfuscating API hashing through dynamic introspection. It details the motivation, design, implementation, dataset construction, and evaluation, divided into two parts: (1) an assessment of effectiveness on a small, labeled dataset, and (2) an exploratory large-scale study that analyzes hashing algorithms and extends the dataset as a first step toward future research.

- **Chapter 5—Fuzzing the Windows Environment to Discover Multiple Behaviors in Malware:** Presents PFuzzer, our second and main contribution. The structure mirrors that of the previous chapter, with a particular focus on dataset construction and on the evaluation of our environment-fuzzing framework, which reveals hidden behaviors in real-world malware samples.

- **Chapter 6—Conclusion and Future Work:** Summarizes our contributions and discusses future research directions for extending both of them.

Chapter 2

Background

Malware—malicious code designed to disrupt, exfiltrate, or control systems—remains a pervasive threat across all sectors, ranging from traditional file-based samples to modern fileless and modular families. Analyzing such software is fundamental to developing and evolving effective defensive solutions. Malware analysis relies on two complementary perspectives: **static** methods, which inspect binaries without execution, and **dynamic** methods, which observe program behavior at runtime [89, 20].

However, the analysis process is rarely straightforward. Modern malware frequently enforces **anti-analysis** techniques that deliberately conceal intent or suppress behavior when analysis is detected or suspected. These techniques include packing, control-flow and data-flow obfuscation, debugger and virtual machine checks, API hiding, and many others. Notably, several of these mechanisms are also found in benign software—for instance, commercial applications and games often employ obfuscation, anti-tamper measures, and environment checks (*e.g.,* emulator or debugger detection) to protect integrity and intellectual property [61, 19].

This chapter introduces the foundational concepts underpinning our work. Specifically, we: (1) summarize the main malware analysis paradigms and their respective trade-offs; (2) survey anti-analysis methods, with particular emphasis on environment sensitivity and API-hiding techniques; (3) outline core program-analysis concepts and the dynamic binary instrumentation (DBI) frameworks on which our work builds; and (4) present a dedicated section on fuzzing, focusing on coverage-guided techniques—the underlying method employed in our primary contribution.

Together, these elements provide the technical foundation for the two contributions of this thesis: exposing multiple program states in environment-sensitive malware, and deobfuscating and analyzing malware that conceals API calls through API hashing.

2.1 Malware Analysis

Because Windows dominates the desktop and server ecosystems used by businesses, public institutions, and critical infrastructures such as hospitals, the vast majority of malicious samples are designed to run on Windows [125, 38]. Attackers *favor* Windows not only for its extensive installed base, but also because Windows-native *Portable Executable (PE)* binaries are well understood, straightforward to build and deploy, and compatible with a wide range of legacy tooling and packers. For similar reasons of compatibility and reach, many samples are distributed as 32-bit PE files: compiling as 32-bit (*PE32*) maximizes the number of machines capable of executing the code and leverages mature packing and obfuscation toolchains that remain more prevalent for 32-bit binaries [120, 134].

Since malware accounts for a substantial portion of computer intrusions and security incidents, the *malware analysis* process is fundamental. It refers to the systematic dissection of malicious software to understand its behavior, identify its presence, and devise strategies to mitigate or eliminate its impact. Insights gained through analysis enable the development of detection signatures that can identify infections across systems and networks.

Detection signatures fall broadly into two categories: *host-based* and *network-based* [123]. *Host-based* signatures detect malicious code on compromised systems, often by identifying files created or modified by the malware or by recognizing specific registry changes. *Network-based* signatures, in contrast, detect malicious activity by monitoring network traffic. These signatures are stored in extensive databases and leveraged by intrusion detection systems (IDS), intrusion prevention systems (IPS), and antivirus products to recognize or block known malware and related threats.

In many cases, analysts are provided only with the malware executable itself, which is not human-readable. To make sense of it, they employ a range of tools and techniques, each exposing a partial view of the underlying logic. Two fundamental approaches underpin this process: *static analysis* and *dynamic analysis*. This high-level taxonomy follows standard malware-analysis references [123], which also distinguish between basic, metadata-oriented techniques and advanced, code-centric methods. *Static analysis* examines the malware without executing it, whereas *dynamic analysis* runs the malware in a controlled environment to observe its runtime behavior.

2.1.1 Static Analysis

The malware analysis literature commonly distinguishes between *static* and *dynamic* techniques, a high-level taxonomy that is consistently adopted both in academia

and practitioners works on the topic [123, 122].

Within this work, and following the didactic classification proposed by Sikorski and Honig [123], we further distinguish between *basic* and *advanced* variants of both static and dynamic analysis. We emphasize that this basic/advanced subdivision is a convenient way to structure the exposition rather than a universally standardized taxonomy.

Static analysis involves examining an executable file without executing it. Because the sample is not run during this phase, static analysis is the safest approach: since no code is executed, the analyst's host cannot be infected. Static analysis can be divided into two main subcategories:

- *Basic Static Analysis*: examines the executable using external tools and metadata extraction rather than inspecting instruction-level code. Typical techniques include inspecting file headers, import tables, embedded strings, file hashes, packer or obfuscator signatures, and entropy measurements. This type of analysis helps determine whether a sample is likely malicious and provides high-level information about its functionality. It is straightforward and can be performed quickly, but it is largely ineffective against sophisticated or heavily obfuscated malware and may miss critical behaviors.

- *Advanced Static Analysis*: entails reverse-engineering the program by loading the executable into a disassembler (*e.g.,* IDA Pro) or decompiler and analyzing its instructions and control flow to understand precisely what the program does. This approach can reveal detailed logic, algorithms, and subtle behaviors, but it is time-consuming and requires substantial expertise in assembly language, calling conventions, and reverse-engineering techniques.

2.1.2 Dynamic Analysis

In contrast to static analysis, *dynamic analysis* executes the sample and observes its behavior at runtime. Because the code actually runs, dynamic analysis can reveal actions and interactions that are invisible to static techniques. However, it must always be conducted in a safe, isolated test environment—or by using reputable sandboxing services—to eliminate any risk to the analyst's host or network. Mirroring the convention adopted above, we distinguish between basic and advanced dynamic analysis techniques:

- *Basic Dynamic Analysis*: similar to *Basic Static Analysis*, this level does not require advanced programming or reverse-engineering skills. The executable is launched in a controlled environment, and monitoring tools are used to collect high-level runtime artifacts (*e.g.,* Procmon, Process Explorer,

WIRESHARK). Basic dynamic analysis provides an overview of file system, registry, process, and network activity, but it may miss stealthy or time-delayed behavior and can produce false positives.

- *Advanced Dynamic Analysis*: employs debuggers and instrumentation to inspect the internal state of a running sample. Techniques range from stepping through execution with a debugger to constructing complex analysis systems (*e.g.,* those based on dynamic binary instrumentation). This approach can expose unpacking mechanisms, decryption routines, and evasive logic, but it is time-consuming and requires extensive expertise in debugging, operating-system internals, and anti-analysis countermeasures [122].

In practice, analysts and automated systems often combine static and dynamic approaches (*hybrid* analysis), as each family of techniques uncovers information that the other may overlook.

Following a more unified convention [122], malware-analysis techniques are commonly grouped into three main categories: *static*, *dynamic*, and *hybrid* analysis, without further subdividing each category into "basic" and "advanced". Static and dynamic analysis largely correspond to the families of techniques described earlier (ranging from lightweight inspection/monitoring to more in-depth reverse engineering and instrumentation). Hybrid analysis combines evidence gathered from both static and dynamic perspectives, with the goal of improving detection and behavioral understanding compared to using either one in isolation [47]. While hybrid approaches can leverage the complementary strengths of static and dynamic analysis, they typically require additional time and computational resources, which may limit scalability in large-scale settings.

2.1.3 Applications of Analysis Techniques

The fundamental principles of malware analysis—static and dynamic inspection—have inspired a wide range of derivative and extended approaches that seek to enhance automation, completeness, and robustness against evasion. Among these, commercial sandbox systems, machine learning-based detection, and dynamic program instrumentation have become particularly prominent.

To observe a program's behavior directly—and because static reasoning is often inconclusive—modern analysis frameworks employ *sandbox environments* [37, 62]. A sandbox provides an instrumented, isolated execution context, typically implemented via virtualization or containerization, that allows analysts to execute untrusted binaries safely. During execution, the sandbox records behavioral indicators such as file operations, registry changes, and network activity, which are later interpreted

to assess malicious intent. Despite their effectiveness, sandboxes face two major challenges. First, malware often includes *anti-sandbox* logic designed to detect and evade execution under virtualized conditions [87]. Second, most sandboxes execute each sample only once, producing a single behavioral trace that may omit conditional or environment-dependent behaviors. As discussed in Chapter 5, this limitation hinders the analysis of malware families exhibiting multi-path execution strategies.

The rise of artificial intelligence has further expanded the analytical toolkit. Recent research explores *end-to-end machine learning* (ML) approaches that process binaries directly as raw byte sequences, eliminating the need for handcrafted features. For instance, Raff *et al.* [112] trained neural networks that ingest complete PE files and automatically learn discriminative representations, while Krčál *et al.* [74] proposed compact convolutional models capable of recognizing malicious byte patterns without expert-defined preprocessing. These results demonstrate that deep learning (DL) can autonomously extract meaningful representations from executables and achieve performance comparable to traditional feature-based pipelines, even though the learned internal representations are often not directly interpretable as human-designed features. However, DL models remain vulnerable to adversarial perturbations and sensitive to dataset bias and evolving malware distributions [29]. Consequently, they are best viewed as *complementary* to conventional analysis—enriching rather than replacing established methodologies.

Another significant extension of dynamic analysis is *dynamic program analysis*, which systematically inspects a program's execution through *instrumentation*. Software instrumentation frameworks such as Pin [86] and DynamoRIO [124] insert monitoring instructions into binaries to track control flow, memory access, or API usage in real time, enabling the construction of complex analysis systems. While instrumentation can be applied statically at the source level, malware analysis typically relies on *dynamic binary instrumentation* due to the lack of source code. This technique underpins numerous research and industrial systems [41, 108], enabling fine-grained inspection of obfuscated or self-modifying code. Although it introduces runtime overhead and implementation complexity, dynamic instrumentation provides strong resilience against anti-analysis mechanisms and forms the methodological foundation for several contributions in this thesis.

2.2 Anti-Analysis Methods in Malicious Software

Analyzing malicious code in controlled environments is a cornerstone of modern cyber defense. By executing malware samples safely, practitioners and researchers can directly observe malicious behaviors and derive detection and mitigation strategies [81].

As defensive capabilities have advanced, adversaries have responded with increasingly sophisticated tactics to evade analysis [25]. In this ongoing cat-and-mouse game, modern malware frequently incorporates *anti-analysis* techniques designed to detect virtualized or instrumented environments, frustrate static analysis, identify monitoring tools (*e.g.,* via anti-debugging checks), or rely on atypical environment-specific characteristics that are unlikely to appear in analyst sandboxes but are expected on real victim systems [24].

The remainder of this chapter examines these techniques in greater detail, providing the context needed to understand how they hinder or evade analysis and motivating the approaches presented later in this thesis.

2.2.1 Environment-Sensitive malware

The term *environment-sensitive malware* covers a broad and sometimes ambiguous class of malicious programs whose runtime behavior depends on properties of the host system. Prior literature has examined this notion from multiple angles: Lindorfer *et al.* [81] characterized many samples in terms of sandbox-evasion tactics, Xu *et al.* [141] investigated binaries that embed customized environment checks to activate only on particular victim hosts, and other work has focused on detecting and bypassing hypervisor-related artifacts [41]. These studies illustrate the diversity of motivations and techniques that fall under the environment-sensitive umbrella.

More concretely, environment sensitivity denotes any dependency of program logic on environmental inputs or host characteristics. On one end of the spectrum are simple anti-sandbox checks—what we term *evasive* malware—which attempt to detect analysis infrastructure (sandboxes, debuggers, virtual machines, instrumentation) and suppress or alter malicious behavior when such artifacts are present [81, 15, 70, 51]. On the other end are *targeted* samples that implement query-then-infect logic, executing payloads only when the host meets attacker-defined conditions (for example, the presence of specific software, particular language or locale settings, domain membership, or other site-specific attributes) [98, 141].

These categories are not mutually exclusive: a single specimen may combine evasion checks to avoid analysis and targeting checks to restrict impact to selected victims. We consider a sample environment-sensitive if (i) it issues explicit environment queries (*e.g.,* file/registry/OS attribute/process checks), and (ii) at least one such query guards a control-flow decision that leads to materially different behaviors (observable actions or executed code regions) under feasible alternative responses. We exclude benign run-to-run nondeterminism unless it is explicitly read and used by the program as a decision input. In practice, many prominent malware families combine these strategies. Banking trojans and loaders routinely

check for virtualization artifacts, debuggers, or specific processes before activating their payloads, while ransomware families often gate encryption logic on locale or domain membership to avoid unattractive targets [10, 2, 127]. These concrete uses of environment queries motivate analysis techniques that can systematically expose hidden behaviors.

A central insight motivating this thesis is that environment sensitivity is a broader property than evasiveness. Even malware that exhibits clearly malicious actions in a baseline run can reveal additional—or qualitatively different—behaviors when one or more environmental features are changed [24]. Standard analysis systems such as sandboxes—which typically execute a sample once and record a single behavioral trace—often miss this dimension of variability. Prior approaches to uncover such hidden behavior have included *forced execution* and fine-grained program-analysis techniques (for example, symbolic execution or taint analysis) that attempt to infer dependencies between code paths and runtime-derived environmental values.

From our experiments, the most common environment-sensitive queries involved file-system and registry checks, time- and scheduling-related queries, and process enumeration. We also observed cases in which samples probed for environmental artifacts tied to vulnerabilities, enabling privilege escalation or stealthier operation. Identifying these queries is valuable both for understanding attacker intent and for designing analysis methods capable of exposing conditional logic.

Chapter 5 introduces PFuzzer, our environment-fuzzing framework, which systematically perturbs the environmental characteristics perceived by a sample and thereby exercises alternate execution paths that single-run methods typically miss. Because many evasive behaviors are implemented as environmental queries, those same queries serve as natural indicators of broader environment sensitivity; the next sections describe in detail the common query types and checks employed by malware.

Evasive Malware

For newcomers to malware analysis, one of the first and most important guidelines is to perform experiments in a virtualized and isolated environment, such as a sandbox or virtual machine-based system, to prevent harm to sensitive data and systems. However, virtualization and analysis methods (*e.g.,* instrumentation) inevitably introduce specific *artifacts* that can reveal the presence of an analysis environment to the malware itself. Once these artifacts are detected, many samples alter their execution—terminating early, idling, or exhibiting only benign behavior—to conceal their true intent. Malware that actively performs such environmental fingerprinting is referred to as *evasive malware*, a term derived from the process of *fingerprinting* [51].

The goal of these techniques is to remain invisible to automated analysis systems and thus prolong the operational lifetime of the malicious campaign.

Several observable factors can distinguish an analysis environment from a real, bare-metal system. Common examples include:

- *Virtualization artifacts*: Instruction-level probes can reveal virtualized execution. For instance, the classic *red pill* sequence measures the latency of the `cpuid` instruction, which typically triggers a VM-exit in a virtualized environment. By comparing the measured timing with expected hardware latencies, malware can infer the presence of a hypervisor.
- *Hardware characteristics*: Emulated hardware often exposes distinctive properties, such as the CPU model and core count, MAC address ranges, disk identifiers, or the absence of certain peripherals (*e.g.,* audio devices or temperature sensors). These anomalies can signal a virtual environment.
- *Windows installation context*: System metadata such as timezone, language, uptime, and installation date can betray virtualization. Hypervisor guest additions (*e.g.,* VirtualBox or VMware Tools) also leave registry entries, processes, and drivers that can serve as reliable indicators.
- *Applications*: The presence or absence of particular software can affect how malware behaves. Some samples require a specific application to trigger a payload; others disable antivirus products or evade analysis tools. The presence of utilities like IDA Pro or debuggers, rarely found on ordinary systems, can alert malware to an analyst's environment.
- *User artifacts*: A minimal browsing history, empty "Recent Files" list, or small number of user documents may indicate a freshly installed or snapshot-restored VM. Malware can use these cues to distinguish real user environments from controlled analysis systems.

Beyond these examples, evasion checks can be systematically categorized according to the system component or data source they query.

- *Filesystem*: these methods rely on the presence of files or directories that do not normally exist on standard hosts but are present in specific virtual environments or sandboxes. The detection of such artifacts can reveal that the sample is being executed in a virtual environment.
- *Registry*: certain registry keys are specific to virtual environments and are absent on typical physical hosts. However, false positives may occur if the

system legitimately has virtualization software installed, in which case some VM-related artifacts may also appear on a normal host.

- *Generic OS queries*: virtual environments often exhibit operating-system characteristics that differ from standard hosts. For example, real systems usually have meaningful and non-generic usernames and computer names, while many virtual environments assign predefined default values. Other detectable differences may include RAM size, disk size, or the number of monitors. Although these checks are not the most reliable indicators of virtualization, they remain commonly employed by malware samples.

- *Global OS Objects*: detection of global objects such as mutexes, virtual devices, or named pipes that are absent in physical hosts but present in virtual environments.

- *UI artifacts*: these techniques rely on differences in the graphical user interface. For example, some window names exist only in virtual environments, not on real hosts. In addition, real systems typically have many open windows, whereas VMs and sandboxes tend to keep the number of active windows minimal, which can be used as a detection heuristic.

- *OS features*: these evasions exploit peculiarities of the operating system, such as checking for debug privileges or leveraging inconsistencies like unbalanced stack operations.

- *Processes*: virtual environments often launch helper processes that are not executed on physical hosts; similarly, some virtualization-related modules may be loaded into process address spaces.

- *Network*: these techniques examine network parameters (*e.g.,* MAC addresses) or invoke network-related functions to identify a virtual environment.

- *CPU*: some evasions rely on specific processor instructions to retrieve CPU information or to execute predefined instruction sequences that behave differently on virtualized systems (*e.g.,* `cpuid`).

- *Hardware*: virtual environments emulate hardware devices and often leave distinctive traces in their descriptions, which can be queried by malware.

- *Firmware tables*: operating systems use reserved memory areas that may contain artifacts when running under virtualization. These firmware tables can be dumped and inspected, though methods vary across OS versions.

- *Hooks*: malware may check for unusual hooks installed in the system, or exploit hooks to detect user presence. Such hooks are uncommon on physical hosts but often appear in analysis environments.
- *Timing*: sandboxed emulation is usually time-constrained due to high sample throughput, rarely exceeding 3-5 minutes. Malware can exploit this by introducing long delays before executing any malicious activity, thereby evading detection.
- *WMI*: Windows Management Instrumentation (WMI) queries can reveal OS and hardware information by leveraging COM interfaces and their methods.
- *Human-like behavior*: these techniques rely on distinguishing between automated environments and real users. For instance, malware may check the number of recently opened documents, the length of browser history, or similar user-driven artifacts.
- *macOS*: there also exist evasion techniques specific to macOS hosts, often based on shell commands such as SYSCTL and IOREG.

As an illustrative example, Listing 1 shows C-like pseudocode for a simple evasive routine that checks for a VirtualBox artifact and measures `cpuid` timing before performing network activity.

The code listed performs a first, simple attempt at identifying a hypervisor-based environment—in this example, VirtualBox—by querying for a file commonly present on such virtualized systems. If that file exists, the sample terminates immediately. If not, the sample proceeds to a second check: it measures the time required to execute the `cpuid` instruction and uses the observed latency as a heuristic for virtualization. Only when these checks indicate that the underlying environment is sufficiently "real" (*i.e.,* the evasive probes do not reveal an analysis sandbox or VM) does the sample proceed to execute its malicious payload (in the pseudocode, by initiating a network connection via the wrapper call `API_HTTPConnectTo`). In practice, authors may interleave such probes with benign-looking operations or with actual malicious actions, making detection and analysis more difficult.

Malware authors frequently stack an arbitrary number of evasive checks in varied orders, combine timing- and artifact-based tests, and mix these with payload execution in ways that frustrate simple single-run analysis. Consequently, evasive malware has historically slowed down or even defeated many automated analysis systems. Studying these techniques and developing robust countermeasures is therefore essential for accurate and comprehensive analysis. Approaches such as BLUEPILL [41] attempt to increase transparency by removing or masking known

artifacts on demand, improving the chance that a sample will reveal its true behavior. However, as discussed throughout this thesis, artifact removal alone is insufficient: more sophisticated or novel detection strategies can still discern analysis environments, and targeted checks that depend on legitimate host attributes remain outside the remit of artifact-suppression methods.

```
    bool check1 = API_checkIfFileExists("VBoxDisp.dll");
    if (check1 != false) {
        API_exit(0);
    }

    int cpuInfo[4];
    long startTime = _rdtsc();
    _cpuid(cpuInfo, 1);
    long check2 = _rdtsc() - startTime;
    if (check2 > 1000) {
        API_exit(0);
    } else if (check2 < 50) {
        executeStallingSequence();
    }

    API_HTTPConnectTo("https://www.evilmalware.com");
```

Listing 1. C-like pseudocode evasive code that tries to detect virtualized environments or analysis systems (*e.g.,* sandboxes) that tries to bypass timing evasive attempts.

For readers who wish to explore further, there are community-maintained catalogs that document known evasive techniques and their implementations; representative resources include the collections maintained by CheckPoint Research and Unprotect project [35, 128].

To illustrate how environment-driven control flow can produce entirely different behaviors, consider what happens if the code in Listing 1 is extended with a simple environment-based branch, as shown in Listing 2. In that example, the presence of a seemingly innocuous file (Tool.exe) causes the program to perform a local process-injection action, whereas its absence causes the program to contact a remote command-and-control server. A conventional sandbox that executes the sample only once would observe at most one of these branches and could therefore miss crucial, context-dependent behavior.

```
    bool check = API_checkIfFileExists("C:\\Tool.exe"):
    if (check) {
        inject_target_process("Tool.exe");
    } else {
        contact_remote_C2("200.123.12.41");
    }
```

Listing 2. C-like pseudocode that changes its behavior based on an environmental characteristic (presence of a file unrelated to known analysis environments).

In this example, the environmental condition is unrelated to any explicit attempt at analysis evasion: the decision is triggered by the presence of a "benign" artifact on the system. Such dependencies make the analysis problem substantially harder, as they introduce behavioral diversity that cannot be systematically explored through single-run observation. Detecting and exercising these alternative execution paths requires analysis strategies capable of reasoning about, or actively mutating, environmental states. The approach presented in this thesis directly targets this challenge by enabling the automated exploration of environment-dependent behaviors, ensuring that multiple execution paths—whether driven by evasive checks or by ordinary contextual dependencies—are observed and analyzed comprehensively.

2.2.2 Hiding API Calls

API hiding denotes a broad class of techniques designed to conceal which operating-system services a binary invokes from static and dynamic analysis. Common strategies include minimizing or wiping the import address table (IAT), delaying resolution via `LoadLibrary`/`GetProcAddress` (including ordinal lookups), manual/reflective mapping of DLLs, and obfuscating strings or internal structures (*e.g.,* encryption/`xor`, case mangling, forwarding indirection). More aggressive schemes employ name-to-value encodings (for example, API hashing) or reconstruct imports at runtime to defeat static signatures and trivial memory scans. [8, 36, 9]

Closely related are *anti-monitoring* and *anti-hook* techniques that attempt to subvert runtime instrumentation and API-interposition mechanisms. Malware may detect or bypass userland and kernel hooks, tamper with hooking tables, invoke syscalls directly, or route functionality through injected or stolen code to evade API monitors and sandboxes. Prior work demonstrates that robust API monitoring typically requires a combination of taint tracking, memory inspection, and import-reconstruction: naïve interception of well-known APIs can be subverted by runtime

obfuscation and manual resolution strategies [68, 8].

These challenges motivate obfuscation-resilient payload-extraction and deobfuscation approaches that recover high-level API usage from process memory and execution traces rather than relying solely on static names or intercepted calls. Systems that reconstruct imports from memory or extract payloads after unpacking (for example, API-XRAY and other API-reconstruction approaches) illustrate the effectiveness of memory- and trace-driven analysis against packed and heavily obfuscated samples [36, 133].

In this thesis we concentrate on one of the anti-*static* analysis variant of API hiding: **API hashing**. To the best of our knowledge, relatively little prior work specifically targets automated analysis of samples that employ hashing-based name encodings; most existing defenses and studies instead address dynamic anti-monitoring and anti-hook techniques. For the latter (anti-dynamic API-hiding methods) there exist several relevant countermeasures and analyses in the literature, to which we refer the interested reader [68, 67, 43].

API Hashing

In this work we concentrate on API hashing—an anti-*static* technique that replaces plaintext API and possibly DLL identifiers with numeric hash values so that import-related signals (which are highly informative for understanding program behaviour) are absent from the resulting binary. Several high-profile families (for example, *Conti*, *Netwalker* and *Emotet*) employ hashing-based name obfuscation to frustrate static inspection. The remainder of this section summarizes the technical background required for Chapter 4 where we present our first contribution.

When analysing untrusted Windows binaries, the Import Address Table (IAT) is a primary source of information: it reveals the APIs a program invokes, gives strong cues about its capabilities, and supports tasks such as malware clustering and threat-correlation. [90] To hide functionality and complicate static triage, malware authors frequently avoid explicit imports and instead resolve API addresses at runtime.

A straightforward dynamic resolution strategy is to call standard loader facilities such as `LoadLibrary` and `GetProcAddress`. [43] Because these calls are both well-known and easy to recognize, many samples take an additional step: they store only hash values of API and/or DLL names and perform a hashed lookup over the export tables of loaded modules. Concretely, the malware enumerates a module's export table, computes its custom hash function over each exported name, and compares the result to the stored target hashes. When a match is found, the malware reads the corresponding function pointer from the export table and uses it directly. Because the plaintext names are never embedded in the binary, this pattern defeats simple

static name matching and complicates trace-based attribution. [1]

To illustrate, consider KERNEL32.DLL, which exports `CreateFileW` among many other functions. Instead of importing `CreateFileW` via the IAT, a hashed-resolution routine will iterate the export names of KERNEL32.DLL, compute the configured hash for each name, and only when the computed value equals the precomputed target will it retrieve the address for `CreateFileW`. This procedure keeps API (and possibly DLL names) out of both the static disassembly and common string searches, significantly raising the bar for automated static and semi-automatic dynamic analyses.

A simplified implementation of this logic appears in Listing 3.

```
1  for (i = 0; i < NumberOfFunctions; i++) {
2      api_name = (char *)(base + AddressOfNames[i]);
3      curr_hash = HASHING_FUNCTION(api_name);
4
5      if (curr_hash == pre_cmptd_hash) {
6          api_addr = (PDWORD)(base + AddressOfFunctions
7              [AddressOfNameOrdinals[i]]);
8          return api_addr;
9      }
10 }
```

Listing 3. C-like pseudocode illustrating the API-hashing loop: iterate exported names, compute a digest $h_\theta(name)$, compare it against embedded constants, and resolve the matching export to an address.

In the snippet, the code enumerates a module's exported names and computes a digest for each (line 3). If the computed value equals a stored target hash (line 5), the routine resolves the corresponding function pointer from the export table (lines 5-6) and returns it. Line 2 shows how exported-name RVAs are converted into pointers by adding the module base; this use of relative virtual addresses (RVAs) to reference export-table fields is pervasive in real samples and forms an important, indicative artifact for static and dynamic recovery.

Two practical extensions of this pattern are common in the wild. First, the same hashed-lookup technique is frequently applied to DLL names as well as function names: the loader will enumerate loaded modules (for example, via the Process Environment Block) and hash each module name to find a matching library before scanning its exports. Second, many families use the resolved addresses to build a

[1] Instrumenting and recording every API invocation at runtime can produce a behavioral trace; however, such traces typically do not reveal the mapping from hash values to API names, the exact resolution points, or which APIs were obfuscated. In addition, anti-monitoring and anti-hooking techniques common in conjunction with API-hiding can prevent reliable interception. [68]

private, runtime import table (an IAT-like structure) that the malware populates at process startup; subsequent code then calls through these resolved pointers instead of invoking well-known import stubs. Both variants increase the difficulty of static analysis and of naïve runtime monitoring, since neither explicit import metadata nor plaintext symbols appear in the binary or in simple trace outputs.

Internally, resolution is preceded by locating the module that exports the obfuscated APIs. Once the module base address is obtained, the resolver walks the module's PE structures to enumerate exported symbols and compute their digests. Concretely, the routine typically performs the following steps:

1. Reads the DOS header (IMAGE_DOS_HEADER) at the module base and uses **e_lfanew** (offset 0x3C) to find the NT headers.
2. Follows **e_lfanew** to the NT headers (IMAGE_NT_HEADERS), which contain the **FileHeader** and the **OptionalHeader**.
3. Looks up the export data directory entry in **OptionalHeader.DataDirectory** at index IMAGE_DIRECTORY_ENTRY_EXPORT. Its VirtualAddress field contains the RVA of the export directory.[2]
4. Interprets that RVA as an IMAGE_EXPORT_DIRECTORY structure (the *Export Table*) [95] and walks its arrays.

The fields of IMAGE_EXPORT_DIRECTORY that are most relevant for API hashing are:

- **NumberOfFunctions**: the total number of entries in the export table, either by name or ordinal.
- **NumberOfNames**: the number of entries in the name pointer and ordinal table.
- **AddressOfFunctions**: RVA of a DWORD array with **NumberOfFunctions** elements. Each element holds an RVA to the function's entry point.
- **AddressOfNames**: RVA of a DWORD array with **NumberOfNames** elements. Each element is an RVA pointing to a null-terminated ASCII symbol name.
- **AddressOfNameOrdinals**: RVA of a WORD array pointing to the export ordinals of all the exported functions by the module.

[2] All table pointers in the export directory are RVAs and must be translated to virtual addresses (VAs) using the module's image base and section mapping.

A typical hashing loop proceeds by iterating over **AddressOfNames[i]**, reading the symbol name string, optionally normalizing it (*e.g.,* through case folding or according to an ANSI/Unicode policy), computing $h_\theta(name)$, and comparing the resulting value against a set of embedded hash constants. Upon a match, the code uses **AddressOfNameOrdinals[i]** to retrieve the function RVA from **AddressOfFunctions** and resolve the final address.

Chapter 4 shows that our approach exploits this layout during dynamic analysis: for each loaded module, the offsets of the relevant export-directory fields are fixed. We record these offsets at module-load time and subsequently recognize accesses to them in the instruction trace, allowing us to identify samples that (potentially) employ API hashing, as well as to extract the information required for deobfuscation and further analysis.

Countering API hashing remains a challenging task. Analysts typically rely on experience and the manual identification of known hashing algorithms. However, systematic documentation is scarce: aside from a few practitioner blog posts [114, 60], community efforts such as [103] offer precomputed hash tables for common APIs but struggle to keep pace with the evolution of malware. Even minor modifications to a hashing algorithm can render these resources obsolete. This variability—where readable API names are replaced with seemingly arbitrary hash values—hampers traditional reverse engineering workflows and highlights the pressing need for automated solutions capable of adapting to diverse and evolving hashing schemes.

A Real-World Example of API Hashing: Analysis of Conti Ransomware Leaked Source Code

Alzahrani *et al.* [4] provide a detailed analysis of the Conti ransomware source code, leaked in February 2022. Conti is a sophisticated ransomware-as-a-service (RaaS) that began operations in 2019 and has been responsible for high-profile attacks, particularly against healthcare providers, often demanding multi-million-dollar ransoms. The authors dissect the malware's functionalities to reveal advanced techniques for evading detection and executing malicious tasks. Their analysis focuses on the portion of the source code responsible for encryption and decomposes the ransomware's execution flow into six main phases: *API hashing*, API unhooking, mutex creation, shadow copy deletion, process termination, and multithreaded encryption.

API Hashing and Dynamic Loading. Rather than listing required libraries and API functions in the executable's import table—where they would be readily discoverable—Conti resolves them dynamically at runtime. This technique complicates analysis for reverse engineers and signature-based detectors (*e.g.,* EDR

systems), obscuring the malware's capabilities.

Execution begins in the *WinMain* function, which calls `InitializeApiModule`. The primary role of this function is to load KERNEL32.DLL, which exposes core functions such as `LoadLibraryA` and `GetProcAddress`. Using these two APIs, the ransomware can load additional libraries into its address space and obtain the addresses of any functions they export.

To conceal the API names it needs, Conti employs API hashing. The specific steps are as follows:

- **Hashing with the Murmur2A algorithm**: The `GetApiAddr` function implements the Murmur2A algorithm to compute hashes of the required API function names. Murmur2A is a non-cryptographic hash function designed for high throughput and good dispersion in hash-table lookups; it is not intended to provide cryptographic security but is well suited for fast name-to-hash mappings. The implementation present in the Conti source matches an open-source reference implementation hosted on GitHub [135], indicating the authors reused a standard implementation rather than a custom variant.

- **Runtime resolution**: During execution, Conti enumerates the exported symbols of a loaded library, applies Murmur2A to each symbol name, and compares the computed hash against precomputed hash constants embedded in the malware binary. When a computed hash equals one of the stored constants, the ransomware infers that the corresponding export is the desired API and records its memory address. This procedure allows Conti to obtain the address of any exported function without ever storing the function's plain-text name in the import table or as a static string in the executable.

This combination of dynamic loading and API hashing constitutes an effective obfuscation strategy: by avoiding explicit imports and by replacing readable API identifiers with opaque hash values, Conti significantly complicates static inspection techniques that rely on parsing the import table or scanning binaries for known API names. The paper further observes that although deobfuscation and emulation-based resolution techniques exist, they can be undermined by the presence of robust anti-analysis measures (for example, anti-virtual-machine and anti-debugging checks) that sophisticated ransomware families such as Conti variants commonly integrate. Consequently, analysts often resort to combined dynamic and static approaches—including careful instrumentation and environment-aware execution—to reliably recover the mapping between hash constants and API names.

2.2.3 Anti-Debugging Techniques

Although not the primary focus of our research, it is worth mentioning this category of anti-analysis techniques, as they are both commonly employed by malware authors and relatively easy to implement effectively.

Anti-debugging techniques enable a program to detect whether it is being executed under the control of a debugger. Debugging allows an analyst to step through code, modify memory or variables, and closely observe execution; when successful, it substantially simplifies understanding a sample's behavior and remains one of the most powerful tools for manual malware analysis. Malware authors therefore try to prevent or detect debugging.

The simplest and most widely known check involves using the Windows API `IsDebuggerPresent`, which indicates whether the calling process is being debugged, and related APIs provide similar information. Malware may also search for debugger-related windows or process names and compare them against known debugger identifiers.

From an analysis-tool perspective, APIs such as `IsDebuggerPresent` also illustrate a practical pitfall: they are frequently invoked early during process startup. Hooks that indiscriminately alter their behavior during initialization may prevent the application from loading at all. As a result, dynamic-analysis frameworks must apply these hooks in a context-aware way—for example, only after module initialization has completed or only for images loaded after startup—to avoid introducing artificial crashes or misclassifying benign samples.

More advanced techniques query the operating system for process-level information, using system calls such as `NtQueryInformationProcess` to obtain fields that reveal debugging status (for example, debugger-related flags or kernel objects associated with a debugged process).

Timing-based checks are another common class of defenses: debugging frequently introduces observable delays in instruction execution. A program can measure expected (native) timing—using instructions such as `rdtsc`—and compare it to observed timing; significant discrepancies can indicate the presence of a debugger or instrumentation.

A comprehensive treatment of anti-debugging methods, their implementations, and countermeasures is available in the literature; we refer interested readers to a detailed survey for further reading [34].

2.3 Program Analysis & Dynamic Binary Instrumentation

Program Analysis refers to the process of examining the behavior of computer programs with respect to properties such as correctness, robustness, safety, and liveness. It can be applied to detect bugs or security vulnerabilities, patch software, and optimize performance. Program analysis can be used statically, dynamically, or by combining both approaches in a hybrid manner. Since static analysis typically requires access to source code, which is not available in our setting, we resort to dynamic program analysis. This approach enables us to study the runtime behavior of malware and directly observe how a sample interacts with its environment.

2.3.1 Dynamic Program Analysis

As noted earlier, dynamic program analysis leverages runtime knowledge of the program to increase the precision of the analysis, at the cost of performance degradation due to runtime checks. The main techniques supported by dynamic program analysis are:

- *Software testing*: used to evaluate software quality and ensure that it behaves reliably as intended. Programs are executed with specific inputs, and their behavior and outputs are examined.
- *Monitoring*: focuses on recording and logging information about the program such as resource usage, events, and interactions.
- *Program slicing*: given a subset of a program's behavior, program slicing, a technique used historically [76, 14] reduces the program to the minimal form that still reproduces that behavior. The resulting reduced program, called a *slice*, is a faithful representation of the original program with respect to the specified behavior subset.

Beyond the broad areas of application described above, dynamic program analysis also enables a number of specific techniques that are widely used in security research and that rely on runtime instrumentation primitives provided by DBI frameworks. These techniques include fuzz testing, system-call interposition, dynamic taint analysis [101, 69], and dynamic program slicing, and many others, each of which leverages runtime visibility and control to expose program behavior that is difficult or impossible to obtain statically.

Fuzz Testing. Fuzzing generates large volumes of both well-formed and malformed inputs and executes the target program while monitoring for crashes, hangs, or

other anomalous behavior. The technique has been extensively studied in academia, applied across multiple domains, and significantly refined over the years [78, 49, 16]. When combined with dynamic instrumentation (*e.g.,* coverage-guided fuzzing), runtime feedback about executed basic blocks or traces can be leveraged to guide input generation toward unexplored program states, thereby greatly improving the effectiveness of vulnerability discovery.

System-Call Interposition. Interposing on system calls allows analysts to observe and mediate a program's interactions with the operating system (file I/O, networking, process control, etc.). Interposition can be implemented at different levels—*e.g.,* user-space hooks, LD_PRELOAD-style wrappers, PTRACE, or DBI-based API/syscall wrapping—and is useful both for detailed logging and for selectively modifying system-level behavior to reveal or constrain malware actions. The runtime behavior of a program can be characterized by the sequence of system calls it executes [15]; for this reason, as we show throughout this thesis, system-call interposition is a fundamental technique for our main contribution.

Dynamic Program Slicing. Dynamic slicing extracts the subset of executed instructions that affect a particular value or program point during a concrete run. This produces a compact *slice* that preserves the observed behavior for the chosen criterion, which helps focus manual analysis and debugging on the code that actually influenced a suspicious outcome.

All of the above techniques benefit from the low-level hooks and code-rewriting features offered by DBI frameworks: instrumentation APIs, event hooks for fragment creation, function-wrapping facilities, and code caches that enable efficient inspection and modification of hot code paths. In this thesis we describe how several of these techniques were adapted to the domain of malware analysis, with particular emphasis on instrumenting samples that implement anti-analysis mechanisms.

2.3.2 Dynamic Binary Instrumentation

In the context of malware analysis, developing efficient and reliable methods to monitor the behavior of samples during execution is crucial, as it enables researchers to study their actions with instruction-level granularity. This task is generally accomplished through instrumentation, that is, the insertion of additional instructions into a program to record its activity or modify its behavior for analysis purposes. Traditionally, debuggers have been the primary tools for dynamic program analysis, providing a straightforward way to instrument and observe programs at runtime. However, as previously discussed in 2.2.3, many malware families actively detect and evade debuggers, limiting their usefulness in hostile settings. To overcome these limitations, a strong alternative has emerged in the form of Dynamic Binary

Instrumentation (DBI) frameworks. These frameworks provide powerful, transparent mechanisms for runtime code manipulation and have become a popular foundation for both academic prototypes [24] and practical tools [69].

Instrumentation refers to the ability to monitor or measure a program's performance, diagnose errors, and record trace information. From a programmer's perspective, instrumentation is implemented through additional code instructions that monitor specific components of a system, effectively introducing extra operations into a program to support analysis and monitoring.

There are two main types of instrumentation: (1) source or static instrumentation, which can be applied to the source code or intermediate code—typically at compile time—and (2) dynamic instrumentation, which supports inserting probes and user-defined analysis routines into running software without requiring access to its source code or modifications to the runtime. Since our work focuses on malware, where source code is typically unavailable, we employ the second type of instrumentation.

At a higher level, a DBI system can be viewed as an application-level virtual machine that interprets the Instruction Set Architecture (ISA) of a given platform while offering instrumentation capabilities. This design allows analysis tools to observe and even alter instructions and data during execution. More concretely, a DBI system can be defined as a user-space execution runtime that provides process-level virtualization [40]. Its main goal is to observe—and, when necessary, modify—the architectural state of the target program under analysis, including register values, memory contents and addresses, as well as control-flow transfers.

To achieve this, most DBI systems rely on dynamic compilation. Instead of executing the original code directly, the program is analyzed, instrumented, and compiled using a just-in-time (JIT) compiler. The JIT compiler, as part of the runtime environment, translates code into native machine instructions at runtime, improving performance while enabling instrumentation. An instruction fetcher component reads the original program instructions as they are executed for the first time, allowing the DBI engine to instrument them before compilation. The resulting instrumented code is stored in a code cache, while a dispatcher coordinates execution between cached traces and new code fetches.

The fundamental compilation unit in DBI systems is typically a trace, defined as a straight-line sequence of instructions ending with an unconditional transfer and possibly including side exits for conditional branches. From the user's perspective, traces correspond to sequences of basic blocks (BBs) from the original program. A basic block is a sequence of instructions satisfying two properties:

- No instruction in the block (except the first, called the *leader*) can be the target of a jump.

- Only the last instruction may alter the control flow (*e.g.,* via a jump or branch).

In practice, the DBI engine continuously builds and manages these traces, rewriting and caching them on demand. Unlike traditional compilers such as GCC, this process does not aim to optimize source code, but rather to rewrite binaries as part of the instrumentation phase.

From the user's perspective, DBI frameworks expose general-purpose APIs that support a wide range of analyses. Through these APIs, users can develop clients (also called DBI tools) that execute alongside the target program. Such clients can perform diverse operations, such as overwriting register contents, replacing program instructions, modifying function arguments or return values, or even redirecting execution to user-defined functions. This flexibility is what makes DBI frameworks powerful tools for malware analysis and other security research tasks.

DBI Frameworks

Intel Pin [86] and DynamoRIO [27] are two of the most widely used Dynamic Binary Instrumentation (DBI) frameworks, particularly in the context of malware analysis, each supporting multiple architectures and operating systems. An alternative for cases where only API-level monitoring and instrumentation are required on the Windows platform is Microsoft Detours [92, 26]. Detours lacks the low-level capabilities offered by DBI frameworks but compensates with higher performance and lower overhead, making it well suited for building lightweight and reliable tracing tools. For this reason, it is also widely used in Windows malware analysis [44, 58].

Intel Pin, developed and maintained by Intel, supports the IA-32, x86-64, and MIC instruction set architectures and enables the creation of sophisticated dynamic program analysis tools. Well-known tools built on top of Pin include Intel VTune Amplifier, Intel Inspector, and the Intel Software Development Emulator. A common criticism of Pin, however, is its closed-source nature, which limits extensibility and transparency.

In contrast, DynamoRIO is an open-source framework that provides full visibility into the instruction stream, allowing researchers to perform fine-grained and low-level code transformations. Its relative performance compared to Pin remains a topic of discussion within the DBI community. For this thesis, DynamoRIO was chosen as the underlying DBI framework on which both of our research prototypes were developed.

DynamoRIO operates as a user-mode process virtual machine that transparently interposes between an application and the operating system, enabling tools to observe and modify execution without requiring changes to the application, operating system, or hardware, as illustrated in Figure 2.1.

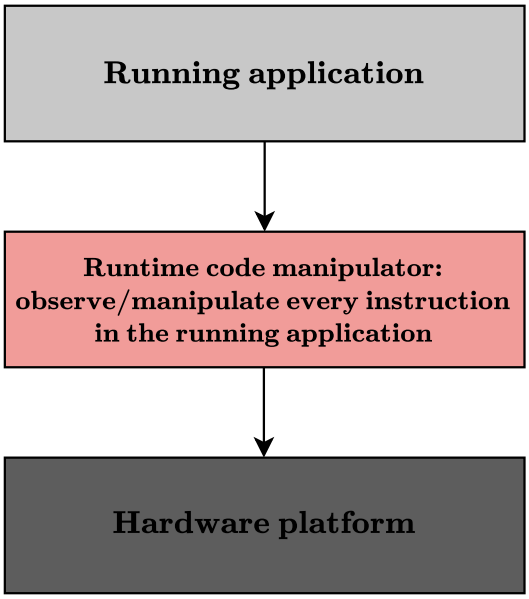


Figure 2.1. DynamoRIO is a user-mode dynamic binary instrumentation framework that interposes on an application's code stream to provide a runtime control point for custom analysis tools, without modifying the underlying operating system, hardware, or the application itself. Tools built on DynamoRIO operate unmodified on both native systems and virtual machine guests.

At runtime, DynamoRIO redirects program execution into a writable code cache, where application code is copied one dynamic basic block at a time, instrumented or rewritten using its instruction manipulation library, and then emitted as native code through a just-in-time compilation pipeline. Blocks that branch to already cached code are directly linked to reduce dispatcher overhead, while frequently executed block sequences are combined into traces (or fragments) stored in a hot code cache. Both blocks and traces exhibit a linear control-flow structure with a single entry point and one or more exits, simplifying instrumentation and analysis.

Clients interact with DynamoRIO through event hooks—callbacks for events such as module load or unload, basic block or trace creation, and system calls—allowing them to inspect or modify fragments before execution. Higher-level services, such as function wrapping and replacement (*e.g.,* `drwrap_wrap_ex`, `drwrap_replace`), enable clients to register pre- and post-function callbacks to examine or alter function arguments and return values. This capability is particularly useful for intercepting library calls or bypassing API-based evasions. By retaining full control over execution, efficiently managing code caching and linking, and exposing rich low-level instruction APIs, DynamoRIO supports fine-grained, high-performance dynamic instrumentation across IA-32, AMD64, ARM, and AArch64 architectures on mainstream operating systems, while remaining equally functional within virtualized environments.

Anti DBI

As with the other technologies and techniques discussed so far, Dynamic Binary Instrumentation (DBI) inevitably introduces execution artifacts that a determined adversary can detect and exploit to evade analysis [40]. Given that one of DBI's principal applications is security research, concerns about these artifacts and corresponding anti-DBI techniques have naturally emerged. Both practitioner reports [48, 80] and academic studies [109, 71] have investigated this problem.

Anti-DBI techniques exploit the imperfect transparency of DBI engines. The authors of [40] discuss runtime transparency for program analysis under two complementary interpretations, based on a survey of the state of the art. First, transparency can be understood as requiring that an application executes as if running natively, without instrumentation, and that interoperability with native applications remains unaffected. Second, transparency can be interpreted adversarially: the system must resist attempts by a program to probe for the presence of a DBI engine and thereby disrupt the analysis. While desirable properties for both interpretations are defined, achieving full transparency in practice is extremely challenging, leaving DBI frameworks susceptible to detection.

This imperfect transparency has led researchers to design detection mechanisms capable of revealing the presence of DBI frameworks during dynamic analysis. Two principal scenarios may result:

- *DBI evasion*: occurs when the instruction or data flows of a program diverge from native execution. This divergence may result either because the code explicitly detects the presence of a DBI engine and alters its behavior, or because of translation defects within the DBI system itself.

- *DBI escape*: occurs when parts of a program's instruction or data flows execute natively on the CPU outside the DBI engine's control, either in the instrumented process's address space or in another process's address space.

Building on this distinction, the authors of [40] proposed a taxonomy of known techniques used to detect or exploit DBI. Representative vectors include:

- *Time overhead.* Translating and instrumenting original instructions into cached traces inevitably introduces runtime slowdowns. Analogous to "red pill" hypervisor-detection techniques (*e.g.,* timing `cpuid`), an adversary can perform fine-grained timing measurements and compare the execution time of a code fragment against a baseline on a reference (native) system. Deviations—such as delays when dynamically loading a library—can indicate the presence of a DBI engine.

- *Leaked code pointers.* A cornerstone of DBI transparency is decoupling the real instruction pointer (IP) from the virtual one exposed to the program. Nevertheless, adversaries can sometimes leak the real IP and compare it against expected values; in 32-bit Windows, for example, the `int 2e` instruction has been used to expose such discrepancies.

- *Memory contents and permissions.* Because DBI engines typically share the same address space as the analyzed program without strong isolation boundaries, memory inspection may reveal anomalies. Examples include unexpected memory sections, exported functions belonging to the DBI runtime, duplicated command-line arguments, or unusual memory-usage patterns—each of which may betray the presence of a DBI framework.

- *DBI engine internals.* Although DBI abstractions mask most CPU context, some runtime bookkeeping remains observable. For instance, Thread Local Storage (TLS) slots—commonly used by DBI engines for internal state—can provide a detection vector when inspected by the program.

- *Interactions with the OS.* Because DBI operates in user space, interactions with the operating system may reveal its presence. Enumerating processes or inspecting parent/child relationships can expose the DBI framework (*e.g.,* Intel Pin or DynamoRIO).

- *Exception handling.* To remain transparent, DBI engines must correctly mediate exceptional control flow and return accurate context for Structured Exception Handling (SEH). Implementations may fail in corner cases; for example, Intel Pin has been shown to mishandle single-step exceptions and `int 2d` instructions—both commonly leveraged by evasive malware—resulting in observable discrepancies from native behavior.

- *Translation defects.* Binary translation is inherently complex and subject to implementation trade-offs. Some instructions may be unimplemented (*e.g.,* `enter` in Valgrind [100]), while others function only under restricted conditions (*e.g.,* far returns in Pin may work only within the same code segment). Self-modifying code (SMC) poses another challenge: cache invalidation should follow SMC, but some engines fail to detect it reliably, breaking transparency.

Regarding DBI *escaping*, the first publicly described attack was reported by Cosmin Gorgovan [57]. One exploitation vector leverages the fact that code-cache locations may remain readable and writable by the instrumented program. In this attack, the adversary encodes a unique byte pattern into an executed code block, locates the pattern in the code cache (using OS memory queries to discover

its randomized address), and overwrites it with a trampoline. On subsequent executions, control transfers to native code outside the DBI engine; the attack also details mechanisms to return execution to the DBI system gracefully. The code cache is not the only target for escapes—strategies may also tamper with DBI-specific internal data structures, callbacks, or stack regions [126].

2.4 Fuzzing and Coverage-Guided Exploration

First introduced in the early 1990s, fuzzing has remained one of the most widely deployed techniques for discovering software security vulnerabilities. At a high level, fuzzing denotes the automated, repeated execution of a program under test (PUT) with many automatically generated inputs (testcases) that are intentionally malformed—syntactically or semantically—to trigger unexpected behavior such as crashes, memory-safety violations, or logic errors [91].

2.4.1 Fuzzing Fundamentals

Fuzzers operate through iterative cycles of input generation, execution, and feedback collection. Testcases that yield new code paths are retained as seeds for further mutation. Mutation strategies include bit flips, arithmetic modifications, and structured splicing. Coverage is typically measured using edge or basic-block transitions.

Figure 2.2 shows a compact view of the typical fuzzing pipeline.

A fuzz campaign is conventionally decomposed into a few stages:

1. **Preprocess / setup:** instrumenting the PUT (compile-time or binary-level), preparing drivers or harnesses, collecting an initial corpus of seeds, and optionally trimming or computing a minset. This step reduces redundant inputs and prepares the PUT for high-throughput runs.

2. **Testcase generation (INPUTGEN):** producing concrete testcases from seeds, models, or grammars using either generation-based (model-driven) or mutation-based approaches. Generation-based fuzzers require explicit input models (encoders/grammars); mutation-based fuzzers apply mutation primitives to seeds.

3. **Testcase execution and monitoring (INPUTEVAL):** running the PUT on each testcase while monitoring execution for coverage and for security violations using crash detection and sanitizers [119]. Grey-box fuzzers collect lightweight feedback, such as coverage bitmaps, to guide future generation.

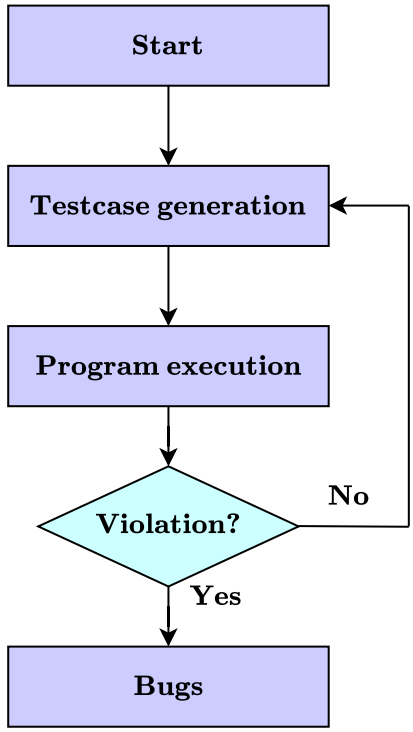


Figure 2.2. Overview of the fuzz testing process, illustrating the iterative cycle of test case generation, program execution, and violation detection leading to bug discovery.

4. **Analysis and update (CONFUPDATE):** triaging crashes, deduplicating and minimizing inputs, and updating the seed pool by adding interesting testcases that increase coverage or exercise new behaviors. The updated seed pool is then scheduled for further fuzzing iterations.

A simple coverage-driven fuzzing loop therefore repeatedly selects a seed, assigns it some *energy* (*i.e.,* a number of mutation attempts), generates mutated testcases, executes them, and retains those that crash or are "interesting" (produce new coverage) for further mutation; this loop continues until time or resource limits are reached.

Fuzzers can be divided by input-generation strategy:

- **Generation-based generators** rely on an explicit model or grammar of valid inputs (for example, an encoder or a format specification). When available, these generators produce well-formed inputs that more easily pass parsing and exercise deeper program logic. However, constructing models can be expensive or infeasible for complex formats.

- **Mutation-based generators** start from one or more valid *seeds* and produce testcases by mutating bytes or fields (bit/byte flips, arithmetic changes, splicing,

dictionary insertions, etc.). Mutation-based fuzzers are easy to start and widely used, but their effectiveness depends strongly on the quality of the initial seed corpus.

Because mutation-based fuzzers rely on seeds, managing the seed corpus is a crucial practical problem:

- **Common seed sources:** seeds are typically collected from standard benchmarks, crawled real-world inputs, canonical example files shipped with software, or existing proof-of-concept (PoC) files. Crawling often yields diverse and realistic inputs for many file formats.
- **Seed trimming and minset:** large seed collections waste time in initial dry runs; many fuzzers therefore perform *seed trimming* (removing bytes that do not affect coverage) and compute a *minset*—a minimal subset of seeds that together achieve the same coverage metric—to reduce duplication and improve throughput. AFL and other tools provide utilities to compute such minsets.
- **Seed selection and scheduling:** during fuzzing, a scheduler chooses which seed to fuzz next and how much energy to allocate. Heuristics include favoring small and fast seeds, prioritizing seeds that exercise rare or deep paths, or using directed strategies [22, 21]. Effective selection policies can substantially improve bug-finding efficiency.

A common classification of fuzzers is by the amount of internal information they use:

- **Black-box fuzzers** only observe I/O behavior and do not instrument the PUT. They are simple but typically less efficient at exploring deep logic [99].
- **Grey-box fuzzers** (the dominant practical family) collect lightweight dynamic feedback such as code-coverage bitmaps to guide mutation (*e.g.,* AFL, libFuzzer, honggfuzz [54, 55, 84]). Grey-box fuzzing balances speed and informed exploration.
- **White-box fuzzers** use heavy program analysis (concolic/symbolic execution, taint analysis) to systematically reason about path constraints in the program's source code. They can solve difficult checks but are more expensive and suffer from path explosion. Hybrid approaches combine white- and grey-box techniques to gain the advantages of both approaches [53].

Fuzzers typically rely on simple oracles (process crash, signals) and augment detection with sanitizers to discover memory errors and undefined behavior. After discovery, crash triage and deduplication (stack hashing, coverage-based grouping) are used to prioritize and minimize results for manual analysis or automated pipelines.

2.4.2 Coverage-Guided Fuzzing

Coverage-guided fuzzing is the strategy underlying most state-of-the-art fuzzers [54, 56], and has proven to be both effective and efficient. The guiding principle is that, to discover deep bugs, fuzzers should aim to explore as many distinct program execution states as possible. Unfortunately, there is no simple or direct metric for program states, since program behavior can be highly complex and nondeterministic. An ideal metric must therefore be easy to obtain at runtime. For this reason, code coverage is widely adopted as an approximation: it measures which parts of a program are executed during a test run, and higher coverage typically correlates with the exploration of new program states.

Programs are naturally decomposed into basic blocks—straight-line code fragments with a single entry and exit point. Instructions within a basic block always execute sequentially, and control flow only transfers at the end. Basic blocks provide a convenient granularity for measuring coverage because they represent the smallest coherent execution units: measuring functions is too coarse, while measuring individual instructions is often redundant. Moreover, basic blocks can be efficiently identified at runtime (*e.g.,* by their entry address) and easily tracked through instrumentation.

The first approach is simple but loses path information. For example, if the program executes path BB1→BB2→BB3→BB4 and later path BB1→BB2→BB4, the same set of blocks is recorded, missing the fact that a new transition BB2→BB4 was exercised. Edge coverage, by recording transitions, avoids this loss and provides finer-grained feedback, as illustrated in Figure 2.3.

American Fuzzy Lop (AFL) [54] was the first practical fuzzer to adopt edge coverage as its core feedback metric. AFL implements this via lightweight compile-time or binary-level instrumentation. Each basic block is assigned a random identifier at compile time (CUR_LOCATION), and a global bitmap of fixed size (64 KB) is shared between the fuzzer and the instrumented program. On every transition between two basic blocks, AFL computes an index as the `xor` of the current block's ID and the previous block's ID (shifted for mixing). The corresponding byte in the bitmap is then incremented. After updating, PREV_LOCATION is set for the next transition. This scheme probabilistically maps edges to bitmap entries and allows the fuzzer to detect when a mutated input exercises a previously unseen edge.

The canonical AFL instrumentation appears as follows in Listing 4. Here:

- CUR_LOCATION is a constant random ID assigned to the current basic block at compile time.
- PREV_LOCATION stores the ID of the previously executed block (shifted right

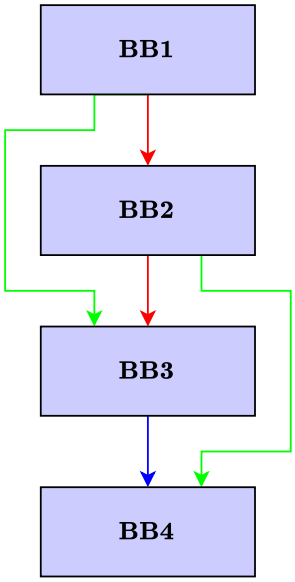


Figure 2.3. Example illustrating basic-block counting versus edge coverage. The transition from BB2 to BB4 would be missed when using block coverage alone.

by one to improve distribution).

- The `xor` expression CUR_LOCATION ^ PREV_LOCATION uniquely (though not perfectly, due to collisions) represents the edge taken.
- The shared bitmap entry is incremented, encoding both the presence and an approximate execution frequency of that edge.

```
/* Location IDs are assigned at compile-time */
cur_location = <COMPILE_TIME_RANDOM_ID>;

/* Compute edge ID and update coverage bitmap */
shared_mem[cur_location ^ prev_location]++;

/* Update previous location */
prev_location = cur_location >> 1;
```

Listing 4. AFL edge coverage instrumentation snippet

Any input that produces a previously unseen bitmap entry (*i.e.,* exercises a new edge) is marked as interesting and added to the fuzzer's seed queue for further mutation. This compact and efficient mechanism forms the foundation of AFL's effectiveness and has since become the standard design pattern for coverage-guided fuzzing.

Coverage-guided fuzzing optimizes exploration by correlating input mutations with coverage gains. Because the problem of multi-path exploration in malware shares conceptual similarities with fuzz testing, we applied this technique to malware analysis. In PFuzzer's context, instead of mutating program inputs, environmental properties (*e.g.,* registry keys, system calls) are mutated. The same feedback principle applies: whenever new behavior or coverage is observed, the corresponding environmental configuration is preserved and evolved.

The next chapter examines how these foundational techniques have been leveraged to analyze malware equipped with anti-analysis defenses, and discusses existing research gaps that motivated our contributions.

Chapter 3

State of the Art

Understanding the current state of malware analysis and the corresponding anti-analysis countermeasures requires an overview of the major research directions and limitations that motivate this thesis. This chapter surveys prior work in four main areas: (1) environment-sensitive malware and evasion detection, (2) multi-path exploration and dynamic analysis frameworks, (3) fuzzing-based approaches for behavioral discovery, and (4) API hiding.

Figure 3.1 presents a chronological timeline of the key studies that form the foundation and related context for our main contribution, PFuzzer. We begin by examining pioneering research on malware whose behavior is influenced by its surrounding environment. Next, we discuss approaches that employ alternative analysis techniques related to our solution, and we conclude with an overview of studies that first introduced fuzzing techniques into the field of malware analysis.

Finally, we provide a summary of the literature on API hiding mechanisms in malware, with particular emphasis on API hashing—the specific API hiding technique explored in our second contribution.

3.1 Analyzing Environment-Sensitive Malware

In this section, we review research efforts spanning over nearly two decades, highlighting how this problem remains highly relevant today. The phenomenon of malware adapting its behavior to the execution environment is well recognized. Some of the earliest influential works tackled this challenge by developing analysis platforms designed to be resistant—or invisible—to detection by malware. For example, Cobra [136] introduced the notion of *stealth* or *transparency* in malware analysis, while Ether [45] leveraged hardware virtualization to conceal the analysis framework from environment checks.

A second line of research approached the issue from a different angle, focusing

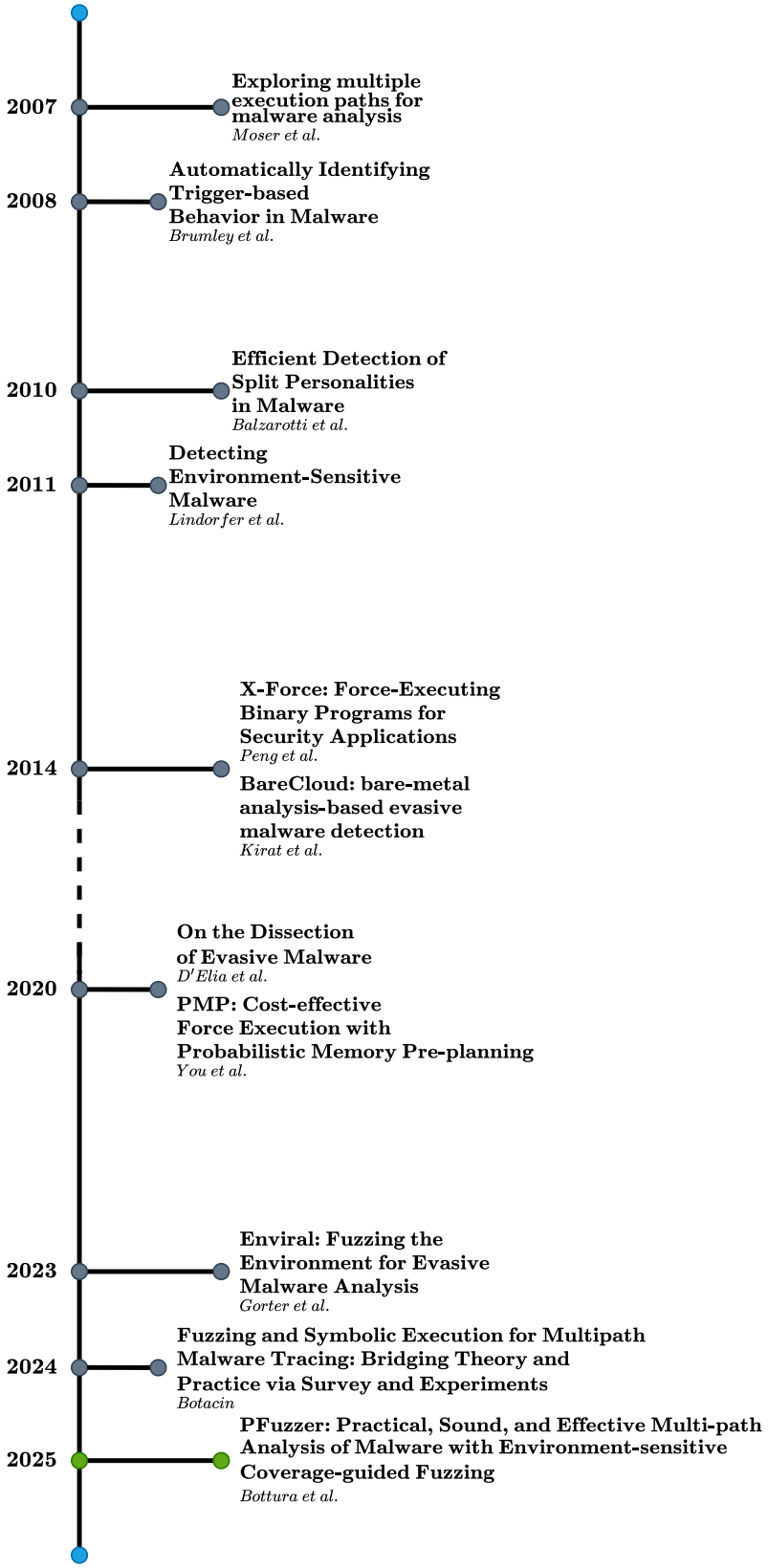


Figure 3.1. Timeline for environment-sensitive malware state-of-the-art overviewed.

on detecting when a malware sample exhibits divergent behaviors across multiple environments [65].

These two research directions laid the groundwork for a wide range of subsequent advances. In the remainder of this section, we review the most relevant efforts, particularly those closely related to our main contribution.

Efficient Detection of Split Personalities in Malware

Balzarotti *et al.* [15] address *split-personality* malware, *i.e.,* specimens that change behavior when executed in an analysis environment rather than on a native host. Their key idea is to enforce *input equivalence* between a reference run and a sandbox run by *recording* OS-level interactions during the former and *replaying* them during the latter, so that behavioral divergence becomes evidence of environment detection rather than incidental OS differences.

Analysis Methodology. The system first executes the sample on a clean reference host and records system-call interactions (including arguments and return values) via a kernel-level component. The sample is then executed in an Anubis-based sandbox [17] that replays the recorded interactions instead of consulting the live OS, aiming to make the two runs execution-equivalent at the system-call interface. Under this controlled replay, deviations between the reference trace and the sandbox trace indicate split-personality behavior.

The evaluation shows record/replay reproduces system-call interactions reliably with low overhead, and flags divergence when anti-VM checks (*e.g.,* Red Pill-style probes) are present. It also detects divergence on real-world samples known to disrupt emulator-based analysis, while remaining faster than fine-grained transparency frameworks.

Key Takeaway. The main value of this approach is that it provides an efficient and scalable *detection* signal for environment-sensitive malware without relying on heavy-weight transparency mechanisms.

Limitations. As acknowledged by the authors, the technique does not cover important practical cases such as multi-process malware, and it can be weakened by strategies that postpone checks beyond the observation window or distribute behavior across temporally distant interactions. More generally, this line of work follows a *transparency/equivalence* philosophy: it aims to neutralize environmental differences to *flag* divergence, but it does not directly provide a mechanism to *discover and characterize* multiple alternative behaviors once a specimen is identified as environment-sensitive.

Detecting Environment-Sensitive Malware

Lindorfer *et al.* [81] propose DISARM, a *multi-environment divergence* detector for environment-sensitive malware. The core idea is to execute each specimen across heterogeneous sandboxes and then quantify whether the observed behavior differs *more across sandboxes than within repeated runs of the same sandbox*, thereby separating evasion-induced divergence from natural run-to-run variability.

Analysis Methodology. DISARM collects behavioral profiles from multiple executions performed in different analysis environments (both with different Windows images and different monitoring technologies, including an out-of-the-box emulator and an in-the-box system-call monitor). It then normalizes the profiles to remove differences caused by benign environment artifacts (*e.g.,* user- and installation-specific identifiers, randomization, and repetitive enumerations), and compares normalized profiles using a set-based distance (Jaccard distance). To avoid flagging benign nondeterminism as evasion, DISARM computes an evasion score that contrasts inter-sandbox distances with intra-sandbox variability (the "diameter" of a sandbox), and classifies samples as environment-sensitive only when cross-sandbox divergence exceeds what is observed within a sandbox.

Results indicate that comparing inter-sandbox divergence against intra-sandbox variability provides a practical screening signal for environment-sensitive behavior at scale. The evaluation supports that normalization plus distance-based scoring can separate "natural" run-to-run changes from stronger cross-environment deviations.

Key Takeaway. The key contribution is a scalable, monitoring-agnostic way to *screen* large malware streams for analysis evasion by turning behavioral divergence across environments into a measurable signal and by explicitly accounting for baseline variability.

Limitations. As a divergence-based detector, DISARM can fail when a sample successfully suppresses malicious behavior in *all* tested sandboxes, especially if the environments share common artifacts (*e.g.,* network restrictions or similar performance overheads that enable timing-based checks). Moreover, since the evasion score is computed from the same observed traces the attacker can influence, a determined adversary can attempt to manipulate the score (*e.g.,* by inflating intra-sandbox variability or crafting behaviors that appear similar across sandboxes), reducing detection accuracy.

BareCloud: Bare-Metal Analysis-Based Evasive Malware Detection

Kirat *et al.* [70] introduce BARECLOUD, a *bare-metal* analysis system for detecting evasive malware by maximizing execution transparency and then measuring behav-

ioral divergence across heterogeneous analysis environments. The key insight is that some specimens will expose malicious activity only when executed on physical hardware, while suppressing it under emulation/virtualization-based sandboxes; BareCloud turns this discrepancy into a detection signal.

Analysis Methodology. BareCloud executes each sample both on bare metal and on multiple popular sandboxing setups (representing distinct execution/monitoring models) and then compares the resulting behavioral profiles. To preserve transparency on bare metal, the approach relies on coarse but robust observations (persistent system changes and network activity) rather than fine-grained in-guest tracing, and it uses normalization to reduce spurious differences across environments. For comparison, BareCloud proposes a hierarchical similarity model that matches behaviors at multiple abstraction levels (*e.g.,* object type/name/operation), addressing limitations of purely set-based comparisons when artifacts such as randomization perturb low-level identifiers.

The evaluation reports that hierarchical comparison improves over simple set-similarity baselines, and that large-scale deployment can flag many evasive samples automatically. A key observed outcome is that some specimens exhibit malicious behavior only on bare metal, remaining quiet in emulated/virtualized analyzers

Key Takeaway. The main contribution is showing that bare-metal execution combined with cross-environment comparison can reveal evasive behavior that remains hidden even in "high-transparency" virtualized analyzers. In the authors' deployment, BareCloud analyzed a large real-world stream and flagged a substantial subset as evasive, with many samples exhibiting malicious activity only on bare metal.

Limitations. Like other dynamic-analysis-based detectors, BareCloud remains vulnerable to stalling behaviors that delay malicious actions beyond the analysis window, and the lack of fine-grained monitoring on bare metal limits mitigation options. Despite increased transparency, bare metal can still be fingerprinted (*e.g.,* through hardware characteristics) and can remain suspicious due to missing realistic user activity and constrained/atypical networking, which can itself trigger evasive branches

3.2 Related Approaches

In this section, we review prior research closely related to our main contribution. Broadly, existing solutions can be grouped into two categories:

- approaches that aim to construct increasingly transparent analysis environments by eliminating artifacts that could reveal their presence to evasive

malware;

- approaches that focus on understanding how a sample's execution interacts with environmental information and on enforcing alternative decision outcomes.

We begin by examining the first class of solutions—those designed to improve the transparency of analysis environments.

3.2.1 Approaches that Builds Transparent Environments

On the Dissection of Evasive Malware

D'Elia *et al.* [41] introduced BluePill, a human-centered framework that bridges automated sandboxing and manual reverse engineering of evasive malware by offering a stealthy yet customizable analysis environment. Rather than fully automating evasion handling, BluePill emphasizes analyst-driven dissection: it helps neutralize anti-analysis checks so that malicious logic can be observed and debugged under controlled conditions.

Analysis Methodology. BluePill is built on Dynamic Binary Instrumentation (DBI) using Intel Pin [86], enabling fine-grained observation and manipulation of the target process while avoiding the semantic reconstruction overhead typical of VMI-based monitoring. Since DBI introduces detectable artifacts, BluePill includes an anti-evasion layer that mitigates common fingerprinting vectors against the instrumentation engine (*e.g.,* timing and memory/layout anomalies). At a high level, BluePill follows an *observe-check-replace* workflow: it intercepts suspicious queries, compares outcomes against an analyst-defined reference environment, and substitutes safer answers when needed to preserve stealth. The framework also provides layered support for platform simulation (hiding analysis artifacts), stealthy introspection primitives (*e.g.,* breakpoints/patching), and extensibility via new hooks for emerging checks.

Results include both automated testing on a large malware set and a small user study, indicating that the framework can neutralize multiple anti-analysis techniques and enable dissection of difficult specimens. Participants reported reduced effort spent "fighting evasion", focusing instead on understanding functionality such as unpacking and self-modifying logic.

Key Takeaway. BluePill represents a method to provide transparency during the analysis of malicious code bypassing evasive queries on-demand, while being easy and practical to be configured and extended when needed by the analyst.

Limitations. While it proved to be effective by the evaluation results provided by the authors, BluePill is limited by the fact that requires frequent manual work to

analyze properly malware samples in a general manner. A more detailed discussion on the limitations is given in Chapter 5 since we selected BluePill as one of the competitors solution to test against our approach [25].

3.2.2 Approaches that Studies the Interactions with the Environment

X-force: Force-Executing Binary Programs for Security Applications

Peng *et al.* [107] present X-Force, a binary-execution engine that prioritizes broad dynamic exploration by *forcing* execution past missing inputs and environment dependencies, targeting difficult settings such as packed/obfuscated and multi-stage binaries. Rather than attempting full path feasibility (as in symbolic execution), X-Force embraces a pragmatic trade-off: explore more code concretely, even if some explored paths may be infeasible in a real execution.

Analysis Methodology. X-Force instruments binaries using Intel Pin [86] as the DBI framework, and implements a crash-free forced execution model that tolerates faults (*e.g.,* via on-demand memory allocation and pointer correction) to keep execution progressing. Path exploration is guided by a worklist algorithm that selectively overrides a limited number of predicates or jump-table decisions per run, and uses taint tracking to focus forcing on decisions influenced by external inputs. To improve robustness on real malware, the system also applies practical stabilization, wrapping I/O and memory routines and constraining concurrency effects so that traces remain usable for downstream analyses.

Across case studies, results show increased executed code and improved recovery of CFG/CG information compared to standard dynamic runs, and the ability to expose guarded malware behaviors missed by conventional sandboxes. The evaluation also reports strong coverage/accuracy figures for type-reconstruction when combined with its reward-driven exploration

Key Takeaway. The key contribution of X-Force is establishing forced execution as a *practical middle ground* between incomplete single-run sandboxing and expensive symbolic reasoning, enabling concrete exploration to reach code regions that are otherwise guarded by missing triggers or complex runtime conditions.

Limitations. Forced execution is inherently *unsound*: explored behaviors are not guaranteed to correspond to feasible real-world executions, which complicates behavioral attribution and can introduce analysis artifacts. The crash-free machinery (fault tolerance, memory/pointer correction) can add overhead and distort program state, potentially deviating from authentic execution and affecting the fidelity of the observed behavior. While more scalable than symbolic execution, the approach can

still become expensive when exploring many branches in large or heavily obfuscated binaries.

PMP: Cost-Effective Forced Execution with Probabilistic Memory Pre-Planning

You *et al.* [142] propose PMP, a lightweight forced-execution framework that targets trigger-guarded or environment-dependent behaviors while reducing the heavy-weight fault handling typical of earlier forced-execution engines. The core idea is to turn crash avoidance into a probabilistic design problem: instead of tracking and repairing invalid pointers precisely, PMP pre-plans memory so that many invalid accesses are redirected into a controlled region with high probability.

Analysis Methodology. PMP pre-allocates a low-address memory area (PAMA) and initializes it to act as a reservoir for unintended dereferences, aiming to keep execution progressing without pervasive pointer tracking. To improve stability across common failure modes, the system applies lightweight initialization strategies across global state, heap allocations, and stack frames, and resorts to minimal demand-driven fixes only when repeated failures arise. Exploration is organized as repeated executions under randomized configurations, retaining outputs that are consistent across runs to mitigate probabilistic errors.

Benchmark results show high-throughput forced execution (many runs per second) and large speedups over earlier forced-execution designs while preserving useful behavioral reach. Malware experiments report that PMP uncovers more behavioral artifacts and deeper paths than sandbox-only baselines.

Key Takeaway. PMP demonstrates that forced execution can be engineered as a high-throughput, "good-enough" exploration technique by trading strict correctness for speed, while still yielding useful behavioral reach in practice.

Limitations. Despite its efficiency-oriented design, PMP inherits two fundamental constraints of forced execution: (i) some explored behaviors may be infeasible in genuine runs (unsoundness), and (ii) exploring many branches can still become costly as path counts grow.

Exploring Multiple Execution Paths for Malware Analysis

Moser *et al.* [98] propose a multipath dynamic analysis system that goes beyond single-trace sandboxing by using dynamic taint tracking to identify environment- and input-dependent branches and then systematically exploring alternative outcomes, extending the automated analysis framework of Bayer *et al.* [17]. The prototype system runs within a QEMU-based Windows virtual machine and produces substan-

tially richer behavioral reports than single-run sandboxes with snapshot-and-restore path exploration driven by taint-derived constraints.

Analysis Methodology. The system taints bytes originating from external sources (*e.g.,* file/network/registry/time results) and propagates taint through execution to detect conditional branches whose predicates depend on tainted data. When such a tainted branch is reached, the analyzer records it as a decision point, snapshots the process state, and later restores execution to force the opposite branch by rewriting the relevant input bytes using a constraint-solving step. To preserve kernel-visible resources (*e.g.,* file handles) across restores, the analyzer intercepts and neutralizes relevant system calls so that resources remain valid within the guest OS after restoration. This design enables targeted multipath exploration while keeping the analysis grounded in observed dependencies rather than brute-forcing it.

Results show environment/input-dependent control flow is common in the tested malware set and that taint-guided branch flipping yields large coverage increases for many samples. The evaluation also reports practical snapshot sizes/restoration times and that a majority of samples complete within imposed time limits.

Key Takeaway. The key contribution is demonstrating that taint-guided snapshots can make multipath exploration feasible for malware analysis: instead of hoping a sandbox run hits the right triggers, the analysis identifies which environmental inputs matter and actively drives execution toward alternate behaviors. This is a conceptually "more semantics-driven" alternative to forced execution, because it attempts to preserve feasibility by solving for inputs that satisfy observed branch conditions rather than bypassing failures indiscriminately

Limitations. Despite being more principled than forced execution, taint-driven exploration is still neither fully sound nor complete in practice: taint propagation and constraint solving can be imprecise, and complex transformations (*e.g.,* non-linear operations, hashing, table lookups) can break the ability to invert conditions reliably. Dynamic taint tracking is also a known target for dedicated evasion strategies designed to mislead or neutralize information-flow tracking, which limits robustness against adversarial samples [30].

Automatically Identifying Trigger-Based Behavior in Malware

Brumley *et al.* [28] propose MineSweeper, an automated framework for discovering *trigger-based* malicious behavior (*i.e.,* functionality activated only when specific trigger conditions hold). Compared to taint-driven snapshotting, their approach is solver-centric: it treats trigger sources symbolically and uses mixed concrete and symbolic execution to derive and solve path conditions that characterize the trigger precisely.

Analysis Methodology. Given a program and a set of trigger types of interest (*e.g.,* time, system events, network inputs), MineSweeper executes the binary while representing the selected trigger inputs symbolically and the remaining state concretely to reduce constraint complexity. Whenever execution reaches a conditional that depends on symbolic trigger data, the system forks the path condition, checks satisfiability with a constraint solver, and—when satisfiable—obtains concrete trigger values that drive execution down the corresponding branch. These solver-derived trigger values are then used to re-run the malware concretely to observe the activated behavior under controlled conditions.

The evaluation on real malware (*e.g.,* worms and a keylogger) demonstrates automated detection of trigger-based behavior and generation of concrete trigger values to activate hidden functionality. Reported outcomes suggest substantial time savings versus manual reverse engineering for the studied cases.

Key Takeaway. The key contribution is showing that (for certain trigger classes) symbolic reasoning can move trigger discovery from "manual reversing" to an automated loop that *both* identifies the trigger condition *and* produces concrete witness inputs that activate the hidden behavior. This is also one of the earliest points in the SOTA where "guessing trigger inputs" is explicitly discussed and rejected as impractical in general—motivating later, more feedback-driven exploration strategies.

Limitations. Symbolic (and concolic) approaches face fundamental scalability constraints on real malware, most notably *path explosion* as branching grows, and heavy dependence on accurate semantic modeling for instructions, libraries, and system interactions [23]. In adversarial settings, these assumptions can be stressed or intentionally targeted, and the analysis can degrade or fail when constraints become too complex or when execution relies on unsupported semantics [30, 105].

3.3 Works Introducing Fuzzing for Malware Analysis

In this section, we review works in the state of the art that introduced or formalized the use of fuzzing techniques for malware analysis. We first examine the study by Botacin [23], which systematically evaluated multiple approaches for exploring different execution paths in malware (including fuzzing, symbolic, and concolic execution). We then discuss the work of Gorter *et al.* [58], who proposed an automatic evasive malware analysis framework that combines existing methodologies with fuzzing-based techniques to bypass anti-analysis checks. Conceptually, their framework lies between the human-centered transparency approach of D'Elia *et al.* [41] and the methodology presented in our main contribution [24].

Fuzzing and Symbolic Execution for Multipath Malware Tracing: Bridging Theory and Practice via Survey and Experiments

Botacin [23] conducted a systematization of knowledge (SoK) and large-scale experimental comparison of multipath malware-tracing techniques, including fuzzing, coverage-guided fuzzing, forced execution, pure symbolic execution, and concolic execution. The author reimplemented representative tools for each category and executed them in parallel on a dataset of approximately 21,000 Windows malware samples, generating more than 21 million execution traces. The analysis measured path discovery, code coverage, and computational cost.

The study revealed several key insights:

- Natural run-to-run variation is common: even repeated executions without stimulation frequently expose alternate paths, implying that standard single-run sandbox analyses cover only a small fraction of a program's code.
- Fuzzing increases code coverage and exposes additional behaviors, though it exhibits diminishing returns beyond an optimal mutation intensity.
- Forced execution discovers the largest number of paths overall but incurs substantial computational overhead due to the high number of required runs.
- Symbolic execution was found to be largely impractical at scale, as most samples produced no meaningful output.
- Concolic execution represents a promising middle ground, discovering many distinct paths per run and achieving comparable path diversity to forced execution, but at significantly lower cost. Its limitations primarily stem from incomplete Windows API function summaries and limited support for packed or armored binaries.

Based on these findings, the author issued several practical recommendations for analysts, sandbox developers, and researchers: execute samples multiple times to capture natural behavioral variability, integrate multipath exploration capabilities into sandboxes, and develop malware-specific path prioritization and evaluation metrics.

Overall, this work highlights that fuzzing is an effective and scalable approach to enhance behavioral discovery in malware. As we demonstrate in our main contribution in Chapter 5, fuzzing can indeed be applied effectively to real-world Windows malware, successfully uncovering hidden and environment-sensitive behaviors.

Enviral: Fuzzing the Environment for Evasive Malware Analysis

Gorter *et al.* [58] propose ENVIRAL, an automated framework that "fuzzes" the malware execution environment by mutating the outcomes of environment-query system calls across repeated executions. Instead of purely pursuing transparency, the framework incrementally defeats evasive checks by retaining mutations that increase observed activity, and it reports higher system-call activity than a transparency-driven baseline in the authors' evaluation.

Analysis Methodology. ENVIRAL combines (i) a hook layer that intercepts system calls and applies mutations (return values and/or parameters) and (ii) a controller that repeatedly re-executes the sample with different mutation sets. Mutations are divided into *definite* (handcrafted substitutions for known evasion-related queries) and *volatile* (speculative changes kept only when they increase a custom "coverage" signal based on distinct system calls deduplicated by call stack).

Results report that mutating environment-query syscall outcomes increases observed syscall activity for a non-trivial subset of a large malware dataset. In a focused comparison against a transparency-oriented baseline, ENVIRAL reports a larger increase in distinct syscall activity on samples exhibiting virtualization-related indicators

Key Takeaway. The key contribution is establishing environment-query mutation with a lightweight feedback signal as a scalable way to surface additional behavior in evasive malware, positioned between transparency-only approaches and heavier multipath analyses, reducing the manual work required to bypass evasive queries needed in previous works [41].

Limitations. As discussed in our Chapter 5 comparison, ENVIRAL still requires manual work as it relies on *definite* mutations similarly to BLUEPILL, and does not support systematic backtracking over earlier decision points. Moreover, in our experiments it underperformed PFUZZER and sometimes perturbed executions into less-informative paths.

3.4 Discussion and Identified Gaps

After surveying prior work on environment-sensitive malware, multipath exploration, and evasion-aware analysis, this section consolidates the main *class-level* limitations that repeatedly emerge across approaches, and clarifies the design space position of PFUZZER [15, 81, 70, 107, 142, 98, 28, 41, 58].

Table 3.1 summarizes the reviewed systems along dimensions that matter operationally for malware analysis—work by Botacin is not included as it conducted a SoK analysis of existing approaches rather than proposing a new system.

Work	Class	Main mechanism	Typical limitations
Balzarotti *et al.* [15]	Equivalence-based detection	Record+replay to enforce input equivalence	Multi-process gaps; delayed checks beyond window; kernel-level adversary can probe logging components
DISARM [81]	Multi-sandbox divergence detection	Run across sandboxes + normalization + distance score	Fails if all sandboxes are evaded; shared artifacts (timing/network); attacker can manipulate variability/score
BARECLOUD [70]	Bare-metal divergence detection	Bare-metal execution + cross-env comparison	Stalling; limited fine-grained observability; absent user activity; hardware/network fingerprinting
X-FORCE [107]	Forced execution reachability	Crash-free forcing + predicate overrides	Unsound paths; state distortion from repairs; expensive under heavy branching/obfuscation
PMP [142]	Probabilistic forced execution	PAMA memory preplanning + retries	Unsoundness remains; exploration cost grows with branching; probabilistic errors/coverage gaps
Moser *et al.* [98]	Taint-guided multipath exploration	Taint + snapshot/restore + (restricted) solving	High overhead; incomplete constraints (nonlinear ops); vulnerable to anti-taint strategies [30]
MINESWEEPER [28]	Symbolic/concolic trigger finding	Concrete+symbolic execution + solver	Path explosion; semantic dependence; adversarial constructs can break modeling/scale
BLUEPILL [41]	Transparency	Observe-check-replace via expert hooks	Rule coverage gaps; "fake answers" can perturb; not a systematic behavior enumerator (discussed in Ch. 5)
ENVIRAL [58]	Fuzzing evasions	Mutate syscall outcomes; keep if activity increases	Limited backtracking; may perturb into less-informative paths (discussed in Ch. 5)

Table 3.1. Qualitative summary of the state of the art and their main limitations.

Recurring gaps (class-level). Across these works, four recurring gaps emerge:

- **(G1) Detection vs. discovery**. A large fraction of prior systems primarily provide a *detection* signal (*e.g.,* divergence/evasion score) rather than a mechanism to *discover and characterize* additional behaviors once environment sensitivity is suspected [15, 81, 70]. This leaves analysts with a binary outcome ("evasive or not") but still without the alternative behaviors and their triggers that are needed for reporting, attribution, and defense engineering.

- **(G2) Over-reliance on environment sets**. Multi-sandbox and bare-metal approaches can miss evasions when all tested environments share artifacts (*e.g.,* timing, network constraints, lack of interaction), or when the malware adapts to the specific monitoring family used [81, 70]. Increasing environment diversity helps, but it pushes cost and operational complexity upward and still does not guarantee trigger coverage.

- **(G3) Semantic heaviness and adversarial brittleness**. Taint- and symbolic-execution approaches provide stronger semantic handles on triggers, but they introduce heavy runtime overheads, incomplete modeling assumptions, and well-known brittleness against anti-analysis constructs (*e.g.,*

non-linear transformations and anti-taint strategies) [98, 28, 30]. Forced execution improves reachability but sacrifices feasibility, complicating attribution and potentially distorting behavior [107, 142].

- **(G4) Practical "budgeted exploration" is under-specified**. Several approaches implicitly assume either a fixed short window (single-run sandboxes) or rely on expensive multipath machinery, leaving open how to allocate a realistic analysis budget toward the *most informative* alternative behaviors [70, 107, 98, 28].

Design rationale for PFuzzer. PFuzzer is designed as a *behavior discovery* method that treats environment sensitivity as an exploration surface: it perturbs environment-query outcomes and uses lightweight feedback to prioritize which mutations to keep under a finite execution budget. This choice aims to avoid the two extremes above: it does not require heavy semantic machinery (taint/solvers) and does not rely on infeasible path forcing, while still producing concrete alternative behaviors that single-run and transparency-only approaches may miss [107, 142, 98, 28], in a automatic manner without requiring expert knowledge [41]. A detailed, head-to-head discussion of BluePill and Enviral (including their practical failure modes observed in our experiments) is provided in Chapter 5.

3.5 API Hashing

In this section, we review state-of-the-art research related to our second contribution: the analysis of malware employing anti-static analysis techniques based on API hashing—a widely used API-hiding mechanism designed to thwart static detection and reverse engineering.

To the best of our knowledge, there is no prior *systematic academic study* that treats API hashing as a standalone research subject (*e.g.,* prevalence, algorithmic diversity, and automated deobfuscation pipelines evaluated end-to-end). Nevertheless, API hashing is well known in practitioner communities and is frequently discussed in malware-analysis write-ups and tooling documentation [4, 131, 103, 114, 60, 104, 113]. In practice, analysts typically rely on ad hoc workflows—identify hashing loops, instrument comparison sites, and resolve hash values either by querying community hash databases or by re-implementing the hash function manually—but there is no widely accepted, standardized procedure or evaluation that confirms how consistently these workflows generalize across families and variants.

Obfuscation-Resilient Executable Payload Extraction from Packed Malware

Cheng *et al.* [36] address a persistent obstacle in malware analysis: recovering executable payloads from packed binaries that employ sophisticated anti-analysis techniques—most notably API obfuscation. Packers commonly compress or encrypt a program's payload and then unpack it at runtime. More advanced packers go further by hiding or corrupting import metadata and resolving API addresses through custom runtime mechanisms, which thwarts static analysis and often leaves memory dumps non-executable or semantically opaque.

To overcome these challenges, the authors propose API-Xray, a hardware-assisted tool designed to reconstruct a fully executable PE by recovering the program's import tables (INT/IAT) from an OEP (original entry point) memory snapshot. API-Xray combines static analysis with targeted dynamic micro-execution of suspected call sites to trace complex control-flow chains and resolve the true API targets—even in the presence of trampolines, stolen-code tricks, or anti-debugging routines.

Study of API Obfuscation Schemes. A central contribution of the paper is an empirical study of API obfuscation techniques used by prevalent packers. From a manual analysis of 29 packers, the authors identified 12 that employ advanced obfuscation and categorized the observed schemes as follows:

- **IAT redirection:** IAT entries point to trampolines rather than the real API; the trampoline performs complex bookkeeping (*e.g.,* via SEH) before jumping to the API.
- **Rewrite callsite:** Indirect calls that should use the IAT are rewritten as direct calls at the callsite, bypassing import tables.
- **Stolen code:** Trampolines include the first few bytes of the target API and execute them before transferring control into the API, evading simple entry-point hooks.
- **Anti-debugging routines:** Trampolines invoke anti-debugging checks (timing, environment probes) prior to reaching the API.
- **ROP redirection:** Trampolines employ ROP-like sequences that complicate return paths and disguise the jump to the true API.

Technique: Hardware-Assisted Micro Execution. API-Xray operates on an OEP memory dump produced by an unpacker. It first statically disassembles the dumped memory to locate candidate API call sites—that is, indirect `call` or `jmp`

instructions with unknown targets. For each candidate, the tool spawns a lightweight thread to perform micro-execution from that call site. Because trampoline behavior is generally argument-independent, the trampoline can be executed in isolation to reveal its control-flow chain without executing the full program.

To observe these micro-executions transparently, API-XRAY leverages hardware features rather than software instrumentation, thereby avoiding common forms of detection. In particular:

- **Intel Branch Trace Store (BTS):** BTS records branch source and target addresses at scale (without a small fixed buffer), a crucial capability since some trampolines produce extremely long branch sequences. Unlike other tracing facilities, BTS also captures unconditional direct branches, ensuring complete trace coverage.
- **NX bit (No-Execute):** API-XRAY marks pages belonging to loaded DLLs as non-executable so that when a trampoline eventually jumps into a DLL, a page fault is triggered. This fault serves as a natural breakpoint, halting tracing and signaling the end of the control-flow chain for analysis.

The resulting branch traces are then analyzed to distinguish genuine API invocations from false positives (*e.g.,* anti-debugging calls or intermediate ROP steps). Using heuristics such as stack-frame validation and address-range lookups—which also handle stolen-code cases—API-XRAY maps the final destination to the correct API symbol. Repeating this process across all call sites, the tool reconstructs the INT/IAT, patches the PE header, and rewrites direct calls back to indirect ones, yielding a fully runnable binary.

Results. The evaluation demonstrates strong practical impact. API-XRAY successfully reconstructed import tables for 98.4% of samples in a large corporate dataset encompassing more than 177,000 packed binaries, each exhibiting at least one API-obfuscation technique.

For 7,514 previously unknown or poorly detected variants, running them through API-XRAY substantially improved VirusTotal detection: on average, 22 additional antivirus engines flagged the samples post-reconstruction, as the recovered imports exposed clear malicious indicators.

The large-scale analysis also quantified the prevalence of API obfuscation: 51.9% of packed malware encountered in the wild used at least one obfuscation scheme. Among these, IAT redirection was the most frequent (36.5%), while ROP-based redirection was less common (6.9%).

Relevance to this thesis. Despite API obfuscation and unpacking address problems different from, yet closely related to, our primary focus, the authors'

findings underscore the diagnostic value of recovering API information for malware analysis. They highlight the dual role of the Import Address Table (IAT): it is crucial both for enabling analysis and for adversaries attempting to conceal it. Their hardware-assisted micro-execution and robust import reconstruction show that restoring API visibility significantly improves detection rates and analytical insight, further motivating our textitasis on resolving environment-dependent API interactions in our own approach.

Practitioner Workflows for API Hashing

Although API hashing has limited dedicated coverage in the academic literature, it is commonly encountered in real-world reverse engineering and is documented in practitioner tutorials, malware write-ups, and community tooling [114, 60, 103, 104, 113]. A representative pattern is to (i) recognize the hashed-export lookup loop, (ii) recover or infer the hash algorithm, and then (iii) resolve constants into API names either statically (by recreating the hash function) or dynamically (by observing the resolver at runtime).

Common analyst strategies. In practice, analysts frequently combine debugger- or disassembler-driven inspection with scripting or lookup services:

- **Lookup-based resolution:** once the hash algorithm is identified, analysts query community-maintained repositories/lookup services to map hash values to candidate API names (and sometimes use IDA plugins to annotate call sites).
- **Manual re-implementation:** analysts reconstruct the hash function from the malware (or from a decompiled snippet), then brute-force a candidate API wordlist to recover names matching embedded constants.
- **Runtime tracing:** analysts place breakpoints or logging at comparison sites and resolution points to correlate hash constants with resolved function pointers, which is useful when the sample resolves APIs repeatedly at runtime.

Why this remains a research gap. These workflows are effective in many cases, but they are typically *best-effort*, require substantial manual expertise, and are brittle against minor algorithmic changes, non-standard wordlists (*e.g.,* module+API hashing), or resolver variants. Moreover, community hash repositories require continuous manual updates and do not by themselves provide an automated way to extract the hashing logic from previously unseen samples. This motivates our contribution on this topic, which aims to formalize and automate the extraction of the hashing slice and to recover mappings using the sample itself as an oracle (Chapter 4).

3.6 Lessons Learned and Design Rationale

Research on dynamic malware analysis has historically pursued *transparent analysis*—observing program behavior without being detected. Over nearly two decades, however, complete transparency has proven unattainable: sophisticated malware continues to detect artifacts of virtualization, emulation, and monitoring despite increasingly advanced concealment techniques.

This persistent challenge motivates a shift in perspective. Instead of attempting to eliminate every trace of the analysis environment, this thesis embraces the interaction between malware and its surroundings as a source of analytical insight. Both systems presented in this dissertation are grounded in this principle: rather than minimizing side effects to remain invisible, they deliberately leverage those side effects to stimulate and interpret the malware's reactions. In doing so, evasive logic—traditionally treated as a barrier—becomes a measurable analytical signal.

The next chapter presents our API-hashing deobfuscation pipeline, focusing on how dynamic introspection and emulation can restore semantic visibility when import information is intentionally concealed. Chapter 5 then introduces PFuzzer, our environment-fuzzing framework, and evaluates how budgeted, coverage-guided mutation of environmental conditions uncovers multiple behaviors that conventional single-run analyses typically miss.

Chapter 4

Automated Deobfuscation of API Hashing via Dynamic Introspection

This chapter presents our first contribution: an automated approach to deobfuscate malware that resolves APIs through hashing rather than explicit imports. Whereas the next chapter (Chapter 5) tackles anti-dynamic analysis by exploring environment-sensitive behavior through systematic environment perturbation, here we focus on a complementary anti-*static* technique that conceals functionality at the code level by hiding API dependencies. Despite targeting a different adversarial mechanism, the contribution follows the same guiding principle developed throughout this thesis: instead of bypassing the defense technique or relying on fragile re-implementations, we exploit the malware's own runtime behavior as a source of ground truth.

Contribution Overview. API hashing removes a crucial static signal—import metadata—and frustrates inspection, clustering, and correlation workflows that rely on readable API names. Our method combines lightweight dynamic tracing with slice-guided analysis of the hashing loop, and applies dynamic introspection to isolate and exploit the code that (i) computes per-symbol digests, (ii) compares them against embedded constants, and (iii) resolves matching exports to function addresses. Concretely, within the recovered slice we (1) delimit the hashing region, (2) locate comparison sites and true hash constants, and (3) identify the accumulator-update instructions implementing the hashing routine, which together enable mapping opaque hashes back to API names even when hashing logic is fragmented across helper routines.

To scale mapping and avoid manual reverse engineering, the recovered slice metadata seeds an emulator-backed *hash oracle*: the specimen is repurposed to

compute digests for chosen strings by injecting them where export names are normally processed and observing the resulting hash values computed by the malware's own code. This eliminates guesswork and translation mismatches, and it enables rapid construction and validation of hash tables that are bit-identical to the sample's native outputs.

We implemented the *Deobfuscation & Analysis* component as a DynamoRIO client and the oracle using the Qiling framework. We evaluate along two axes: (**RQ1**) end-to-end effectiveness of the three-stage deobfuscation pipeline on a labeled training set spanning multiple families; and (**RQ2**) oracle fidelity—agreement between emulated and native hash outputs for injected strings and full DLL export sets. We also provide a preliminary study of the hashing algorithms utilized and an *in-the-wild* study over a broader corpus filtered for dynamic API resolution beyond the IAT and `GetProcAddress`.

This chapter extends and consolidates the results appeared in the short paper *All Right Then, (Don't) Keep Your Secrets: Exposing API Hashing in Malware* [25], published at the 22nd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2025).

4.1 Introduction and Problem Context

When examining untrusted Windows software, the Import Address Table (IAT) is a compact but powerful signal: the set of imported APIs lets analysts form early hypotheses about program capabilities. Beyond intuition about functionality, IAT contents are routinely used for static workflows such as clustering and cross-campaign correlation—*e.g.,* bespoke backdoors tied to specific actors have been tracked via their import lists [90]. Consequently, IAT inspection remains a staple of static analysis.

To frustrate this, adversaries employ API-obfuscation techniques [36]. A common tactic is dynamic API resolution: rather than statically listing imported functions in the IAT, the malware locates and binds them at runtime—typically through `LoadLibrary` and `GetProcAddress`, though numerous variations exist—so import metadata no longer reveals true dependencies [43]. While this defeats naïve static checks, analysts recognize the pattern and modern tools can often recover or disambiguate dynamically resolved references.

A more sophisticated evasion is *API hashing*. Instead of resolving functions by name—which would require readable strings in the binary—the malware stores **hashed** encodings of API names and performs covert runtime resolution. Concretely,

it iterates the exports of loaded modules, applies a hashing routine to each symbol name, and compares the result against precomputed hash constants embedded in the binary. On match, it has recovered the target function's address. This pattern defeats string-based recovery and raises the bar for analysis: either the hashing routine must be reconstructed and inverted, or execution must be emulated to map hashes back to API names.

Practitioners often defeat API hashing using experience and recurring patterns, but comprehensive documentation is scarce. Public guidance largely consists of case studies and blog posts [114, 60] and community-maintained repositories of hashing algorithms and seeds [103], while academic work specifically targeting API hashing remains limited.

We investigate a defensive technique that reduces dependence on manual reverse engineering. Because API hashing complicates static understanding, we rely on dynamic program analysis to harvest actionable information in a lightweight, broadly applicable way. When overhead is acceptable, this enables automated, scalable deobfuscation. Moreover, we repurpose a sample as a *hash oracle*: by driving its hashing routine with chosen inputs at the point where symbol names are normally processed, we observe the hashes it computes. The malware's own code thus becomes the ground truth, eliminating manual recovery and simplifying the mapping from opaque hashes to human-readable API identifiers.

Contributions. Conceptually, the contribution lies at the intersection of dynamic binary instrumentation, program slicing, and selective emulation: we automatically delimit the hashing logic used by malware and then repurpose it for oracle-like introspection.

- A slice-guided, three-stage method that automatically exposes API-hashing logic and recovers hash→API mappings.
- A practical *hash oracle* that reuses the malware's own code (via emulation) to compute correct digests for arbitrary strings.
- An open implementation: a DBI-based deobfuscator and an emulator-backed oracle, with an evaluation on real-world samples and a large filtered corpus.

4.2 Motivation

Building on the background of Chapter 2, we now revisit why API hashing represents not merely a nuisance for static inspection but a fundamental obstacle to scalable malware analysis. Understanding this challenge clarifies the design rationale of our automated introspection framework.

API hashing effectively hides API calls from static analysis. Although the technique is well known [4], recent work shows API hashing still evades commercial antivirus (AV) and Endpoint Detection and Response (EDR) [33] and defeats naïve static inspection. In typical analytic workflows, investigators rely on visible call sites and imported API names as anchors for reasoning about program structure and intent; vendors and research groups also compute fingerprints from library/API names to cluster samples and track actor toolchains [90]. Because these fingerprints derive from PE import metadata, hiding imports undermines standard triage and hunting techniques.

Few academic works focus specifically on API hashing; most treat it as one facet of hiding API calls—either to defeat static analysis via obfuscation or to evade dynamic API monitoring. Chatzoglou *et al.* [33] empirically evaluate how obfuscation techniques—including API hashing—affect AV/EDR detection. Cheng *et al.* [36] target unpacking and IAT recovery more generally. Other studies strengthen API monitoring against evasions (*e.g.,* anti-hooking, stolen-code) [68, 67, 43, 8].

Community resources like HashDB [103] aggregate known algorithms and provide lookups from hash values to strings. Useful as they are, hand-curated tables are brittle: small routine variants (*e.g.,* normalization, seeding, or per-sample preprocessing) can invalidate mappings, and updating repositories typically requires manual reverse engineering. In practice, analysts often recover hashed imports by locating the hashing routine and single-stepping through the loop to compute mappings. This is time-consuming, does not scale, and is fragile in the face of additional obfuscations (*e.g.,* split hashing logic, opaque predicates, anti-disassembly techniques). Moreover, not all embedded hashes are exercised at runtime: some remain inactive due to environment checks [24], and some are planted as decoys. Dynamic API instrumentation can reveal resolved APIs once called, but it can be evaded by anti-monitoring measures [8] and forces analysts to abandon static workflows when that is impractical.

As shown later (Section 4.4.2), many samples normalize API-name strings (*e.g.,* to lowercase) before hashing or seed the computation with module-specific values. Such preprocessing is often not modeled by public repositories, yielding digest mismatches even for canonical algorithms. Enumerating variants is impractical.

These challenges motivate our approach: treat each specimen as a *hash oracle* that computes digests with the exact per-sample semantics. By extracting the hashing slice and reusing the malware's code in emulation, we can generate and validate hash tables that are *bit-identical* to runtime values, improving robustness, scaling to large corpora, and supporting clustering and attribution by hashing strategy.

We therefore seek an approach that maintains the empirical fidelity of dynamic execution while retaining the controllability of static methods. The next section presents our design for such a hybrid system.

4.3 Design

We designed and implemented a dynamic analysis tool that automatically deobfuscates API hashing in malware and extracts actionable information. The primary objective is to recover hidden imports and characterize the hashing routine, to fulfill this goal we designed a deobfuscation and analysis method. Based on the information retrieved with such method we then leverage this information to operate the sample as a *hash oracle*: drive the malware's hashing logic with chosen strings—via emulation of the relevant code region—to obtain correct outputs and build reliable hash→name mappings. This removes fragile manual steps (re-implementing the routine) and guarantees correctness because the malware computes the digests.

To meet the above goals, we formalized a minimal and portable model of API hashing behavior and derived a three-stage analysis pipeline from it. Each stage captures one level of semantic understanding—from structural delimitation (Stage 1) to logical inference (Stage 3).

Deobfuscating APIs. As introduced in Chapter 2 (Subsection 2.2.2), API hashing consists of iterating over a module's export table (the *hashing loop*), computing a digest for each candidate symbol name, and resolving the function address when a match with an embedded constant is found. Our goal is to delimit the fragment of code responsible for these steps and to extract key signals from it to deobfuscate API calls and analyze the sample.

Model of API Hashing. Let $\mathcal{E}(M) = \{n_1, \ldots, n_K\}$ be the exported symbol names of module M. For each $n_i \in \mathcal{E}(M)$, the sample computes $h_\theta(n_i)$, where h_θ is a family-specific hashing routine parameterized by θ (*e.g.,* seed, case/encoding policy). The binary embeds a set $C = \{c_1, \ldots, c_J\}$ of target hash constants. A match is detected when $h_\theta(n_i) = c_j$ for some i, j, after which the corresponding function address is resolved.

We define the *program slice* $S = [b_S, e_S]$ as a contiguous dynamic code region with the following properties:

P1: S contains the instructions that compute $h_\theta(n_i)$ for successive n_i.

P2: S includes at least one comparison that evaluates true at least once during iteration, *i.e.,*

$$\exists\, t:\ \mathrm{cmp}\big(\textsc{computed\_hash}(t),\ \textsc{constant}\big) = \textsc{true}$$

P3: Control flow binds the resolved name to a function address *only* after a match within S.

Here, $\textsc{computed\_hash}(t)$ denotes the value produced by the hash-update sequence on iteration t, and $\textsc{constant} \in C$ is the embedded target.

This model is intentionally general: it abstracts away from implementation specifics (*e.g.,* choice of hashing algorithm or normalization policy) while providing invariant properties (**P1**-**P3**) that can be detected dynamically even in heavily obfuscated code.

From Model to Method. Guided by **P1**–**P3**, we recover S and the associated artifacts in three stages: (1) extract the slice by detecting accesses to export-directory structures and the subsequent resolution path; (2) mine comparison sites inside S and identify those with a fixed operand that evaluate true at least once; and (3) localize the value-updating instructions that feed the comparison and produce the matching digest. Each stage runs in a fresh execution to avoid cross-stage interference; in our training dataset, hashing-loop behavior was deterministic across runs[1].

Figure 4.1 sketches the workflow.

Stage 1—Extracting the Hashing Slice.

Goal. Identify the code fragment responsible for computing API-name digests.

We first locate the code region implementing API hashing. After instrumenting the target sample, we monitor executed instructions and look for accesses to DLL data structures used to retrieve API names and addresses (Chapter 2, Section 2.2.2). These typically appear as additions/subtractions between a module base address and offsets into its export directory (*e.g.,* `add`/`sub`). By applying program slicing, we isolate a compact region S, which narrows the search space and enables the subsequent stages.

During this stage we also collect the candidate API names the malware attempts to resolve dynamically. When a correct match occurs, execution reaches the end of

[1]We observed determinism in our training dataset (Section 4.4); this assumption may not always hold and is revisited in the limitations.

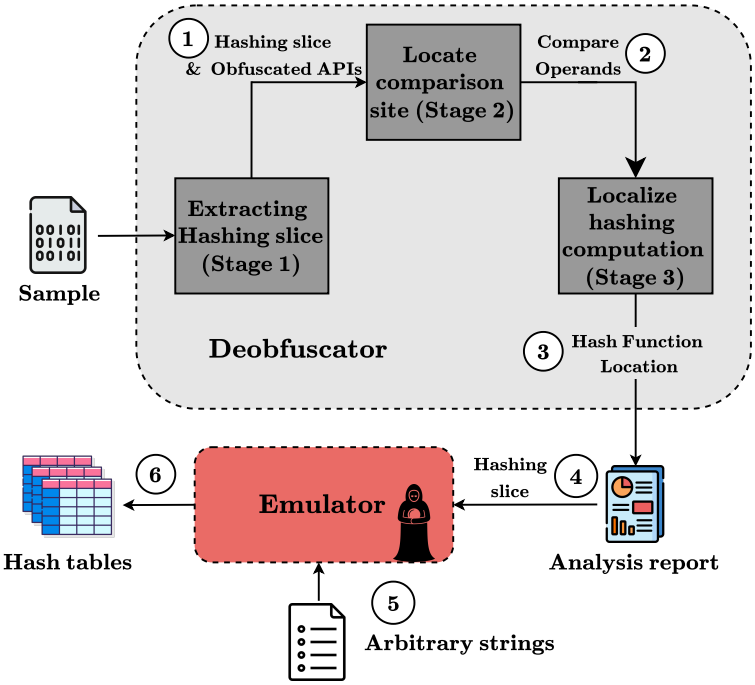


Figure 4.1. Workflow of the proposed approach. (1) Instrumentation locates the hashing slice and collects the APIs resolved dynamically. (2) Within the slice, the tool logs comparison sites and operands. (3) Using these values, it identifies the arithmetic/bitwise updates that produce the matching digest, locating the hashing computation. The analysis report (slice boundaries, compare sites, update sites) then seeds the (4–6) emulator-driven hash oracle, which injects arbitrary strings and produces hash tables mapping strings to their hash values.

our slice: the short sequence of instructions that performs dynamic resolution. Note that a successful match does not guarantee later use—some may serve as decoys or called in alternative executions depending on external characteristics [24]—so functionality should not be inferred from resolution alone. Nonetheless, the list of resolved (obfuscated) APIs is valuable and enables preliminary hash→name mappings once the comparison point is located.

Stage 1 output. The slice boundaries (instruction addresses) and the list of API names whose addresses were resolved dynamically upon a correct match.

Stage 2—Locating Comparison Candidates.

Goal. Identify candidates locations where the hash function can be observed, to help both Stage 3 and emulation as the location serves as a sink where to look for name→hash mappings.

Given the slice boundaries, we restrict attention to comparisons within S. By **P2**,

the slice must contain a comparison between the computed hash and a precomputed constant that evaluates true at least once. Even if the constant is initially unknown, the true comparison is characterized by one fixed operand (the constant) and one varying operand (the running hash). We therefore record each comparison (*e.g.,* `cmp` instructions) and its operands; once an execution reaches the end of S (*i.e.,* a successful resolution), we terminate collection—all interesting comparisons have been observed.

Stage 2 output. A list of comparison instructions in the slice with their operand values.

Stage 3—Recovering the Hashing Computation.

Goal. Identify the hashing logic responsible for turning the plaintext API name string into an hash.

Typical API-hashing routines update an accumulator with simple arithmetic/bitwise instructions (*e.g.,* `add`, `xor`, `rol`, `mul`, **shifts**). Using the output obtained from Stage 2, we scan the slice for update instructions whose operands or outputs relate to the candidate values, without assuming a single callable *hash function*. Ideally, the last update prior to the compare yields the matching value, confirming the location of the hashing logic.

Stage 3 output. The location of the hashing computation site, which should correspond to the implementation of the hash routine.

Collectively, the three stages produce a compact but semantically rich description of each sample's hashing mechanism. The resulting slice metadata—code boundaries, comparison anchors, and value-update locations—suffices to recreate the hash computation in isolation. We now leverage this information to convert each specimen into an executable oracle.

4.3.1 Turning the Malware Into a Hash Oracle

The insights from deobfuscation let us automate hash-table creation. We reuse the slice and comparison sites to *inject* arbitrary strings at the point where the sample reads API names during the hashing loop and observe the hash value at the comparison or last-update site. In effect, the sample serves as a *hash oracle* that produces on-demand digests for arbitrary strings. This sidesteps re-implementing the function in a high-level language and prevents translation mismatches; the values are guaranteed correct because the malware computes them.

Public services such as HashDB [103] remain complementary: when the exact sample is available, our pipeline can enumerate module exports (*e.g.,* KERNEL32.DLL) or arbitrary strings of any kind to build complete tables; when only a few lookups are needed without access to the sample, public repositories are still useful. Our approach automates local table updates for variants and provides a way to cross-check and correlate samples.

Conceptually, this step closes the loop between dynamic observation and automated reasoning. By transforming the adversary's own code into a functional component of our analysis, we operationalize the principle that introspection can be both self-contained and adversary-agnostic.

Although we currently treat each specimen as an independent oracle, malware families often share code, including their API-hashing routines. In principle, once a hashing function has been localized and characterized for one family member, it could be reused across other variants, amortizing the cost of the dynamic analysis. Investigating robust criteria for detecting such reuse (*e.g.,* via code-similarity measures on the recovered slices) and safely sharing oracles at the family level is left as future work.

Emulating (part of) the Slice. To operate the oracle repeatedly, we need to re-execute the relevant portion of S: place a chosen string where the API name normally resides and step execution to the observation point. We use binary emulation (*e.g.,* QEMU [18], QILING [111], UNICORN [132]) to strike a balance between static analysis and full dynamic analysis in a virtual machine. Emulation is scalable and resource-efficient for malware analysis [46, 79, 138, 77]. Limitations include incomplete Windows API/syscall coverage, but API-hashing loops typically occur early, reducing dependence on unimplemented system functionality. While a determined adversary could move unsupported API/syscall calls earlier, this is an engineering hurdle that can be mitigated by extending emulator handlers (which is already well maintained).

4.3.2 Discussion

The design outlined above provides a blueprint for reclaiming obfuscated hashing logic, but its practical significance and boundaries must be evaluated in relation to alternative strategies. We therefore discuss the conceptual position of our method and its interplay with existing families of analysis techniques.

One might argue (see Chapter 2, Section 2.2.2) that the question "which APIs does the sample actually call?" can be answered by instrumenting exported APIs from loaded modules and observing invocations. While such tracing yields a fast, low-effort signal, it conveys only which APIs are invoked—without elucidating

how their names were obfuscated or which exports the malware intentionally concealed. Consequently, it provides behavioral data but lacks explanatory context regarding the underlying hashing strategy. Moreover, API tracers are susceptible to anti-hooking/anti-monitoring techniques common in real-world malware; these can frustrate naïve interposition [67, 68, 43]. Given that concealment is the goal, encountering defenses against monitors is likely, reducing the reliability of pure interposition using common API tracers.

The problem introduced by API hashing overlaps conceptually with information-flow and taint techniques (for example, taint analysis and input-to-state correspondence) [117, 7]. In principle, taint systems could track data originating from symbol-name sources to comparison sinks and hence link constants to names. In practice, however, tainting can be heavy-weight and brittle when faced with obfuscated or self-modifying code; adversarial constructions (implicit flows, deliberate obfuscation of data movement) can easily undermine taint propagation [30]. Lighter-weight input-to-state correspondences that infer which inputs influence internal program state may fail when the program applies complex transformations or when the relationship between inputs and internal comparisons is non-trivial. Crucially, these techniques tend to focus on which inputs affect state, not on reconstructing the exact hashing function or mapping opaque hash constants to plaintext API names. Thus neither approach yields a turnkey solution for API hashing in the general case.

For what concern emulation instead, emulating the entire sample within a DBI framework would introduce significant overhead and expose the analysis to anti-DBI artifacts (see Chapter 2, Section 2.3.2). A hybrid DBI strategy that intercepts each hashing-loop iteration and injects arbitrary names would, in theory, produce the same mapping but at higher cost and with more exposure to anti-DBI checks. In contrast, our slice-guided emulation targets only the minimal, relevant computation, improving efficiency and robustness.

Overall, our design anticipates most of the problems and drawbacks we may encounter during analysis of this technique in malware while remaining as general as possible. This yields a practical balance between accuracy, performance, and resistance to common anti-analysis techniques.

Limitations. As with any dynamic instrumentation-based technique, our approach inherits both the strengths and fragilities of its underlying platform. We summarize here the main technical and conceptual limitations, drawing also on our broader experience gained while working on PFuzzer (Chapter 5).

Our slice-guided strategy has several limitations. First, as noted by D'Elia *et al.* [40], anti-DBI techniques can impede correct analysis. We built on DynamoRIO, which exhibits fewer known anti-DBI artifacts than some alternatives (*e.g.,* Intel

Pin). Still, our broader study in Chapter 5 found two practical issues: (1) failures concentrated on samples protected with strong packers (*e.g.,* Themida), which could be mitigated by porting to a different DBI; and (2) a small fraction of failures (1.76% of our dataset) attributable to DynamoRIO internals rather than malware logic. These observations suggest that framework choice matters, and DynamoRIO offers a good trade-off between robustness and overhead.

When hashing and comparison logic are split across loosely coupled loops or spread over many instructions, the dynamic slice can become large. A larger slice increases instrumentation and emulation costs and may degrade performance. Stage 3 may produce multiple candidate sites (*e.g.,* similar arithmetic/memory updates), leading to false positives. Here, Stage 2 acts as a strong anchor: observing a true precomputed hash constant at runtime provides a reliable target even when the producing sequence is ambiguous. In practice, these limitations primarily affect efficiency and may require modest manual triage, but they do not prevent automated mapping.

The emulation stage depends on correctly identifying the hashing loop and its pivotal points. If the slice misses relevant instructions (*e.g.,* due to incomplete traces or anti-tracing tricks), emulation can fail or produce incorrect results. While our experiments are encouraging, guaranteeing correct identification in all cases remains future work. Notably, the emulator is modular: an analyst can supply manually identified points if automatic identification fails.

When the hashing algorithm varies across obfuscated APIs (*e.g.,* per-module seeding), the emulator may not match author-intended digests unless the seed is propagated. In such cases, multiple instances of the hashing loop must be emulated to obtain correct mappings, or the entire loop should be emulated to collect all variants. Further automating seed detection and normalization inference is left for future work.

From an attacker's perspective, our approach raises the bar but does not close the problem. Adversaries could respond by distributing hashing logic across multiple functions, obfuscating comparison operations further, or combining multiple hash functions. Our slice-guided derivation remains effective as long as (i) export-name bytes are eventually consumed, and (ii) the final digest is compared against a fixed target (or otherwise used as a control-flow guard) that is observable at runtime.

Our method assumes determinism in the hashing loop. A determined adversary could randomize the order of names or vary loop behavior across runs. This would not make mapping impossible but would degrade efficiency: if the correspondence between iteration index and exported symbol changes, more iterations may be required before encountering a target mapping. As a mitigation, Stage 2 could trace

comparisons until the end of an iteration rather than stopping at the first successful match.

A more direct countermeasure targets our reliance on stable code *anchors*. In the current prototype, slice boundaries and observation points are recorded as instruction addresses (e.g., b_S, e_S, and the comparison/update sites). An adversary can destabilize these anchors across executions by introducing indirection stubs (trampolines), repeated re-unpacking with address randomization, or polymorphic rewriting of the resolver code. While such transformations preserve functionality, they can relocate the hashing loop and invalidate address-based reuse of previously recovered slice metadata.

Mitigating this would require making anchors *semantic* rather than address-based, *e.g.,* (1) re-discovering the slice each run from invariant runtime events (export-table accesses and subsequent resolution), (2) identifying comparison/update sites by data-flow signatures (values flowing from the name buffer to the match), or (3) switching to cross-run matching based on instruction bytes, basic-block hashes, or CFG fragments instead of absolute addresses. Studying these evolutions empirically and hardening anchor stability is an important direction for future work.

Other countermeasures could include decoy constants or dead comparisons (*i.e.,* populating many constants and comparisons that never guard a real resolution), forcing more robust *true-use* validation and potentially slowing down the analysis.

In short, the main trade-offs of our approach are between automation, performance, and adversarial robustness. The technique is effective and practical in a wide range of real-world samples, but it remains vulnerable to sophisticated anti-analysis measures and to scenarios that substantially enlarge the analysis slice or invalidate determinism assumptions. We expect these gaps can be narrowed through incremental implementation improvements and continued research.

4.3.3 Implementation Details

Having described the conceptual design, we now turn to its concrete realization. This section details the instrumentation, monitoring, and emulation mechanisms that implement the three-stage deobfuscation pipeline.

Our pipeline comprises two components: (1) a *deobfuscator & analysis* tool that reveals obfuscated APIs and gathers slice/compare/update information; and (2) an *emulator engine* that drives the hash oracle to populate hash tables. The first component requires instruction-level monitoring and operand inspection to detect accesses to export-directory offsets and to correlate comparison and update values. The second requires binary emulation to re-run the slice fragment with arbitrary inputs.

We implement the deobfuscator as a DynamoRIO [27] client (preferred over Intel Pin [86] due to startup latency and familiarity). For emulation we use the Qiling framework [111], which builds atop Unicorn [132] and provides PE loading, syscall handlers, a virtual file system and registry, and other conveniences for running Windows binaries or snippets without a full OS; it is actively maintained.

Program Slice. On module-load events, the client walks each module's internal structures to reach the export table and record offsets relevant for API hashing (*e.g.,* `AddressOfNames`, `AddressOfFunctions`, `NumberOfFunctions`), along with module image name and base/end addresses. We then inspect executed instructions, searching for references to these offsets. In particular, we look for code retrieving names and resolving function addresses—typically implemented as additions between the module base and the relevant offsets (*e.g.,* `add`). We test instructions with at least two operands, check whether one operand references a recorded module base (using an interval tree containing information for each loaded module for efficiency), and if so, test whether the other references one of the recorded offsets. Once such instructions are located, we define the program slice and log the names of resolved APIs.

In the second stage, we inspect each comparison instruction within the slice, recording operand values until the first time we reach the slice end (successful resolution). We then filter comparison candidates to those with equal operands at least once between slice start and end—one being the computed hash, the other the precomputed constant.

In the third stage, we search for the candidate values in arithmetic/bitwise instructions commonly used in hashing (Table 4.1).

Arithmetic	**Bitwise / Shift / Rotate**
ADD (`add`)	AND (`and`)
ADC (`adc`) — Add with carry	OR (`or`)
SUB (`sub`)	XOR (`xor`)
SBB (`sbb`) — Subtract with borrow	NOT (`not`)
MUL (`mul`)	SHL (`shl`) — Logical left shift
IMUL (`imul`) — Signed multiply	SHR (`shr`) — Logical right shift
DIV (`div`)	SAL (`sal`) — Alias of `shl`
IDIV (`idiv`) — Signed divide	SAR (`sar`) — Arithmetic right shift
INC (`inc`)	ROL (`rol`)
DEC (`dec`)	ROR (`ror`)

Table 4.1. Instruction types monitored in Stage 3 when searching for hash-update operations.

In samples with multiple candidate slices, we currently explore each of them, but using simple heuristics we could prioritize the most promising ones to improve performances of the analysis. An example could be ranking regions by combining slice length and the number of hash comparisons observed at runtime. Shorter slices with frequent comparisons can be inspected first, which biases the analysis toward compact, table-driven hashing loops. This ranking provides a basic budget-management mechanism: in resource-constrained settings, an analyst or automated system can restrict attention to the top k ranked regions per sample.

As discussed in Section 2.2.2, API hashing may also obfuscate module (DLL) names. We therefore apply the same heuristics to deobfuscate module-name hashing when needed. In Section 4.4, one sample in our training set performs both API-name and module-name hashing [130, 83].

As discussed in Section 4.3.2, our heuristics have trade-offs. Address computations that define the slice can be decomposed into smaller steps, complicating detection. Relaxing strictness can recover such cases at the cost of more false positives; since the technique must access the relevant offsets, the slice remains discoverable. Adversaries may intentionally enlarge the slice; this impacts performance but not correctness. Finally, operations of interest can be implemented with multiple instruction sequences; we target a broad but finite set and extend coverage as new patterns appear.

Emulator. The emulator must execute a *code fragment*—the portion of the hashing loop inside the slice until the comparison instruction—repeatedly, with arbitrary input strings, *without* running the whole program or writing a bespoke driver. The difficulty is that an arbitrary basic-block entry is not self-contained: correctness depends on CPU state (*e.g.,* general registers, EFLAGS, etc.), stack and heap contents, module layout, and OS structures (*e.g.,* PEB, TEB). If those are inconsistent, emulation diverges (*i.e.,* wrong outputs) or crashes (*i.e.,* invalid memory, bad SEH, etc.).

To overcome this challenge we opted for a process snapshot at the slice entry (b_S) of the context, by dumping the program state we are able to ensure a valid emulation. At each iteration of the fragment we can restore such context to ensure that each repetition is valid, and produces correct values for each string injected. For the program state dump we opted for an IDAPython script, and the information is then placed in a file so that the emulator framework can utilize to restore the context when needed. Arguably we could dump these information during deobfuscation once Stage 1 locates the hashing slice, but to avoid adding additional instrumentation we opted for the lighter services offered by IDA Pro.

To inject arbitrary strings into the location where the API name is read, we use the emulator's memory utilities: the replacement string is written to the memory

region that holds the name (typically the result of the `add` computing the name pointer).

These implementation choices are driven by pragmatism: they aim to balance transparency and extensibility, allowing future researchers to replicate, extend, or adapt the method to new binary formats or architectures with minimal modification.

4.4 Evaluation

To validate our claims, we conduct a two-tier evaluation. The first (Section 4.4.1) tests the pipeline's technical effectiveness and the fidelity of the emulated oracle on controlled samples. The second (Section 4.4.2) explores scalability and real-world applicability through corpus-level screening. Together, these analyses assess both the accuracy and the practical reach of our method.

We divide the evaluation into two parts: (1) effectiveness of our analysis pipeline (RQ1 and RQ2) on a labeled training set; and (2) a study of hashing routines and a first *in-the-wild* analysis on a broader corpus filtered for dynamic API resolution beyond the IAT and `GetProcAddress`. This extends our DIMVA'25 work by introducing the emulator engine and broadening the dataset.

For (1), we construct a small labeled dataset of eight real-world samples from seven families that employ API hashing, including widely studied families such as *Emotet*, *LokiBot*, and *Revil*, with ground truth derived from manual reverse engineering and online resources [4, 130, 83, 131, 102].

For (2), we analyze a larger corpus (2018–2023) to obtain a first view of API hashing in the wild and to identify practical constraints and improvement opportunities for our approach.

Experiments use the same environment as in Chapter 5 (Section 5.5). Each sample analyzed with the deobfuscation tool is tested for up to one minute; because API hashing typically appears early and DBI introduces overhead, this budget suffices for observing the relevant signals despite the introduced overhead.

4.4.1 Answering the Research Questions

In this section we address two research questions:

- **RQ1 (Pipeline effectiveness).** How effectively does the three-stage deobfuscation recover API hashing logic end-to-end across diverse malware?
- **RQ2 (Oracle fidelity).** To what extent does the emulated hashing slice reproduce the exact hash values computed by the malware during native execution, when driven with the same inputs?

		Stage 1		Stage 2	Stage 3	
Family	**Sample**	*Slice start*	*Slice end*	*Compare value*	*Hash value*	*Hash function*
BlackMatter	50c4970003a84cab1bf2634631fe39d7	✓	✓	✓	✓	✓
BlackMatter v2	d0512f2063cbd79fb0f770817cc81ab3	✓	✓	✓	✓	✓
Conti	bc92ea510a5630c770d9443be4b40fde	✓	✓	✓	✓	✓
Emotet	68c76c3403570a22ce7a60a1b68d9056	✓	✓	✓	✓	✗
Lockbit	628e4a77536859ffc2853005924db2ef	✗	✗	✗	✗	✗
Netwalker	73de5babf166f28dc81d6c2faa369379	✓	✓	✓	✓	✓
Revil	890a58f200dfff23165df9e1b088e58f	✓	✓	✓	✓	✗
Zloader	5c76c41f9d0cc939240b3101541b5475	✓	✓	✓	✓	✓

Table 4.2. Outcomes for the three stages on the training set. Stage 1 reports slice boundaries; Stage 2 reports a valid compare constant; Stage 3 reports (1) the recovered hash value at an update site and (2) whether the enclosing hash function was identified.

RQ1: Deobfuscator effectiveness on the training dataset

RQ1: How effectively does the three-stage deobfuscation recover API hashing logic end-to-end across diverse malware?

We executed our deobfuscator on the training samples; results are shown in Table 4.2.

To answer this research question we assess the deobfuscator's effectiveness per stage. A stage is considered successful if it achieves its intended goal: (1) Stage 1 correctly extracts the hashing slice by delimiting the loop or loop-like region that iterates over API names; (2) Stage 2 identifies at least one valid comparison site acting as an observation anchor (a conditional whose evaluation determines whether a candidate matches the target); and (3) Stage 3 recovers a concrete hash value at an update site and attributes it to the enclosing hash function. Results are summarized in Table 4.2.

Stage 1 succeeded in 7 out of 8 samples (87.5%), accurately delimiting the hashing region by tracking references to export-table offsets. The only failure occurred with *Lockbit*, where DynamoRIO crashed during dynamic tracing—an engineering issue rather than a conceptual limitation. Stage 2 achieved the same rate (7/8), successfully identifying valid comparison constants in all functioning cases. This corroborates **P2**: any complete traversal of the slice must evaluate at least one true comparison, providing a reliable point of observation. Stage 3 produced mixed outcomes. The correct hash value at an update site was recovered in 7/8 samples (87.5%), but identifying the enclosing hash function succeeded only in 5/8 (62.5%). For *Emotet* and *Revil*, although the update site was located, the final digest transformation occurred outside the callee implementing the main hashing algorithm, often in a short post-call transformation or helper routine. This pattern indicates that hashing logic is sometimes distributed across multiple functions, challenging the assumption

that hashing occurs within a single function scope.

Overall, the pipeline produced at least one correct deobfuscation in 7/8 samples (87.5%), confirming the intended interplay among stages: Stage 1 effectively narrows the search space, Stage 2 provides stable anchors, and Stage 3 performs fine-grained localization when the computation is contained. The main weakness lies in function attribution, which fails when hashing logic spans call boundaries. Extending Stage 3 with cross-function slicing would address this limitation.

These findings indicate that the proposed deobfuscation pipeline is both general and effective: it operates without relying on overly specific heuristics that could be easily bypassed, while recovering meaningful and verifiable information from real-world malware. We next evaluate whether the recovered data suffices to emulate the hashing behavior of each sample outside its native execution environment.

RQ2: Oracle Fidelity

> **RQ2:** To what extent does the emulated hashing slice reproduce the exact hash values computed by the malware during native execution, when driven with the same inputs?

We evaluate RQ2 by checking whether the emulator, when driven with the same API names observed during native execution, reproduces identical hash values. Concretely, for each sample we first collected the set of observed *(API name, hash)* pairs during dynamic tracing of the original process (*i.e.,* APIs that are obfuscated and resolved at runtime). Using the slice metadata produced by our deobfuscator and analysis tool (for *Lockbit*, where our tool crashed, we manually retrieved the slice information), we injected the same API names into the emulated name buffer and executed the slice until the observation point (the compare or last-update site). For each injected name we compared the digest produced by the emulator to the digest recorded during native execution.

This agreement rate provides an empirical measure of fidelity between the emulated and native hashing logic and serves as a practical validation criterion for automated slice extraction. This validates the conceptual foundation of our deobfuscation method—namely, that instruction-level inspection of export-structure accesses is sufficient to reconstruct and emulate the hashing semantics of obfuscated binaries.

On our training set, the string-injection test succeeded for all samples: the emulator reproduced the native digest for injected names and the hashing loop behaved as expected (including early termination on match). This demonstrates that (1) the emulator can replay the relevant fragment in a valid context, and (2) it

can serve as a reliable hash oracle for the tested inputs.

We observed a recurring limitation in two *BlackMatter* variants (Section 4.4.2): these samples compute per-module digests by applying a ROT-13-like transform seeded with the hash of the module name. In such cases, the digest for an API depends on a seed computed prior to the emulated slice. While primarily an engineering challenge (the seed can be reconstructed or propagated), it complicates fully automated emulation. A practical mitigation is to emulate the *entire* hashing loop to collect the family of variant hashes and reveal how the seed affects outputs.

This successful emulation also substantiates the intuition underlying our deobfuscation approach—inspecting instructions that access module-internal structures automatically extracts accurate and valuable information about API hashing.

Having established correctness on controlled specimens, we now extend our scope to understand diversity and prevalence of API hashing in the wild, and to evaluate how our approach generalizes beyond the training dataset.

4.4.2 Studying Hashing-Routine Internals and API Hashing in the Wild

In this second part of our evaluation, we seek for studying and get some valuable insights on the hashing algorithm used by the sample. We believe that this information is useful to provide a complete analysis of these malware, but it also introduces the possibily of clustering malware by algorithm, helping in the analysis of new samples in quickly identifying similarities with previous malware.

We then analyze a bigger corpus of samples, obtained from VirusTotal Academic dataset and Bazaar feed from VX Underground between 2018 and 2023. This set of experiments serves to obtain a first look to API hashing in the wild, observing trends, as well as identifying room for improvement for our approach.

This second part of the evaluation, with the introduction of the emulator previously described and evaluated in Section 4.4.1, extends the work presented at DIMVA'25. The work is still preliminary and requires further studies, extending the research questions also to the larger representative dataset as well as introducing new experiments to better study both this technique and our approach to it.

Hashing-Algorithm Characterization

We conducted a focused characterization of the API-hashing routines present in our dataset to understand implementation diversity and to guide clustering and emulation strategies. Leveraging the slice metadata produced by our deobfuscation tool, we manually inspected the decompiled pseudocode for each specimen and compared the recovered logic against canonical implementations (*e.g.,* HashDB)

and online resources [131, 102]. This manual comparison was necessary to resolve subtle per-sample behaviors—such as string normalization or module-dependent seeding—that are often not captured in reference repositories.

This analysis not only validates the correctness of our deobfuscation but also enriches the threat-modeling perspective of the dissertation: understanding how adversaries evolve their hashing logic informs both static tooling and dynamic countermeasures.

Table 4.3 summarizes the hashing algorithms identified across families.

Family	Sample	Algorithm
BlackMatter	50c4970003a84cab1bf2634631fe39d7	ROT13 Seeded
BlackMatter v2	d0512f2063cbd79fb0f770817cc81ab3	ROT13 Seeded
Conti	bc92ea510a5630c770d9443be4b40fde	Murmur2A
Emotet	68c76c3403570a22ce7a60a1b68d9056	SDBM
Lockbit	628e4a77536859ffc2853005924db2ef	ROT13
Netwalker	73de5babf166f28dc81d6c2faa369379	CRC32
Revil	890a58f200dfff23165df9e1b088e58f	Polynomial Rolling Hash
Zloader	5c76c41f9d0cc939240b3101541b5475	PJW Hash Function

Table 4.3. Hashing algorithms utilized by samples in our training dataset.

We observed substantial heterogeneity. Some families use canonical functions (*e.g.,* SDBM, Murmur2A, CRC32), while others adopt custom or composite schemes (*e.g.,* seeded ROT13 variants or multi-stage PJW-like designs). Comparing the pseudocode obtained with decompilation with HashDB reference implementations revealed material differences affecting digests, notably explicit case-folding and module-dependent seeding. These behaviors cause direct lookups against external repositories to fail unless the same preprocessing is reproduced.

Two recurring patterns yielded the largest divergence from reference code. *String normalization* appears in *Netwalker* and *Zloader*: API names are case-normalized (*i.e.,* lowercased) before hashing. *Module-dependent seeding* is prominent in our *BlackMatter* specimens: a ROT13-derived routine seeds the hash with a digest of the module name, causing the same API to hash differently across modules. Naïve emulation with a fixed seed thus fails to reproduce runtime digests; correct emulation must either iterate the module-resolution loop and apply per-module seeds or inject the appropriate module names when constructing tables.

These patterns make blind string lookup brittle. Treating the specimen as a *hash oracle—i.e.,* extracting and executing its hashing slice—reproduces runtime digests exactly and complements repository-based workflows. The most time-consuming reconstructions were those where computation is fragmented across helpers or interleaved with control-flow obfuscation (*e.g., Zloader*). Our deobfuscation tool significantly reduced manual effort by isolating slices and the sequences that manipulate name buffers.

Looking ahead, we plan to automate this characterization pipeline—detecting normalization, inferring seeding strategies, and other techniques that may be relevant to accomplish a complete analysis—to scale analysis across large corpora with minimal manual effort. Automating these steps should improve clustering by hashing strategy and make our emulation-based hash-table generation more robust. Overall, the diversity we measured—from canonical algorithms to seeded and preprocessed variants—underscores that external repositories alone may miss per-sample semantics; sample-grounded extraction and execution of hashing slices provides the high-fidelity matching required for reliable triage and clustering.

In broader perspective, these results suggest that hash-based obfuscation is not merely a matter of concealment but a design pattern with its own evolutionary dynamics. Systematic, automated introspection of such routines could thus serve as a window into lineage evolution, code reuse, and attacker tradecraft across malware families.

Prevalence of API Hashing: Large-Scale Corpus Screening and Characterization

While targeted experiments confirm feasibility, large-scale evidence is needed to assess validity—*i.e.,* whether API hashing is frequent enough and structured enough to justify systematic countermeasures. This section therefore scales our analysis to tens of thousands of real-world specimens.

To complement the focused evaluation on labeled samples, we conducted a large-scale study to estimate the prevalence of API hashing and to identify practical constraints for automated deobfuscation in the wild. This study extends the DIMVA'25 poster by incorporating our deobfuscation approach to broadening the dataset considerably.

Corpus Construction. Starting from the dataset introduced in Chapter 5 (Section 5.4), we considered 315,000 Windows PE samples collected from VirusTotal Academic and the VX Underground Bazaar feed (2018–2023). We deduplicated the corpus using VirusTotal's *vhash*, yielding 71,151 unique binaries. Unless otherwise noted, all experiments in this subsection use the same execution environment and one-minute per-sample budget adopted throughout the chapter. We adopt a one-minute budget to mitigate the impact of stalling code sequences while still extracting as much behavioral information as possible under a constrained analysis window. In practice, however, this setting is not directly scalable for an AV vendor that processes large volumes of samples daily. A more scalable workflow would first screen samples using static methods, for example by developing ad-hoc rules (*e.g.,* Yara [137]) to identify, with acceptable precision, binaries that are likely to contain

API-hashing logic. Such pre-filtering would significantly improve the throughput of our approach; nonetheless, since our goal is to study the technique itself, we prioritize a more precise dynamic analysis for this work.

Screening for API-Hashing Indicators. Because API hashing is only one among several mechanisms for concealing API dependencies, we first applied a lightweight dynamic screening to identify specimens that plausibly perform dynamic resolution beyond the IAT and outside of textbook `LoadLibrary`/`GetProcAddress` usage. Concretely, we developed a DynamoRIO-based prefilter that flags a sample if, within one minute of execution: (1) it invokes APIs not present in its IAT; and (2) those APIs are not preceded by standard dynamic-resolution calls. This screening surfaced 36,978 candidates. To reduce obvious false positives (*e.g.,* specimens invoking very few APIs due to early termination or benign stubs), we retained only samples with at least four such invocations, resulting in a working set of 21,991 specimens for the subsequent analysis.

Running the Deobfuscation Pipeline at Scale. We executed our three-stage deobfuscation tool (Section 4.3) on all 21,991 screened samples, allotting one minute per specimen. For each sample we recorded whether the pipeline reached: Stage 1 slice extraction (both boundaries found); Stage 2 a valid comparison site with equal operands observed at least once within the slice; and Stage 3 the hashing-update site consistent with the comparison, *i.e.,* a plausible localization of the hash computation.

Table 4.4 summarizes results by year of first-seen. Across the full set, Stage 1 succeeded for 1,686 samples (7.67% of 21,991), spanning 330 distinct malware families (identified using AVClass [118, 11]). Among these, Stage 2 identified at least one comparison with matching operands in 382 samples (22.66% of those with a slice). Finally, Stage 3 localized a matching update site in 156 samples (9.25% of those with a slice).[2] The year-by-year distribution indicates a persistent presence of candidates across 2018–2023 rather than a single-year anomaly apart from the starting number of samples in year 2020.

Discussion. The Stage 1 rate (7.67%) reflects that many dynamically resolving samples do not traverse PE export structures in the canonical manner our slice detector targets (Section 4.3.3). Stage 2 and Stage 3 conditional rates (22.66% and 9.25% over Stage 1, respectively) indicate that, once a valid slice is identified, we can often anchor on true comparisons and, in a non-trivial subset, localize the corresponding update sequence. The drop from Stage 2 to Stage 3 may be due to hashing logic that (1) interleaves with additional transformations or utilizes currently unmanaged cases, or (2) simply is not actually using API hashing.

[2]For clarity: 156/21,991 corresponds to 0.71% of the screened set. We report both normalizations because Stage 3 is conditional on Stage 1.

Year	N. Samples	Stage 1 *Slice boundaries*	Stage 2 *Compare value*	Stage 3 *Hash function*
2018	3705	383	66	29
2019	2300	520	114	36
2020	11,950	192	41	26
2021	1031	122	25	11
2022	1474	226	57	22
2023	1531	243	79	32
Total	**21,991**	**1,686**	**382**	**156**

Table 4.4. Stage-wise outcomes on the screened corpus. For each year, we report: (1) samples with a valid slice (Stage 1), (2) samples with a matching-operand comparison inside the slice (Stage 2), and (3) samples where a consistent hashing-update site was localized (Stage 3).

Manual Inspection and Derived Observations. To understand failures to reach Stage 3 and to calibrate priorities for future improvements, we manually analyzed some samples across years that stalled after Stage 1 or Stage 2. Two recurring patterns emerged:

- **Instructional diversity.** Several specimens use comparison or "check" idioms that our current Stage 2 heuristics do not flag. For example, we observed cases where the computed accumulator is validated via `xor` against a constant rather than by a direct `cmp`/`sub` equality check. This is compatible with our model (one operand fixed, one varying and respecting **P2** of the slice), but the instruction palette must be expanded to cover such idioms. Doing so will likely increase Stage 2 recall and, transitively, Stage 3 opportunities—albeit with a potential increase in benign collisions that Stage 3 will have to disambiguate.
- **Non-hashing dynamic resolution.** A distinct subset performs textbook iteration over export names but resolves targets by string equality against a small hardcoded set, *i.e.,* without hashing. These specimens are structurally similar at Stage 1 (same export-table traversal) but fall outside the scope of API hashing proper. Distinguishing them automatically (*e.g.,* by confirming the presence/absence of a running accumulator consumed at the match site) will improve both precision and the representativeness of the final API-hashing cohort.

Threats to Validity. First, the one-minute budget trades completeness for scalability; longer runs may surface additional slices or matches in specimens with heavy unpacking/initialization. Second, our Stage 1 detector focuses on canonical access patterns to export-directory structures; samples that compute intermediate addresses in more obfuscated ways, or that use alternative resolution mechanisms (*e.g.,* direct LDR/NT-layer calls, COM/.NET reflection, custom loaders) will not be

captured. Third, family/year assignment depends on external metadata (first-seen dates, AV labels) that can be noisy; we therefore treat family counts as indicative rather than definitive. Finally, Stage 3 localization is conservative by design; fragmented implementations distributed across helpers may produce true positives that we currently classify as indeterminate.

Takeaway and Implications. Even under conservative assumptions, we identified 1,686 specimens with valid hashing-like slices and localized hashing updates in 156 of them, spanning 2018–2023 and 330 families. While 156 corresponds to 9.25% of the Stage 1 cohort (and 0.71% of screened samples), the absolute counts and longitudinal distribution underscore that API hashing remains a persistent obfuscation strategy. These results motivate two concrete engineering directions: (1) broaden Stage 2 to include additional comparison idioms (*e.g.,* `xor`-based equality, masked compares), and (2) introduce a fast discriminator to separate true hashing from string-based resolution at Stage 1. We expect these refinements to increase Stage 2/Stage 3 yields and to produce a cleaner, larger cohort for automated oracle-based characterization (Section 4.4.2).

We presented a dynamic introspection-based method for automated deobfuscation of API hashing. By recovering and emulating hashing routines extracted from malware binaries, the approach reconstructs API mappings with high accuracy and minimal manual intervention. This complements PFuzzer's environment-driven behavioral analysis, together forming a unified strategy for engaging with evasive malware both semantically and behaviorally instead of aiming for transparency as a way to overcome anti-analysis tactics.

Chapter 5

Fuzzing the Windows Environment to Discover Multiple Behaviors in Malware

In this chapter, we present our main contribution: PFuzzer, a novel approach for multi-path exploration of environment-sensitive malware through coverage-guided fuzzing. This system is designed to uncover hidden execution states that manifest only under specific environmental conditions. We show how environmental features—often exploited by malware for evasion—can instead be leveraged as an analytical vector to expand behavioral visibility.

We begin by defining the challenges of analyzing environment-sensitive malware and by motivating our approach. We then give an overview of PFuzzer's design and describe key implementation details, walking through each component that enables automated malware fuzzing. Finally, we evaluate the system with experiments designed to answer four research questions: (1) does the approach expose environment-sensitive behavior in real-world malware? (2) how does PFuzzer perform compared to state-of-the-art solutions? (3) can PFuzzer construct environments matching the characteristics requested by malware? and (4) what alternative behaviors does PFuzzer reveal?

The research contents described in this chapter was presented at 10th IEEE European Symposium on Security and Privacy (EuroS&P 2025) by the title of *PFUZZER: Practical, Sound, and Effective Multi-path Analysis of Environment-sensitive Malware with Coverage-guided Fuzzing.*

5.1 Introduction

Malware has long been a major hazard within the cybersecurity domain. Because static inspection of an untrusted binary frequently falls short, contemporary defensive workflows commonly rely on dynamic analysis to observe runtime behavior and collect execution traces. Many tools operate under the assumption that observing a single run suffices for characterization [23]; however, that assumption breaks down rapidly when confronted with malware whose behavior depends on properties of its execution environment.

Environment-sensitive malware exhibits markedly different behaviors depending on features of the host system. A prominent subset—so-called *evasive* malware, whose presence is rising [88, 51]—can detect telltale signs of analysis platforms and respond by terminating early or otherwise hiding malicious activity. Beyond deliberate sandbox-detection, behavioral variance can stem from legitimate operational attributes of the machine: the malware may rely on specific local processes or services (*e.g.,* for data exfiltration), or it may be crafted to run only on victims with particular system characteristics, as in targeted attacks.

We can define environment-sensitive malware as follows. Let M be a malware sample and let E denote the observable environment (*e.g.,* file system, registry, processes, timing). We say that M is *environment-sensitive* if there exist two environments E_1 and E_2 such that the sets of executed basic blocks differ, and this difference is causally driven by explicit environment queries:

$$\exists E_1, E_2 : I(M, E_1) \neq I(M, E_2) \ \wedge \ \exists Q \text{ such that } Q(E_1) \neq Q(E_2),$$

where $I(M, E)$ is the set of basic blocks executed by M under environment E, and Q ranges over environment-querying instructions (*e.g.,* API calls, system calls). This definition excludes purely incidental variation (*e.g.,* natural variation) that does not arise from explicit environment-dependent branching.

Although the research community has addressed these challenges for many years [81], environment-sensitive malware continues to pose substantial difficulty for existing defenses.

Broadly speaking, prior work follows two directions. One direction aims to make analysis platforms more transparent by removing or disguising artifacts that reveal the analysis environment; the other inspects how a sample's instructions consume environmental information and attempts to drive alternative decision paths during execution. Each of these strategies has practical limitations: they may only cover a subset of fingerprinting techniques and decision heuristics, often need manual updates to handle new evasion methods, or face scalability and performance constraints that

limit their applicability to large-scale, real-world malware analysis.

We trace these shortcomings to two central methodological pitfalls: (1) depending on fixed, a priori expectations about what a sample will do, and (2) insisting on costly, fine-grained analysis of its code. The first is risky because no single anticipated outcome covers every sample and many possible actions are simply unforeseeable; the second is problematic because deep code inspection is computationally expensive and adversaries can deliberately make program logic arbitrarily harder to analyze.

Viewed more closely, these problems echo a successful line of work in software testing: *coverage-guided fuzzing*. Although the targets of fuzzers are not adversarial, fuzzing methods share a useful paradigm: they start without precise knowledge of valid inputs or which inputs are good, and through repeated executions driven by input mutations and generic runtime feedback, they gradually discover the space of inputs a program accepts.

Experts in fuzzing might correctly point out that many standard fuzzing components (*e.g.,* how inputs are obtained and mutated, what coverage signals are used, and how seeds are scheduled) do not immediately translate to malware analysis or would perform poorly out of the box. With this work we close that gap by adapting and evaluating techniques that enable coverage-guided fuzzing to analyze environment-sensitive malware effectively.

Concretely, we treat alternative execution environments as an input-generation problem. We introduce the notion of execution policies as a unified mechanism to intercept and modify a sample's interactions with environmental information sources; the heart of a policy is a set of mutation rules that alter those interactions. To measure progress in constructing useful environments, we design a coverage feedback mechanism that combines sensitivity to externally observable execution effects (because these often best indicate malicious behavior) with inspection of code coverage so partial progress is not overlooked. Our external coverage component separately records (1) accesses to environmental sources—since these are natural mutation targets—and (2) conspicuous actions that change system state—since new actions can signal that certain expectations were satisfied. As described later, these coverage facts both guide the application of mutations and drive the fuzzing scheduler.

Unlike prior multi-path techniques that rely on forced execution [107], our approach guarantees soundness: any analyst or automated tool can apply the prescriptions produced by an execution policy and deterministically reproduce the observed behaviors. Unlike work that depends on symbolic execution or taint-tracking, we deliberately avoid heavy code-analysis; instead, we observe program behavior across a bounded set of real executions.

Differently from many existing defenses, our method does not depend on expert-crafted expectations or fixed assumptions about attacker tactics—it attempts to automatically synthesize appropriate outcomes to environmental queries using only coverage-driven feedback. The result is a method that is practical, formally sound, and—as our evaluation demonstrates—highly effective.

To quantify the method's capabilities, which we realize in an implementation named PFuzzer, we assembled a manually annotated corpus of 1078 samples that should be useful to the research community. These samples were selected by deduplication, clustering, and reversing, starting from 315,231 Windows malware binaries gathered by vetted sources between 2018 and 2023. We evaluate how PFuzzer uncovers additional behaviors for these samples and compare its effectiveness against two state-of-the-art systems, BluePill [41] and Enviral [58]. Across our benchmarks, PFuzzer clearly outperforms the competitors: for instance, it reveals conspicuous activity that the strongest baseline misses in 36.09% of the samples. Sometimes this newly discovered activity stems from evasive checks that PFuzzer neutralizes; more commonly it is additional malicious behavior that augments the conspicuous actions seen during the baseline execution. Analyses that rely on a single run—the common industry practice [23]—would likely miss these additional behaviors. Moreover, PFuzzer can also activate latent evasive behaviors in samples that already show their primary malicious activity during the baseline run. Our experimental section presents statistics and observed trends for the alternative execution paths we unlock and examines the internal performance factors that drive PFuzzer's results.

In summary, the contributions of our research are:

- Novel designs for core fuzzer components (mutation rules, coverage signals, and scheduling) specifically adapted to malware analysis;
- A curated dataset of environment-sensitive samples, annotated with the evasive tactics we observed;
- An extensive experimental evaluation of the PFuzzer implementation that quantifies its capabilities, performance characteristics, and the alternative behaviors it reveals.

5.2 Motivation

As reviewed in Chapter 3, environment-sensitive malware has been studied for over more than a decade, with researchers taking several distinct approaches. Early work followed two main threads: (1) building analysis platforms that are hard for malware to detect (*e.g.,* Cobra [136], Ether [45]); and (2) detecting meaningful differences

between executions in different environments or between instrumented sandboxes and uninstrumented hosts [81, 15, 70].

As adversaries and their techniques evolved, the field branched into two broad directions. One direction seeks ever more transparent environments by returning fake indicators to defeat known evasive checks [41]. The other focuses on analyzing how code interacts with environmental inputs and on forcing alternative decision outcomes at runtime using program-analysis techniques [107, 142].

A central observation of our work is this:

Takeaway: Environment-sensitive malware is not necessarily evasive.

While evasive malware is an important and increasingly prevalent threat [51, 88], yet many samples suppress or alter their behavior based on characteristics of the execution environment that are unrelated to analysis artifacts. In particular, *targeted malware* [141] is designed to remain dormant and only activate on machines that exhibit distinctive properties associated with the intended victim (*e.g.,* specific organizations or government entities). Solutions like BluePill, which are tailored to counter evasive malware, are therefore inherently incapable of handling this broader class.

Consequently, the community has increasingly favored multi-path exploration as a general strategy. Existing multi-path approaches include forced execution—where branch outcomes are flipped and execution is repaired as needed—and fine-grained program analyses (*i.e.,* symbolic execution, taint analysis) that locate dependencies between code and environmental inputs.

As discussed in Section 3.4, these approaches have important limitations: single-path methods are brittle and often require manual tuning, while accurate multi-path code modeling is expensive and does not scale well.

Recently, fuzzing has emerged as a promising alternative. Coverage-guided fuzzing, in particular, repeatedly exercises a program under light instrumentation and automatically learns inputs that reach previously unexplored paths—without requiring expert knowledge or complete code models [58, 23, 24]. Although malware analysis differs from conventional software testing, we show that careful fuzzer design can adapt coverage-guided techniques to this domain. This thesis develops and evaluates such adaptations, and presents a practical, effective fuzzing-based approach for multi-path analysis of environment-sensitive malware.

Conventional analysis frameworks strive to minimize their footprint to avoid detection. PFuzzer instead embraces the opposite philosophy: we intentionally perturb environmental conditions to provoke different execution paths. By doing so, we exploit the very mechanisms malware uses to hide, transforming evasion logic

into a driver of exploration. This shift—from transparency to engagement—lays the conceptual groundwork for the complementary analysis presented later in this thesis.

5.3 Design

To address the issues described in Sections 5.2, we design and implement a coverage-guided fuzzer called PFuzzer, tailored specifically to the peculiarities of environment-sensitive malware.

Overview. Our approach frames the analysis as an iterative process in which the fuzzer incrementally alters the sample's view of its environment and detects consequential changes in behavior via a coverage mechanism. Following the usual coverage-guided paradigm, PFuzzer keeps a queue of distinct environment configurations that produced previously unseen execution states; the fuzzer selects from this queue to generate new environment variants, with the aim of progressively satisfying additional conditional checks or expectations embedded in the sample.

This view recasts the traditional notion of a mutatable input: the input becomes the set of environmental attributes we can change as the sample queries them. Practically, we accomplish this by interposing on the sample's accesses to environmental information sources (for example, operating-system APIs and certain CPU instructions) and optionally applying mutations to those interactions—for instance, pretending that a file is absent or supplying a different registry value. We call these mutatable entities *execution policies*, and each policy is realized as a collection of mutation rules.

At runtime, a monitor performs the interposition necessary to apply a policy's mutations and, at the same time, gathers coverage feedback for that execution. The fuzzer treats a policy as *interesting* when its execution yields novel coverage; such policies are retained for further exploration. For each policy we compute (1) a significance score that estimates how much progress the policy has produced and (2) an energy score that governs how aggressively we should attempt further mutations of that policy.

A scheduler balances these scores to allocate fuzzing effort across policies. This mechanism prevents the fuzzer from wasting resources chasing partial, ultimately fruitless progress in one direction (enabling backtracking), while ensuring it does not overlook alternative malicious behaviors that only surface under different environment configurations (for example, when specific system services are present, particular programs are running, or certain configurations hold).

Below we describe the construction of an effective coverage feedback, the principles for soundly mutating environments, and the concrete design and implementation

decisions behind the fuzzer, followed by a discussion of other methodological considerations.

5.3.1 Coverage Feedback

The first major obstacle is to decide how well an execution environment presented to a sample matches the conditions and run-time behaviors the sample expects. This is a pervasive difficulty for almost all automated malware-analysis techniques, and it is fundamentally hard because no authoritative ground truth exists.

In practice, malware probes its environment by reading one or more *items of environmental information* and later executing decision logic that depends on those readings. The difficulty for an analyst is that this decision code can have arbitrary complexity and may not reveal, in an obvious way, which items caused a decision to be taken.

Several factors complicate the picture. A sample may act on a retrieved value only long after the initial query. Obfuscation and adversarial constructs (*e.g.,* implicit flows [30]) can deliberately hinder static or dynamic code inspection. A program may aggregate results from multiple sources into an opaque summary and decide solely on that summary, leaving little hint about which individual items failed to meet expectations. Malware authors may also perform queries whose purpose is not to influence control flow but to distract an analyst or a human reviewer.

These and related tactics substantially weaken the effectiveness of existing environment-sensitive analysis techniques.

Our approach departs sharply from attempts to reason about code fragments or to forcibly steer specific code outcomes (*e.g.,* [98, 23, 141]). Instead, by repeatedly *manipulating* the environment perceived by the sample (details follow), we produce a set of diverse executions that lets us estimate whether—and to what extent—we are constructing environments that satisfy the sample's implicit expectations. The objective is to collect run-time facts that remain informative even when adversarial obfuscation or deception is present.

To quantify exploration progress we introduce a *coverage feedback* mechanism: a lightweight, run-time observable record of execution facts. In the adversarial context we target, many feedback signals that work well in other domains can be rendered unreliable. For instance, relying solely on code coverage is risky because a malicious author may pepper the binary with conditional stalling or deceptive code paths; discovering those regions can produce a misleading impression of progress.

To obtain a more robust picture of execution, our feedback combines two complementary sources:

- *external coverage*: observations about externally visible actions performed on

the machine by the sample, emphasizing (1) the environmental queries the sample issues and (2) state changes it effects on the host; and

- *internal coverage*: observations about internal program states traversed during execution.

External Coverage. This component captures two classes of externally observable behavior. First, we record which environmental items the sample queries. This encompasses invocations of OS-level APIs that reveal environment state (for example, queries about files, running processes, registry keys, hardware interfaces, timing sources, GUI objects, and system configuration) as well as the use of CPU instructions that expose environment-dependent information (for example, `cpuid` and `rdtsc`).

Second, we monitor both transient and persistent modifications the sample attempts on the global system state. Transient effects include attempted network communications and the creation of ephemeral IPC artifacts (*e.g.,* mutexes). Persistent effects include creation, modification, or deletion of files and processes, the appearance of GUI windows, and alterations to system settings.

Internal Coverage. This feedback source includes information about internal program states, such as code portions traversed by the sample during one execution: for example, the identity of the basic blocks covered. Optionally, it can include lightweight data-flow hints, such as argument values passed to functions whose semantics consist of comparing strings or byte sequences.

Discussion. We contend that both coverage sources are necessary and that omitting either can materially reduce effectiveness by creating blind spots.

> **Takeaway:** Malware fuzzers must protect against attacks targeting internal coverage by pairing it with an external-coverage metric.

If external coverage is neglected, internal coverage alone may not tell us whether we have satisfied any of the sample's environment-based expectations: observing new code regions does not necessarily imply that the sample received environment values it wanted. Conversely, externally observed queries and state changes are often the most direct and reliable indicators that some expectation was met-machine-state changes, in particular, are strong evidence of progress.

Nonetheless, external observations can be misleading: newly observed queries might be deliberate decoys, and there may be situations where no externally visible effect occurs because certain decision points remain unmet. In those cases internal coverage is highly valuable: it can reveal that new portions of the binary are being exercised and can suggest which interactions should be manipulated in subsequent attempts. Still, internal coverage by itself is insufficient: it can be deliberately noisy

(*e.g.,* code present only to confound analysis) and it does not always distinguish executions that include meaningful environmental interactions from those that do not. Combining both sources therefore yields a more resilient and informative feedback signal for driving our environment-mutation strategy.

This thesis uses coverage feedback primarily as a *means* to discover additional behaviors, not as an object of study in itself. Future work could therefore focus on systematically refining coverage metrics for adversarial binaries: for instance, improving how internal and external signals are weighted, how decoy-heavy executions are discounted, and how progress is measured when coverage grows but externally visible behavior remains unchanged. Such an effort would mirror the role that coverage-metric research has played in software-testing practice, where more discriminative feedback often translates into better path discovery and more actionable findings [110, 139].

5.3.2 Mutating the Environment

The second major design challenge is how to meaningfully change what a sample perceives of its environment so that our fuzzer can provoke different behaviors. The coverage signals introduced in Section 5.3.1 let us distinguish when a sample reacts differently under two—possibly related—environment configurations; however, to systematically present those configurations we introduce two central ingredients: the notion of an *execution policy* and an *execution monitor* that enforces the policy while collecting coverage.

Execution Policy. An execution policy (or, for brevity, policy) is a collection of rules that instruct the execution monitor how to alter, prior to retrieval, the contents of environmental information items accessed by the sample. The intent is to produce *feasible* query outcomes and to observe—via coverage feedback—how the sample responds.

A rule specifies how to deviate from the system's normal behavior for a particular access: for example, by changing the value the OS returns for a query, by denying an access that would otherwise succeed, or by forging a plausible result for an operation the host would normally refuse. A rule is agnostic about which program instructions will later act on the returned value; it only prescribes how the access itself should behave.

A policy may be empty, in which case the monitor does not alter machine operation and only records coverage during execution. We annotate each policy with the external coverage observed while running under it; this annotation enables us to produce successor policies by applying fuzzing-style *mutations* to the current rule set.

Mutations. A mutation consumes an execution policy together with the external coverage observed under that policy. For each access reported in the coverage trace, we check whether the current policy already includes one or more rules for that access and act accordingly.

If no rule exists for a recorded access, the mutator may nondeterministically add a new rule implementing one of the mutation options described above (modifying the returned contents, denying the access, or forging a successful result), or it may leave the access unmodified. If one or more rules are already present, each rule can be altered (*e.g.,* by changing value-related fields), replaced by a different rule, removed, or kept as-is.

Mutations are designed to be aware of the structure of the environmental information. They can add, modify, or delete portions of the contents by means of operators (*e.g.,* bit and byte flipping, substitution with random or controlled contents) that preserve the layout and type of the involved environmental information item. Mutations for primitive values can apply to one or more constituent bytes, whereas for structured values—such as with a list of files or processes—to their structure, too (by, for example, adding, duplicating, splitting, merging, or deleting list items).

Execution Monitor. The execution monitor has two responsibilities: (1) recording coverage facts, and (2) intercepting and manipulating accesses to environmental information according to the active execution policy.

Both roles rely on standard interposition techniques available in dynamic program analysis toolchains. When logging external coverage for an access, the monitor records not only the operation used (instruction or API) but also its origin in the sample, the concrete input arguments, and the exit status where applicable. These details permit us to generate rules that precisely match future occurrences of the same access and to control their behavior as desired.

When the monitor modifies an access, it guarantees that the outcome presented to the sample is a feasible result a properly configured machine could produce. For example, if the sample queries a non-existent file, the monitor does not merely flip a status code; instead, it may redirect the access to a synthetic file created on demand. This policy reduces the likelihood of producing spurious executions that would confuse the exploration process.

Discussion. The central aim of our design is to identify, and further investigate sample states and actions that have not previously been observed. We use coverage traces and mutation rules to derive new execution policies; these policies in turn drive fresh runs in which the sample's perception of the environment is altered at each step.

Coverage information lets us detect policies that yield previously unseen internal states or externally observable actions. Operating fully automatically, our method (1) evaluates the end-to-end effects on the sample of the values it obtains for environmental items, and (2) synthesizes execution policies that can expose novel behaviors. As we illustrate throughout this chapter, step (2) is achieved by progressively learning from past executions conducted under diverse environment policies.

Note that distinct rules may prescribe different behaviors for semantically identical information when accessed through different operations (for example, different APIs). This is intentional: it enables us to probe implementation-dependent integrity checks that a sample might perform. Through subsequent mutations, the system can converge toward *eventual consistency*, *i.e.,* a policy in which coherent rules govern all distinct accesses to the same logical item.

Optionally, mutations can exploit internal coverage information about recorded data flows. For instance, capturing strings or byte sequences passed to comparison routines may hint at specific values that a mutation should hide or materialize-analogous to input-to-state analyses used to overcome fuzzing obstacles. In our experiments this mode provided no measurable benefit for the evaluated samples (and is therefore disabled by default), as the policy-generation process described above already sufficed to handle their queries. Nevertheless, this strategy could help with highly targeted or complex cases that demands precise values embedded in system-produced outputs.

5.3.3 An Efficient Fuzzer for Environments

The components introduced above close much of the conceptual gap required to apply fuzzing to the analysis of untrusted binaries that exhibit environment-sensitive behavior. In this section we present a concrete fuzzer design that, by leveraging execution policies and a coverage-driven signal, can distinguish and efficiently explore multiple execution paths to reveal environment-dependent behavior in malware. Figure 5.1 gives a compact view of PFuzzer's workflow.

An Atypical Fuzzer. Traditional fuzzer architectures are not directly suitable for malware analysis for several reasons. In adversarial settings, simple feedback channels such as raw code coverage can be intentionally undermined; yet a feedback signal with good sensitivity remains essential for driving modern fuzzers. Our coverage mechanism is designed to remain informative in the face of multiple fingerprinting and deception attempts, thereby reducing the risk that the fuzzer will be misled by adversarial artifacts.

Moreover, unlike conventional targets that accept inputs from a single, well-

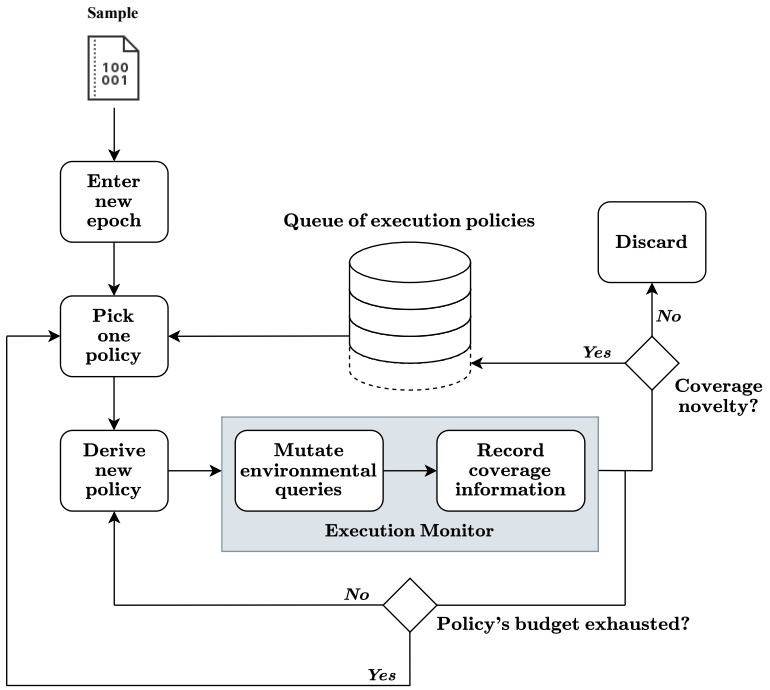


Figure 5.1. Workflow of PFuzzer: deriving new execution policies and checking the coverage feedback for retaining interesting policies in the queue.

defined channel, malware interrogates many heterogeneous sources of environmental information. The execution policy abstraction unifies these sources: it provides the fuzzer with a canonical input to mutate by interposing on the sample's environmental queries and modifying their normal outcomes. By changing the environment the sample observes, we can drive it down different environment-sensitive code paths without touching or statically analyzing its code. The observed queries and state changes then guide—automatically and adaptively—which parts of the environment to alter in subsequent runs.

While the two ideas above are sufficient to provoke and differentiate behaviors, additional design choices are necessary to ensure the fuzzer operates *efficiently*. The remainder of this section outlines those choices.

Epochs and Policy Generation. Classic coverage-guided fuzzers produce new test cases by applying mutations freely to prior inputs; mutations and derivation history may be visible only through naming conventions or external metadata. In our setting, unconstrained mutation risks losing the specific changes that produced novel behavior. To address this, PFuzzer evolves execution policies in discrete *epochs*, balancing exploration (discovering new behaviors) with exploitation (further refining promising policies).

At the start of an epoch we pick a single policy from the fuzzer's *queue* and

generate new child policies from it. Each epoch has an execution budget and a time limit. Child policies inherit their parent's rule set and are extended nondeterministically: for each environmental access recorded in the parent's external-coverage annotation, we decide whether to apply one of the allowed mutations to that access (or to leave it unchanged). Each generated policy is then exercised by running the sample and collecting coverage; a *coverage map* is used to determine whether the policy yields coverage that is novel relative to all prior runs. Policies that produce new coverage are enqueued for further exploration. When the epoch's budget is exhausted, a new epoch begins and the cycle repeats.

Scheduling. Execution begins from an empty policy (or from a human-provided seed policy, *e.g.,* from prior analysis). The scheduler that selects which queued policy to fuzz next is critical to overall efficiency. For every policy in the queue we maintain two scheduling metrics: a *significance score* that ranks the policy's relative promise and an *energy score* that specifies the epoch budget it receives if selected.

When an epoch yields one or more novel child policies, we compute each child's significance as a weighted sum of three components: externally visible system-changing actions, external environmental queries, and internal coverage (basic blocks). Formally:

$$w_1 * actions_{sys\_change} + w_2 * actions_{query} + w_3 * BBs$$

where we count unique *actions* of each type and the number of covered basic blocks BBs. We use $w_1 \geq w_2 \geq w_3$ to bias the scheduler toward policies that produce externally visible, system-changing activity, which we treat as a stronger proxy for analyst-relevant progress than internal coverage alone. We did not perform a sensitivity analysis/ablation on (w_1, w_2, w_3) due to time constraints quantifying the robustness of PFuzzer to different weight settings is left as future work.

At the end of an epoch we sort newly added policies by significance. We boost the significance of the highest-ranked policies (top-2 in our implementation) and lower the scores of the others. Energy assignments follow a matching ordering: top-ranked policies receive a larger epoch budget while remaining policies receive equal, smaller budgets. This policy prioritization encourages focused exploitation of policies that delivered additional coverage while constraining bias toward many closely related children.

A policy's energy score determines how much work it can receive when selected. After an epoch concludes, we decrement the selected policy's energy to account for the exploration (*i.e.,* mutation) performed. Consequently, policies can accumulate high significance yet run out of energy, at which point the scheduler will favor other candidates.

If no policy in the queue has remaining energy, the scheduler replenishes energy for queued policies according to a predefined policy, beginning with policies from older epochs. This replenishment facilitates backtracking by giving previously explored policies another opportunity to be mutated and re-executed, increasing the chance of discovering execution paths that require different sequences of mutations.

Performance and scalability. A fundamental limitation of PFuzzer is that coverage-guided exploration is intrinsically more resource-intensive than single-run dynamic analysis, because it requires executing each sample multiple times under different environmental mutations and collecting coverage feedback at each iteration. In our evaluation, this cost is explicit in the experimental design: PFuzzer operates under a bounded per-sample budget (*e.g.,* a 10-minute campaign) composed of many individual executions (*e.g.,* runs capped at 55 seconds), which directly trades throughput for behavioral completeness. As a consequence, PFuzzer is not intended to match the throughput of large-scale triage pipelines that aim to process massive malware streams through a single short sandbox run; rather, it is best positioned as a second-stage analysis for samples that appear low-activity, evasive, or otherwise high-value to analyze more thoroughly.

This trade-off is also visible when comparing PFuzzer to lighter-weight systems such as Enviral for environment-fuzzing, which are designed with a stronger emphasis on throughput and reduced instrumentation overhead. In our head-to-head comparison, we configured Enviral with shorter per-run time limits than PFuzzer (to reflect its lower overhead and its original evaluation methodology) and we discuss in detail the methodological pitfalls of extremely short budgets for DBI-based systems, where early seconds can be dominated by warm-up and code-cache population. Overall, this positions Enviral as a reasonable option when the operational goal is to rapidly probe many samples under a tight per-sample time cap, whereas PFuzzer prioritizes deeper, more analyst-oriented multi-path behavioral reconstruction within a larger (but explicit) resource budget.

Compared to *static* analysis, the throughput gap is even more pronounced: static pipelines can often process very large corpora by extracting features from PE metadata, byte histograms, section characteristics, imports/exports, and strings, without executing the sample. This scalability is one reason why static learning-based detectors and triage systems are widely studied and used in practice, including approaches trained on large static PE feature sets [5] or directly on raw bytes [112]. However, static methods fundamentally observe the program *as a file*, which makes them sensitive to the same concealment mechanisms that motivate this thesis: packing and API hiding (including API hashing) can remove or distort highly informative static signals such as import tables and readable API names. Moreover,

static analysis cannot directly observe which branch is taken under a specific host configuration, and therefore cannot by itself characterize environment-triggered, multi-path behavior that only manifests under particular environmental conditions. For these reasons, PFuzzer should be interpreted as complementary to scalable static analysis: static methods can serve as a first-stage filter at scale, while PFuzzer can be applied selectively to recover behaviors that are unlikely to be observable via one-shot execution or file-only inspection [120].

5.3.4 Implementation

Realizing our design requires both the ability to interpose on accesses to environmental information and the ability to record internal coverage traces. For practicality and flexibility we implement PFuzzer using dynamic binary instrumentation (DBI). We selected the DynamoRIO [27] framework rather than Intel Pin [86] because, in our experience, Pin incurs larger startup latency with its just-in-time compilation of instrumented code.

To concentrate on behavior originating from the sample itself, we filter out calls and low-level instructions that originate inside OS library internals by using an interval-tree approach similar to [43]. We further leverage call-site information to distinguish multiple invocations of the same operation arising from different locations inside the sample.

Coverage. To interpose on environmental-information accesses, we instrument a curated set of 68 APIs (counting `A/E/ExA/ExW` variants as a single entry) that prior work considers relevant [41, 58]. Where applicable we also capture and later mutate important arguments (for example, output buffers). These APIs span the main sources of environment-dependent information attackers exploit: files, processes, registry, drivers, hardware identifiers, timing sources, GUI objects, networking, and similar.

We additionally instrument `cpuid` and `rdtsc` instructions because they disclose device and timing characteristics frequently used for fingerprinting [41]. We do not attempt to instrument every instruction associated with anti-analysis checks (*e.g.,* `int` and `popfd`), since many of those are tightly coupled to DBI transparency issues[1].

To observe transient and persistent modifications of host state, we monitor a broader set of 146 APIs assembled from a popular API catalog [82] after applying minor refinements.

For internal coverage we adopt a fuzzer-style bookkeeping: we maintain a per-execution coverage map and a cumulative map of all previously executed basic blocks.

[1] Such gaps are rarer in DynamoRIO than in Pin [42].

At the end of a run we compare these maps to determine whether a policy exercised any previously unseen basic blocks. We reuse these maps to detect new invocations of APIs in our categories of interest as well. Our implementation offers the option to choose between classic *basic blocks counting* and *edge coverage*, but in our evaluation we used edge coverage.

Mutations. We provide four general mutation operators to steer the outcome of environmental accesses:

- *Normal course:* leave the access unmodified and allow it to proceed as on a normal machine;
- *Force success:* if the access would normally fail (*e.g.,* a missing file), make engineering provisions so it succeeds (for example, redirecting to a dummy file);
- *Force failure:* cause an otherwise successful access to fail;
- *Retarget:* substitute a different, semantically meaningful value for the one returned by the operation (for example, changing an output list element such as a process name).

We only apply operators that are sound for the given operation—*i.e.,* outcomes that a legitimately configured machine might produce. For instance, a call to `GetUserNameA` can either be left unchanged or retargeted to a different username string, but it would be unsound to present an incompatible type or layout.

Time-related operations receive specialized handling: we can alter delay primitives (zeroing or scaling delays) and distort elapsed-time readings (to fabricate expected time intervals). Rather than following a fixed policy (*e.g.,* BluePill), we nondeterministically choose at each run whether to apply timing adjustments and record the chosen parameters.

Execution. We orchestrate runs using a single driver process that spawns a fresh child process for each sample execution and tears it down on timeout. While this choice reduces raw throughput compared to in-process forkserver or snapshot techniques, it preserves soundness across runs.[2] When samples spawn child processes, we interpose on those children as well and apply the same policy rules used for the original process.

[2]Fuzzing mechanisms like forkservers or snapshots [49] would improve throughput but mature Windows equivalents for user-space code are limited at the time of writing.

5.3.5 Discussion

Our method operationalizes practical multi-path execution by adapting the coverage-guided fuzzing paradigm to the environment-manipulation problem. To handle environment-sensitive behavior, we combine coverage signals sensitive to different notions of progress (new environment queries, conspicuous system changes, new code reached) and use them to drive a biased exploration of the immense space of feasible environment configurations. This exploration uncovers both "faulty" environments in which a sample conceals its intent and the "right" environments where its intended activities manifest—which can differ substantially from one another.

Significance scores let us prioritize execution policies according to their coverage novelty, while energy scores regulate how much effort each policy receives, enabling the fuzzer to spread work across multiple promising alternative scenarios. This scheduling makes the approach applicable not only to evasive samples (often characterized by a single "right" path) but also to malware that exhibits multiple behaviors depending on environment attributes. Given that prior studies report such behavioral diversity as common [115, 12, 23] while single-run analyses routinely miss it, our technique may also yield benefits for downstream tasks (*e.g.,* classification), which we leave outside the scope of this research.

Our experimental evaluation demonstrates that PFuzzer is both practical and effective: it outperforms competing solutions on evasive samples and reveals extra behaviors even for binaries that appear malicious in a baseline run. We do not claim to have solved evasion in general; rather, our goal is to demonstrate that coverage-guided fuzzing offers a viable and productive direction for multi-path analysis that avoids many conceptual roadblocks of previous approaches (Chapter 3). We expect follow-up work to refine and extend these core ideas.

Limitations

We acknowledge several limitations.

Stalling and time budgets. Dynamic analyses inevitably operate under finite time budgets, and adversaries aware of our design can craft stalling strategies that do not rely on standard delay primitives. While this is a general problem, existing detection techniques for such procrastination have been proposed [73]. In

[3] When the set of possible outputs is small (*e.g.,* keyboard layouts), PFuzzer draws replacement values from partial lists embedded in the implementation. For collections (*e.g.,* process lists) or resources, it selects from predetermined entities (fixed strings, paths to typed decoy files). For simple calls (*e.g.,* `GetCursorPos`), it picks values randomly within the argument's valid range.

[4] Fuzzing mechanisms like forkservers or snapshots [49] would help but embodiments for Windows user code are currently missing or far less mature than on Linux. We are confident this will change in future.

addition, higher-fidelity implementations could mitigate this limitation by leveraging live snapshots and checkpointing, instead of always restarting executions from the program entrypoint.

Dependency on a curated API set. Although the fuzzer's entropy determines which APIs to mutate and which values to try, the system still depends on limited expert input to compile (i) the initial list of environment-querying APIs and (ii) the relevant arguments to consider. An adversary may query the environment via an API outside our curated set. Extending the set is straightforward and could be partially automated: a simple extension would observe previously unseen API calls, introspect their prototypes, and apply conservative mutations; more advanced extensions could use language-model-assisted parsing of Microsoft documentation [93] to identify environment-related APIs and synthesize valid inputs.

Anti-fuzzing strategies. Anti-fuzzing techniques exist that aim to foil automated analysis [64, 59]. Some of these are already mitigated by our design (*e.g.,* SpeedBump and BranchTrap [64], and fake-code approaches [59]), while others are largely orthogonal because they target crash-based feedback, which PFuzzer does not rely on. Nevertheless, attackers could develop strategies that target specific implementation choices or assumptions on which PFuzzer relies; no obvious generic attacks were identified at the time of writing.

Hard-to-guess values and indirection. Our technique can struggle when malware requires very specific, non-general values (*e.g.,* a uniquely chosen username) or when relevant values are only reachable via indirection (*e.g.,* embedded in files that the malware opens). For the former, Section 5.3.2 explains how optional internal data-flow observations could expose such values through input-to-state correspondence [7]. For indirection, the model could be extended to explicitly represent these dependencies and make their contents manipulable by the fuzzer.

Exotic invocations and uncovered corner cases. Finally, adversaries might hide API invocations using exotic techniques [8] or exploit corner cases that our mutations do not cover (*e.g.,* novel timing channels). These gaps can likely be narrowed through incremental implementation improvements and continued research.

Limits related to the DBI substrate. From an implementation standpoint, DBI is not perfectly transparent [40], although it remains widely used in malware analysis (*e.g.,* [41, 108, 88, 51]). In our experience, DBI frameworks may exhibit incompatibilities with certain commercial protectors and may also be affected by anti-DBI techniques. A concrete example is an incompatibility we observed with the Themida protector, which we reported to the DynamoRIO maintainers (GitHub issue #6567).

After extensive testing on (i) real-world malware samples protected with Themida

and (ii) synthetic test programs protected with Themida, across multiple DynamoRIO versions and in discussion with the DynamoRIO developers, we concluded that addressing the underlying cache-coherency behavior would require deep changes to DynamoRIO internals. Given that only a small fraction of our dataset was affected (approximately 1200 samples), we did not pursue a full fix and treated this as a limitation of the current implementation that could be addressed with additional engineering effort.

More broadly, we did not attempt to harden DynamoRIO against fingerprinting vectors [42], because such issues were rarely observed in our evaluation corpus. With dedicated engineering effort, PFuzzer could be ported to alternative substrates (*e.g.,* a different DBI framework or a more transparent virtualization-based substrate) [66]; the primary objective of this thesis is to validate the feasibility and utility of environment fuzzing as an analysis strategy, rather than to fully eliminate substrate artifacts.

Limits in generating new mutations. The mutation rules associated with the 68 monitored APIs that access environmental information are manually engineered. This introduces a scalability limitation: as new environment-querying APIs (or undocumented variants) and artifacts are discovered, the monitor must be extended with additional interposition logic and mutation operators. That said, our approach typically requires less *per-technique* manual effort than bypass-oriented systems such as BluePill. Rather than implementing handcrafted bypasses for specific checks, PFuzzer treats each query outcome as a fuzzable variable and explores alternative results under a bounded budget. Once an API is interposed, PFuzzer can often trigger (or neutralize) environment-dependent branches without requiring an analyst to first recognize and model the precise evasion logic.

In practice, the engineering effort required to add support for a new public Windows API is usually limited to implementing a wrapper according to the official documentation (*e.g.,* Microsoft documentation) [93]. Undocumented APIs can require additional reverse engineering, but community resources and prior work often provide sufficient behavioral descriptions for instrumentation. However, manually adding support mechanisms for each new API does not scale well. Automating (i) the expansion of the monitored API set and (ii) the synthesis of mutation operators remains an important direction for future work. For example, one could mine frequent environment-query patterns from large corpora, or leverage code-assistance tools (*e.g.,* LLM-based assistants [3, 106]) to accelerate the implementation of wrappers, with final validation still performed by an expert analyst [13, 116, 140, 129].

5.4 Dataset Construction

Challenges

A persistent difficulty in malware research is the lack of authoritative ground truth for analyses of real-world samples. This shortcoming is acute for environment-sensitive malware: prominent studies that measure evasion or propose countermeasures [108, 88, 41] often draw conclusions from indicators collected during a single execution per sample. Without precise knowledge of the techniques embedded in each binary, such evaluations can reliably compare relative performance across systems but cannot conclusively assess the absolute effectiveness of any single method. Because manual reverse engineering remains the only dependable route to recover detailed ground truth, assembling a large, manually annotated dataset suitable for rigorous evaluation is costly and remains an open problem for the community.

To alleviate this shortfall and enable further research, we assembled a curated collection of 1078 PE32 samples exhibiting environment-sensitive behavior. Every sample in the set was inspected manually and is accompanied either by the catalog of behaviors (both evasive checks and post-evasion actions) revealed by our multi-path analysis or, when automatic analysis proved inconclusive, by the results of manual reverse engineering that explain what impeded automated exploration.

We selected the dataset size to balance the intensive labor of manual reverse engineering with the need for a substantively useful corpus. The final scale is comparable to prior state-of-the-art efforts that rely on manual review [41], while aiming for greater heterogeneity by avoiding the skew introduced by near-duplicate binaries—an issue recently highlighted in related work [58].

Collection

Samples were gathered from the wild between 2018 and 2023 using two principal sources. For the 2018-2021 window we relied on the VirusTotal Academic dataset, which contains binaries first observed in the wild during that period and flagged as malicious by at least 15 AV engines. For later years we used the Bazaar feed from VX Underground, which is refreshed monthly and curated by industry practitioners.

Starting from 315,231 initial entries, we applied a pipeline of filtering steps to retain a tractable, heterogeneous set of environment-sensitive samples suitable for manual curation.

The **first** filter removes near-duplicates via structural similarity. We use the *vhash* clustering value computed by VirusTotal to identify structural duplicates. After this de-duplication we retain 71,151 samples (including 2,908 that lack a computed vhash).

The **second** filter aims to focus our manual effort on binaries that exhibit static indicators suggestive of environment-sensitive behavior. Because not all malware depends on environment characteristics, and manual review is expensive, we prioritized samples that match known static patterns for anti-analysis; this approach mirrors prior work [41].[3] We proceeded as follows using public Yara rules [137] for anti-analysis detection:

1. Applying the rule set leaves 40,189 samples (56.48%) that match at least one rule;

2. We discard samples that match only anti-debugging rules, since debugger-specific evasions typically exploit implementation gaps in monitoring tools and are addressed in a one-off engineering effort; this reduces the set to 12,341 samples;

3. We refine the rule set to exclude patterns that target sandboxes or hypervisor fingerprints irrelevant to our testbed (we do not consider hypervisors other than VirtualBox in Section 5.5). The remaining set comprises 15 rules covering queries about drivers, files, firmware strings, hardware features, libraries, and processes, yielding 2,018 candidates.

The **third** filter removes samples written in managed runtimes. We statically detect managed-language binaries (*e.g.,* .NET) using [1] and exclude 163 such samples as done in other studies [88]. Analyzing managed code introduces orthogonal challenges (*e.g.,* tracking CIL-level coverage and filtering CIL-originating API calls), and therefore we focus on native binaries. After this step 1,855 samples remain.

The **fourth** filter reduces over-representation from prolific families. We cluster binaries into families with AVClass [118, 11] and cap the number of samples per family per year at 10. We also limit multiple occurrences of executables shielded by the same protector (using versioning when available) by matching protector fingerprints from [1]. After applying these family- and protector-based constraints, 1,262 samples from 239 families remain.

The **fifth** and final filter uses dynamic execution to catch cases missed by static heuristics. In this pass we identify 16 additional .NET samples and 46 samples that execute 64-bit code despite being labeled as 32-bit; we remove these spurious entries to preserve collection homogeneity. The dataset at this stage contains 1200 likely environment-sensitive samples.

[3]This choice can exclude environment-sensitive samples that evade the static patterns; the alternative would require manually inspecting many more samples, which is infeasible for our objectives.

Category	Samples
Fully evasive (and countered)	89 (72.95%)
Undetermined evasions	16 (13.11%)
Unable to start program	13 (10.65%)
Interactive	2 (1.64%)
External dependencies	2 (1.64%)
Total	122

Table 5.1. Themida-protected samples hitting one of two known cache coherency bugs in DynamoRIO.

During dynamic testing we also discovered 122 samples that trigger two specific DynamoRIO implementation defects. We reported these DBI cache-coherency bugs to the maintainers, but they remain unresolved (see GitHub issue #6567). All such samples were packed with the Themida protector using a particular configuration. For the experiments in this research we excluded these cases from our evaluation dataset.

To verify that these cases reflect an implementation shortcoming of our chosen DBI substrate rather than a conceptual limitation of our method, we implemented a compact prototype of our approach using Intel Pin. The Pin prototype focuses its fuzzing effort narrowly on the operations we observed in the baseline runs for these samples (notably `cpuid` and two variants of `RegOpenKey`).

Table 5.1 summarizes the outcome. We categorized the samples based on their outcome using the labels that we will use to cluster our results for the rest of this work, refer to Section 5.5.1 for the definition of these labels.

Under the Pin-based prototype, 72.95% of these Themida-protected samples yield to our method: we bypass their evasive checks and expose substantial conspicuous behaviors as judged by our evaluation metric. A further 13.11% of the samples remain evasive under this treatment and do not show noteworthy activity; these likely employ additional or more sophisticated evasion techniques that require deeper analysis. The final 13.99% fall into the same categories we labeled *Inconclusive* in RQ1 in Section 5.5.1—these are faulty executables, cases that need external resources, or samples that require human interaction.

These results support the conclusion that the Themida cases are principally an engineering-platform problem: once DynamoRIO's bugs are addressed, our full PFuzzer implementation should be able to handle them without methodological changes.

After the full pipeline, the final collection comprises 1078 samples, of which 138 are unlabeled and 940 are attributed to one of 239 families. Table 5.2 provides the

detailed breakdown of our dataset construction process.

		Step 1	Step 2			Step 3	Step 4	Step 5
Year	**Initial**	**Vhash**	**Yara**	**Skip dbg**	**Systems**	**.NET**	**Families**	**Dynamic**
2018	64 313	18 105	12 043	4 042	586	499	306	281
2019	59 104	12 989	7 588	2 854	510	480	315	295
2020	52 512	16 829	3 555	1 175	203	196	109	104
2021	25 176	6 189	3 841	1 946	362	354	242	213
2022	65 919	8 555	6 261	1 260	186	169	153	97
2023	48 207	8 484	6 901	1 064	171	157	137	88
Total	**315 231**	**71 151**	**40 189**	**12 341**	**2 018**	**1 855**	**1 262**	**1 078**

Table 5.2. Detailed Breakdown of the Dataset Construction Process, from the starting dataset obtained from VirusTotal and Bazaar feed to our final dataset.

5.5 Evaluation

We evaluate PFuzzer on the curated dataset to quantify its practical capabilities and to answer four research questions:

RQ1: Can our approach expose environment-sensitive behavior in real-world malware?
RQ2: What is the performance of PFuzzer compared to state-of-the-art solutions?
RQ3: How can PFuzzer build environments with the very characteristics samples are sensitive to?
RQ4: What alternative behaviors can we observe for a sample when analyzed in PFuzzer?

Analysis Environment. Our experimental setup follows contemporary best practices (*e.g.,* [96, 41]) to provide virtual machines that exhibit realistic system characteristics and wear-and-tear (applications, documents, and historical artifacts) so as to approximate a deployed host. All experiments execute on Windows 10, 64-bit, build 1803 with English locale. To avoid spurious failures caused by privilege restrictions, we disable User Account Control (UAC), enable the Administrator account, and run samples with elevated privileges. Network access from the analysis VMs is blocked; we emulate common network services locally using a REMnux VM running INetSim and PolarProxy to safely absorb and simulate outbound traffic.

The host machine is a physical server with an Intel Xeon E5-2620 v3 @ 2.40GHz, 48 GB RAM, running Linux OpenNebula (kernel 5.4.0). We provision five sibling virtual machines with the Windows configuration above, each assigned 2 CPU cores and 4 GB of RAM. Each sample analysis starts from a shared live VM snapshot captured with negligible background activity. We run the VMs inside VirtualBox

7.0.10r158379.

Conspicuous Activity. To quantify what each system exposes, we develop an automated metric that identifies *conspicuous activity*—runtime behaviors that can be attributed to malicious intent with high confidence. The goal of this metric is to approximate the semi-qualitative judgments analysts make when composing reports, and to avoid misleading, purely quantitative proxies such as raw API counts as in Enviral [58]—a slippery road—or to manual labeling of activity logs as in BluePill [41]—which is expensive and difficult to replicate.

Concretely, PFuzzer logs the identity and call site for a curated set of 146 APIs that can effectuate transient or persistent changes to the host—for the RQ2 head-to-heads we instrumented BluePill and Enviral in an analogous fashion so that all systems produce comparable API traces. From each execution trace (one for BluePill, multiple for the fuzzing systems) we extract the first occurrences of any externally observable system changes, distinguishing identical API invocations by their originating call sites.

We then prioritize a subset of these 146 APIs that are most strongly indicative of malicious behavior. For example, among service-related APIs we emphasize calls that create or start services rather than those merely querying status. This prioritization reduces noise from benign housekeeping calls and focuses the metric on actions with higher analyst salience.

Beyond single API hits, we attempt to attribute observed system changes to higher-level malicious behaviors (for instance, code injection, process/service launching, networking, data exfiltration, or spying). This attribution considers API call sequences (*e.g.,* memory allocation + write + remote thread creation for injection [44]), salient parameter values (*e.g.,* registry keys commonly abused for persistence), and recurrence or domain-specific patterns (*e.g.,* cryptographic operations together with mass file writes suggest ransomware-like activity). The output is a compact, reproducible list of conspicuous activities per execution that approximates the kind of semi-qualitative assessment an analyst would produce while remaining automated and comparable across systems.

The output of the metric is, for each run, a compact list of the conspicuous activities we detected over the execution. We use these per-run activity lists both to compare multiple runs of the same system (*e.g.,* to assign the RQ1 labels that characterize the additional behaviors PFuzzer uncovers) and to compare different systems head-to-head as in RQ2.

5.5.1 RQ1: Unveiling Additional Behaviors

RQ1: Can our approach expose environment-sensitive behavior in real-world malware?

We begin by measuring how often PFuzzer uncovers environment-sensitive behaviors that a single baseline execution on the same host misses. For each sample we allocate an overall analysis budget of 10 minutes. Individual runs are capped at 55 seconds (we justify this choice in Section 5.5.3), leaving up to 5 seconds per run for log extraction and snapshot restoration. The 10-minute budget reflects an upper bound typical of automated sandbox use by analysts and guarantees at least ten executions per sample (including the baseline); in practice we often perform many more runs because evasive paths frequently terminate early, which yields additional opportunities for mutation and re-execution (Section 5.5.3 reports measured counts).

Table 5.3 summarizes the dataset results by the *additional* conspicuous behaviors we observe while fuzzing. We classify each sample according to the most relevant conspicuous activity observed across all mutated-environment runs. Here, "conspicuous" follows the output of our evaluation metric (Section 5.5), which synthesizes transient and permanent system effects into analyst-oriented activity labels. We group samples into five qualitative categories:

- **No extra activity.** The baseline execution already exhibits conspicuous activity, and fuzzing does not reveal further conspicuous behaviors.
- **Mild additions.** The baseline shows conspicuous activity; fuzzing discovers additional conspicuous events that are new but do not clearly map to higher-level malicious behaviors beyond the baseline signals.
- **Strong additions.** The baseline shows some conspicuous activity, but fuzzing uncovers additional conspicuous behaviors that unambiguously indicate concrete malicious techniques (for example, code injection, keylogging, persistence, or new network connections) that were absent in the baseline run.
- **Highly evasive.** The baseline run produces no conspicuous activity, yet fuzzing is able to elicit conspicuous behaviors that can be confidently attributed to canonical malware actions.
- **Inconclusive.** Both baseline and fuzzed runs fail to surface significant conspicuous activity; fuzzing may explore alternative paths but none yields clear malicious effects.

The baseline run is the first 55-second execution performed by PFuzzer using an empty execution policy. Whenever fuzzing produces a candidate new activity, we validate it by extending the baseline execution to 8 minutes to ensure the initial short baseline did not simply miss a delayed behavior.

Across the corpus, samples in the **No extra activity** category represent 31.91% of the dataset. Even for these samples, PFuzzer adds value: Section 5.5.4 shows that for 70.64% of them we can still discover alternative execution paths that lead to evasion (for example, when an analysis tool is present); identifying these paths and their triggers is useful for analysts.

Samples classified as **Mild additions** comprise 13.17% of the dataset (142 samples). The most common newly revealed behaviors here are file operations (113 samples) and registry changes (37 samples), with a few cases involving system service modifications (3 samples). Such behaviors are unlikely to surface under single-run analyses.

Strong additions account for 15.68% of the dataset. These cases are particularly noteworthy because fuzzing frequently exposes substantive malicious techniques that are entirely absent in the baseline execution; Section 5.5.4 provides a detailed breakdown and examples.

Finally, the **Highly evasive** group comprises 13.54% of the corpus. Samples in this class produce little or no observable activity in the baseline execution (apart from occasional minor file operations discussed in Section 5.5.4), so a conventional single-run analysis would typically mark them suspicious but leave results inconclusive. By contrast, PFuzzer is able to coax many of these binaries into exhibiting clear malicious actions under mutated environment policies.

These results show that environment-sensitive logic in real-world malware is often not a single, irrevocable prelude to visible malicious actions. Instead, authors commonly embed conditional behaviors that may or may not be triggered by subtle environmental differences. As a result, single-run analysis systems frequently flag samples in our mild or strong additions categories as malicious, but their reports can miss substantial behaviors that only surface after further exploration or manual reverse engineering.

Overall, PFuzzer uncovers additional conspicuous behaviors for 42.39% of the corpus (457 samples). For 31.91% (344 samples) the baseline run produced no further observable actions; however, as Section 5.5.4 shows, this apparent inactivity can be misleading when deeper analysis is applied.

The remaining 277 samples (25.69% of the dataset) are classified as Inconclusive because our instrumentation and fuzzing did not surface conspicuous behaviors. To understand why, we manually triaged these cases and separated those in two groups:

	Category	Samples
	No extra activity	344 (31.91%)
457 (42.39%)	*Mild additions*	142 (13.17%)
	Strong additions	169 (15.68%)
	Fully evasive	146 (13.54%)
Inconclusive 277 (25.69%)	Cmd-line arguments	11 (1.02%)
	DynamoRIO bugs	19 (1.76%)
	External dependencies	51 (4.73%)
	PoC evaders	5 (0.46%)
	Interactive	49 (4.54%)
	Unable to start program	67 (6.21%)
	Undetermined evasions	75 (6.95%)
	Total	1078

Table 5.3. Breakdown of samples by additional behaviors emerged via fuzzing and confirmed by manual analysis.

(1) samples that fail due to limitations of PFuzzer and (2) sample that require capabilities outside PFuzzer's current scope.

In the first group we found 19 samples that trigger known DynamoRIO implementation issues (*i.e., DynamoRIO bugs* set) and 75 samples for which we have strong indications of evasive behavior (*i.e., Undetermined evasions* set) but where neither our multi-path runs nor available sandbox reports disclosed clear details about the employed techniques—they exhibit evasive behavior (*e.g.,* early termination after a few instructions) but whose root cause we could not determine with confidence. Taken together, these limitations implicate at most 94 samples (8.72% of the dataset).

The remaining 183 samples (17.07% of the dataset) fall into practical categories our current design does not handle:

- *Command-line arguments*: Some samples require specific arguments at launch. Using a debugger, we supplied arguments identified through manual reversing, which revealed code sections with interesting behaviors. Certain samples even supported multiple arguments, each unlocking different behaviors.

- *External dependencies*: These binaries depend on external resources not bundled in the PE file, such as DLLs (steam_api.dll, lib_game.dll), configuration files (*e.g.,* setup.cfg), or valid network connections (not covered by our simulator). Even when our method bypassed initial checks, missing or malformed dependencies often prevented execution from continuing.

- *Interactive behavior*: This category includes installers that require GUI-driven installation, samples that trigger malicious activity only after user interaction (*e.g.,* dismissing message boxes), and samples with GUIs that demand text input (*e.g.,* keygens).

- *Program failures*: Some binaries failed outright in our environment, for reasons such as application crashes, invalid Win32 mode executables, or incompatibility errors reported by the OS.

Addressing these cases would typically require orthogonal engineering (UI automation, richer dependency provisioning, or environment orchestration) and therefore lies outside the scope of this work.

Finally, we examined how PFuzzer's ability to reveal additional behaviors varies across malware families. For families represented in our filtered corpus, we observe that a large majority (69.04%) of families have their samples confined to a single behavioral category. When a family spans multiple categories, the most common combinations involve the *mild additions* and *strong additions* sets. At a glance, this pattern suggests that inter-sample variability within those families is often due to benign behavioral diversity or configurable functionality rather than purely to evasion; this observation aligns with prior reports of behavior variability in the wild [12, 23].

5.5.2 RQ2: Comparison Against State of the Art

RQ2: What is the performance of PFuzzer compared to state-of-the-art solutions?

We compare PFuzzer against two representative systems: BluePill and Enviral.

BluePill [41] represents the class of defenses that increase the transparency of the analysis environment by returning handcrafted, expert-derived fake answers to fingerprinting queries. Its authors report strong results on highly evasive samples and claim performance that rivals, and occasionally exceeds, commercial sandboxes in specific case studies.

Enviral [58] occupies an intermediate design point between PFuzzer and BluePill. Although presented as a fuzzer, Enviral effectively performs speculative, single-path evolution: it repeatedly replays the same execution path while injecting predetermined responses for known evasions and choosing random outputs for other queries. When a random choice yields a larger end-to-end API call trace, Enviral

keeps that choice, but it does not support backtracking to explore alternative decision points.

Across our benchmark, PFuzzer substantially outperforms both BluePill and Enviral. In particular, Enviral is the weakest performer in our experiments, a finding that contrasts with the claims in its original evaluation.

We measure comparative effectiveness using our conspicuous-activity metric and always compare each system's best run for a given sample (the run that exposes the most conspicuous activity). For each sample we inspect which environmental queries PFuzzer manipulated in that top-performing run. If a competing system lacks a mechanism to capture or interpose a particular query, we create an ablated variant of PFuzzer where that specific manipulation is disabled and use it as the fair comparator. This ablation step is required only for Enviral with respect to the `cpuid` and `rdtsc` instructions; for the API-based queries we observed, the three systems provide functionally equivalent interception capabilities.

Because both baselines natively target Windows 7, we construct a Windows 7 SP1 32-bit VM whose configuration mirrors the Windows 10 environment described above. For any sample where PFuzzer outperforms a baseline on the Windows 10 setup, we re-run the corresponding experiments on the Windows 7 VM to rule out rare, OS-dependent discrepancies and to confirm that the superiority is not an artifact of differing guest platforms.

We exclude the 183 samples from these head-to-head comparisons because they raise orthogonal issues (UI interaction, missing external dependencies, etc.) that none of the evaluated systems is designed to address. We did, however, verify that the analysis reports produced by the systems show no meaningful differences on those excluded samples.

BluePill. BluePill performs a single-execution analysis. It leverages Intel Pin for instrumentation, which in our experience exhibits higher startup and JIT latency than DynamoRIO. To provide a cautious and fair setting for BluePill, we allot it 4 minutes per sample—roughly four times the duration of a single PFuzzer execution—before evaluating its produced report. We then compare the amount of conspicuous activity PFuzzer finds for each sample against what BluePill reports according to our metric.

Table 5.4 reports the comparison results. Across 895 samples, PFuzzer surfaces more conspicuous activity than BluePill in 323 cases (36.09% of the set), while it finds less activity for only 14 samples (1.57%).

To clarify the practical advantage of PFuzzer, the table further breaks down outcomes by the RQ1 labels. Most ties occur within the *No extra activity* category, where PFuzzer did not need to counter evasive checks. Notably, however,

Category	Better	Equal	Worse
No extra activity	28	315	1
Mild additions	66	76	0
Strong additions	135	34	0
Fully evasive	94	51	1
Undetermined evasions	0	70	5
DynamoRIO bugs	0	12	7
Total	323	558	14

Table 5.4. Activity found: PFuzzer vs. BluePill.

BluePill's strategy of always returning predetermined fake answers causes a *regression* in 28 of these tied samples: forcibly modifying the normal execution at every suspicious point can actually harm the analysis of some binaries[4].

PFuzzer delivers especially strong gains on samples we categorized as *Strong additions* (outperforming BluePill on 135 of 169 samples) and *Fully/Highly evasive* (better on 94 of 146). We do not observe any clear dependence of these results on the year the samples were first seen, so obsolescence of BluePill's rule set seems an unlikely explanation. Rather, we attribute PFuzzer's advantage to two design benefits: (1) fixed, expert-derived fake responses—however accurate—may not match every sample's expectations, and (2) attempting to cover every suspicious query with such canned answers can be counterproductive.

BluePill exceeds PFuzzer on a small number of cases: 5 samples labeled as *Undetermined evasions*, seven of the 19 samples affected by DynamoRIO implementation bugs, and two additional samples scattered across the other categories. For the 5 *Undetermined* cases we suspect DBI-related interactions that BluePill (via mitigations available for Pin) neutralizes or that specifically affect DynamoRIO.

Enviral. For Enviral we allocate the same overall 10-minute budget used for PFuzzer. We configure each individual run to last 25 seconds—a tenfold increase over the short 2.5-second runs used in Enviral's original evaluation [58]. We demonstrated and we will show in Section 5.5.2 that 2.5 seconds is insufficient to capture meaningful behaviors. While its authors granted BluePill 4x longer runs to offset DBI overheads, for the same imbalance we give PFuzzer only 2.2x more time (55 seconds) per run than Enviral (incidentally, this may potentially let Enviral attempt many more executions).

The left portion of Table 5.5 summarizes the head-to-head results. Across 895 samples, PFuzzer uncovers more conspicuous activity than Enviral in 393 cases (43.91% of the set—a 7.82 percentage-point improvement compared to the BluePill comparison) and finds less activity for 22 samples (2.46%). As before, we break

	PFuzzer			*Ablated PFuzzer*			
Category	**Better**	**Equal**	**Worse**	**B.**	**E.**	**W.**	**Σ**
No extra activity	45	285	14	46	288	17	+7
Mild additions	108	34	0	103	34	0	-5
Strong additions	150	19	0	137	19	0	-13
Fully evasive	90	54	2	68	54	2	-22
Undet. evasions	0	72	3	19	80	9	+33
DynamoRIO bugs	0	16	3	0	16	3	-
Total	393	480	22	373	491	31	-

Table 5.5. Activity found: PFuzzer vs. Enviral.

these outcomes down according to the RQ1 categories.

Notably, Enviral produces substantial regressions on the subset of samples we labeled *No extra activity* under PFuzzer: for 45 of those samples (13.08% of that group) Enviral reports less activity than the baseline PFuzzer run. This pattern indicates that Enviral's interventions can perturb execution in a way that drives the sample down evasive paths. We suspect this regression is due to a combination of factors: instrumentation and runtime overheads, the conspicuousness of its trampoline-based hooks, and possible implementation bugs.

On the other hand, Enviral does reveal additional activity in 22 samples from that same group; inspection of the logs shows these newly seen actions align with our RQ1 definition of *Mild additions* (this time as exposed by Enviral). A likely explanation is that, because Enviral can perform far more runs within the allotted time (thanks to its shorter per-run interval and lower instrumentation burden), it is able to stumble upon some of these paths—paths that PFuzzer would likely also find given more time.

Overall, PFuzzer overwhelmingly outperforms Enviral for the *Mild* and *Strong* categories (better on 76.06% and 88.76% of their samples, respectively). We trace this advantage to a core design difference: Enviral focuses on evolving a single high-promise path without backtracking, which limits its ability to explore alternative decision points. By contrast, PFuzzer's scheduler (energy and significance scores; Section 5.3.3) explicitly redistributes effort across multiple promising directions and leverages a richer feedback signal that combines external queries, system-changing actions, and internal coverage — improving sensitivity to incremental progress.

For the *Highly evasive* set, Enviral can match PFuzzer for the subset of samples (36.99%) where its predefined expert answers happen to be sufficient; in those cases its behavior resembles that of BluePill, relying on canned responses to

bypass common checks.

To rule out an implementation-specific limitation of ENVIRAL, we run an extra experiment in which we disable in PFUZZER the instrumentation of assembly instructions (`cpuid` and `rdtsc`) that ENVIRAL's Detours-based approach cannot intercept. The right side of Table 5.5 reports the results of this ablation. Because this change reduces PFUZZER's capabilities, we recompute the labeled sets accordingly, so both the size and membership of each category can change.[5] The overall trends remain unfavorable to ENVIRAL, leading us to conclude that the lack of `cpuid`/`rdtsc` interception is not the primary reason for ENVIRAL's comparatively poor performance in our study. While ENVIRAL's hybrid strategy of expert answers plus speculative single-path exploration is conceptually appealing, the evaluation issues discussed in Section 5.5.2 and the experimental outcomes above indicate that additional improvements are needed.

Sandboxes. Some readers may ask why we did not include commercial sandboxes in our head-to-head comparisons. Although the *Fully evasive* subset (146 samples) might be appropriate for such a comparison—because their malicious behavior appears only when a sandbox successfully counters evasions—most of the remainder of our corpus requires environment perturbations unrelated to classic anti-evasion techniques to reveal additional activity. Off-the-shelf sandbox designs typically lack mechanisms for automatically and flexibly tweaking these broader environmental aspects, which were key to discovering extra behaviors in samples that already showed maliciousness in baseline runs.

For completeness, we evaluated the *Fully evasive* samples with a prominent open-source sandbox that includes anti-evasion features: CAPE. We excluded commercial products because their closed implementations would hamper interpretability and licensing often forbids benchmarking.

CAPE labels 67.58% of the *Fully evasive* samples as malware, 20.68% as suspicious, and 11.72% as benign. Note that CAPE incorporates static signatures and additional techniques (for example, in-memory inspection) that influence its classifications; nevertheless, it misclassifies roughly one out of three of these *Fully evasive* samples. In many cases the API traces it records contain fewer observable behaviors than either PFUZZER or BLUEPILL capture, suggesting that its effective behavioral coverage is lower than the headline classification numbers might imply.

[4]We observed a few subtle anti-patterns; for example, BLUEPILL sometimes shortens delay operations that are necessary for correct thread completion.

[5]The *Undetermined evasions* category acquires 33 additional samples under the ablation: 22 were previously *Fully evasive*, 5 were *Strong additions*, and 6 were *Mild additions*. Among other shifts, eight samples that were formerly *Strong additions* move to other categories: one becomes *Mild additions* and seven move to *No additions*.

Evaluation Issues of Enviral

While experimenting with the systems used in our comparison, we identified some issues in the experimental methodology used by the authors of Enviral when they compared their system against BluePill. The original Enviral evaluation considered 338 likely-evasive malware samples from 2019 taken from the VirusTotal Academic corpus. To reproduce a comparable analysis, we ran experiments on the 220 samples from 2019 that survived our own dataset-construction pipeline, using a Windows 7 VM as in the Enviral study.

When reproducing the workflow (using the modified BluePill binary supplied by the Enviral authors), we discovered three principal issues that, taken together, invalidate the claim that Enviral clearly improves upon BluePill:

1. a clerical mistake introduced when adapting BluePill to intercept execution termination;
2. an unsound per-run time budget for BluePill that fails to account for DBI warm-up costs; and
3. an evaluation metric that is coarse-grained and prone to distortion.

Below we explain each point and show how they affect the reported conclusions.

Clerical Error. The Enviral authors changed BluePill's exception handling so that any Windows exception caught by Pin's exception handler caused the instrumented run to be terminated. This change is problematic for two reasons. First, not every exception signals an execution failure—some exceptions are deliberately triggered by malware as anti-debugging maneuvers, and the normal machine behavior is to let the program handle them. Second, BluePill's original handler contains logic to deal with specific exceptions in a way that preserves normal execution semantics. By terminating runs on exceptions, the modified BluePill abruptly loses a substantial portion of observable activity, artificially reducing its measured effectiveness and biasing comparisons in favor of Enviral.

Unsound Budget. The Enviral evaluation limits BluePill to 10 seconds per run (15 seconds in a verbose logging mode). Such tight caps are ill-suited for DBI-based instrumentation: the first seconds of an instrumented run are dominated by JIT compilation and code-cache population, so short budgets systematically undercount what a DBI-equipped system can observe. In our experiments we found that increasing BluePill's per-run budget by only a few seconds caused many of the apparently Enviral-only behaviors to appear in BluePill's logs—not because Enviral had superior anti-evasion logic, but because BluePill had been starved of runtime to amortize its instrumentation overhead. To produce a fairer head-to-head

Timeout (s)		Settings		Geomean Activity			Per-sample Comparison		
Env	BP	Verb.	Err.	BP	Env	Δ (%)	BP>Env	Env>BP	Eq.
2.5	10	Off	Yes	1.07	1.14	6.54	85	54	81
2.5	10	Off	No	1.03	1.14	10.32	61	52	107
2.5	15	On	Yes	0.89	1.14	28.54	146	35	39
2.5	15	On	No	1.14	1.14	0.10	193	7	20
10	30	Off	Yes	1.13	1.23	8.67	114	47	59
10	30	On	No	1.29	1.23	-4.77	185	15	20
25	120	Off	Yes	1.14	1.25	9.94	104	58	58
25	120	On	No	1.24	1.25	0.84	182	15	23

Table 5.6. Comparison of **BluePill** (BP) and **Enviral** (Env) on the 220 samples from 2019. Columns: per-run time budget (Env/BP, in seconds), logging configuration (*Verbose*), presence of the exception-handling modification clerical error (*Error*), geometric mean of activity scores (as in [58]), and per-sample head-to-head counts (*BP>Env*, *Env>BP*, *Equal*).

comparison we experimented with multiple budget settings (Table 5.6); under more reasonable per-run durations the two systems produce closer results for the activity metric used by ENVIRAL's authors.

Imperfect Metric. ENVIRAL's evaluation metric is the geometric mean of the activity increase relative to a baseline, where activity is the raw count of "noisy" API invocations. This design is fragile for two main reasons. First, total-volume counts can be inflated by adversarial constructs that deliberately drive executions toward dead-end or noisy code (producing large API counts without revealing meaningful malicious effects). Second, the implementation does not discriminate the origin of an intercepted call (whether it stems from the sample or from OS/library code), and it double-counts some higher-level events by including both wrapper APIs and their underlying system calls (for instance, counting both `CreateProcessInternalW` and `NtCreateUserProcess`). In short, the metric rewards raw activity volume rather than analyst-relevant, conspicuous actions. For our research we therefore adopt a metric focused on externally conspicuous behaviors (transient or persistent machine-state changes attributable to the sample), which we use throughout the main evaluation.

Experimental Results. Table 5.6 summarizes experiments on the 220 reproduced samples under multiple configurations (varying per-run time budget, verbose logging, and whether the clerical exception-handling change was present). The data show that a 25-second run budget yields the largest geometric-mean activity increase for ENVIRAL, which motivated our choice of per-run durations in the RQ2 experiments. However, even under that configuration the apparent geomean advantage of ENVIRAL over BLUEPILL is marginal (1.25 vs. 1.24), and a per-sample comparison of API

counts reveals that the geomean metric is misleading: only 15 samples show more activity under Enviral, whereas BluePill shows more activity on 182 samples and the two systems tie on 23.

Using our conspicuous-activity metric instead, neither system exposes relevant behavior for 39.09% of the samples. Among the remainder, the two systems are tied on 13.64%, BluePill dominates on 32.72%, and Enviral on 14.55%. These results corroborate that the original Enviral claims of clear superiority over BluePill do not hold under careful, fair comparison criteria.

5.5.3 RQ3: Behind PFuzzer's Performance

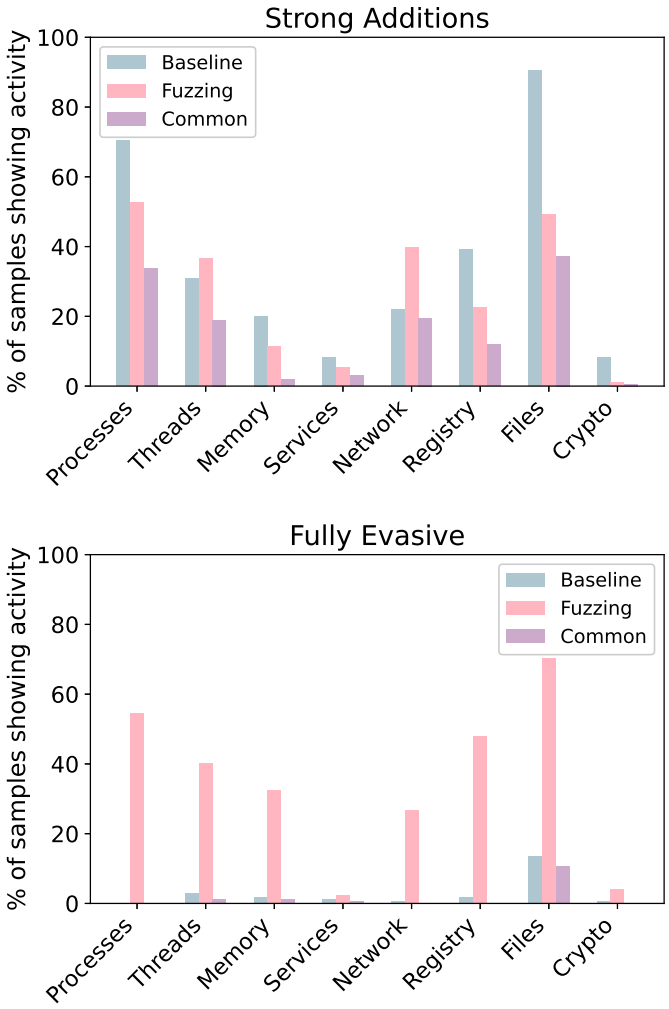
RQ3: How can PFuzzer build environments with the very characteristics samples are sensitive to?

This section investigates how PFuzzer progressively synthesizes execution policies that satisfy the environment-dependent conditions samples check for. We instrumented and collected operational metrics for the 457 samples (Table 5.3) for which PFuzzer discovered additional behaviors, and report here on one key tuning knob: the maximum allowed duration per run.

Time Sensitivity. Given our per-sample wall-clock budget of 10 minutes, we studied the effect of varying the per-execution cap by 25 seconds from the default. Concretely, we compare a shorter run length (30 seconds) and a longer one (80 seconds) against the chosen default of 55 seconds (plus 5 seconds for post-processing). The default allows at least 10 executions per sample within the budget; with the shorter cap we can perform at least 17 runs, and with the longer cap at least 7 runs.

Using the conspicuous-activity metric from Section 5.5 to compare results, the 30 seconds configuration reproduces the default outcome for 298 samples (65.21%) but yields strictly less conspicuous activity for 158 samples (34.57%). In other words, despite allowing substantially more executions, the shorter per-run interval is often insufficient to exercise the behaviors of interest. Only one sample benefited from the extra iterations: it moved from the *Mild additions* set to the *Strong additions* set after PFuzzer could continue evolving a promising policy in the additional runs.

Conversely, the 80 seconds configuration produces the same outcome as the default for 315 samples (68.93%) and less activity for 141 samples (30.85%), indicating that extending single-run time beyond 55 seconds yields little additional benefit on our corpus. We observed a single exception where the longer run uncovered new network activity absent at 55 seconds. These findings align with observations in prior work that selecting an appropriate single-run horizon is delicate (*e.g.,* [75]), but that many observable behaviors appear within the first couple of minutes. In our dataset, the



	Number of samples	
Behavior	*Strong add.*	*Fully evasive*
Communication	66	45
Data wiping	69	40
Encryption	2	7
Injection	6	42
Launching	91	55
Persistence	46	92
Spyware	8	14

Figure 5.2. Additional activities: type of operations (top) and manually identified behaviors (bottom) under the best execution policy *w.r.t.* baseline run.

55 seconds cap provides a practical sweet spot: it captures sufficient per-run novelty to drive effective policy evolution while allowing a useful number of iterations under

		Mild additions	*Strong additions*	*Fully evasive*
Total trials in 10' budget		43.5	47.2	103.7
Policies bringing novelty		18.85	18.73	29.06
Trials until first novelty		6.18	4.06	4.6
Trials until best policy		16.23	17.51	29.22
Mutations in best policy	Normal course	4.09	3.26	2.56
	Force success	2.26	1.78	0.95
	Force failure	2.01	1.76	1.00
	Retarget	5.75	5	3.73
	Alter time	0.61	0.50	0.42

Table 5.7. Average internal metrics of PFuzzer across the three behavior groups in RQ3 (*Mild additions*, *Strong additions*, *Fully evasive*), under the default 55 s per-run configuration. The table reports how the 10-minute budget is spent (total trials, trials until first novelty and best policy), how many distinct policies yield novelty, and the composition of the most effective policy by mutation type.

the fixed budget.

Internal Fuzzer Metrics. We next examine how PFuzzer spends its allotted time budget: how many executions it performs, how many policies it generates, and how exploration progresses. Table 5.7 aggregates the measured internal metrics, averaged per-sample within each of the three groups considered in this RQ3 analysis.

A first notable observation is that PFuzzer issues substantially more runs than the theoretical lower bound of ten dictated by the budget. This increase is chiefly driven by early termination of executions: many samples—particularly those in the *Fully evasive* category—exit before the per-run timeout, and in other cases PFuzzer itself induces early exits by triggering evasive behavior that aborts an execution (behavior that, by construction of the group labels, is absent in the baseline run). Less frequently, runs end early because the sample injects code into another process or spawns a child that continues execution outside the monitored context.

On average, PFuzzer retains roughly 18 distinct execution policies for samples in each of the first two groups, and about 29 policies for *Fully evasive* samples. Each retained policy corresponds to a configuration that produced novel external or internal coverage relative to all prior attempts. For the first two groups these policies capture both additional malicious activity and evasive paths that PFuzzer discovered; the larger policy count for the *Fully evasive* group reflects the multiplicity and interleaving of evasive techniques commonly observed in the wild [51, 88]. When such activity is uncovered it is particularly valuable: the fuzzer keeps the producing policy in the queue and leverages it to pursue further mutations that may satisfy subsequent environmental dependencies.

Driving mutations from external coverage (*i.e.,* the set of environmental queries observed; see Section 5.3.2) proves effective: PFuzzer typically requires only a small number of trials—on the order of 4-6 attempts on average—before it discovers new coverage. This efficiency is important because each trial is relatively expensive (unlike conventional software fuzzing we cannot run thousands of executions per sample), so quick discovery of novelty materially improves overall throughput. To reach the particular configuration that yields the most conspicuous activity, PFuzzer needs on average roughly 16-18 trials for samples in the two additions groups and about 29 trials for *Fully evasive* samples, the latter figure plausibly reflecting the greater complexity of the environmental expectations those samples encode. Crucially, these per-configuration trial counts are far smaller than the total number of runs performed, indicating that PFuzzer typically discovers fruitful configurations well before exhausting its time allotment.

Turning to the content of policies, Table 5.7 reports the number and types of mutations present in the execution policy that produced the most activity for each sample. Two patterns stand out. First, *retargeting* an operation—substituting a different, semantically plausible value for a queried item—is the most frequent and valuable mutation on average. This suggests that many environment-sensitive checks hinge not merely on success versus failure, but on particular returned values. Second, leaving other accesses unmodified is also important: indiscriminately altering every query is often unnecessary and may be harmful, either by triggering evasive checks that are irrelevant to the current machine (as we observed with BluePill in Section 5.5.2) or by breaking legitimate program logic. Unlike existing approaches, PFuzzer can discriminate which manipulations to keep based on their measurable effects on execution via the combined coverage feedback, allowing it to retain beneficial mutations and discard detrimental ones automatically.

Environmental Artifacts. Beyond measuring how PFuzzer achieves its gains, a question of practical interest for defenders is which environmental attributes malware most often depends on. We inspected execution logs to enumerate the query classes that PFuzzer most frequently manipulated when producing the policy that yielded the greatest amount of newly observed activity. Table 5.8 summarizes these findings for the *Strong additions* and *Fully evasive* subsets, which we prioritize because they produced the richest set of revealed behaviors. Across both groups, the dominant categories of critical checks are file-system probes, existence checks on registry paths, and time-related queries.

For *Fully evasive* binaries, the conspicuous VM- and analysis-related probes spanned files, registry keys, GUI objects, running processes, and hardware identifiers. We observed lookups for AV and monitoring product artifacts, queries for

	Fully evasive			*Strong additions*		
Category	**Rank**	**Samples**	**Avg #**	**Rank**	**Samples**	**Avg #**
File metadata/contents	1	88.36%	5.02	1	92.31%	5.78
Open registry key	2	76.03%	3.93	2	52.07%	4.24
Time/delay check	3	69.86%	1.57	3	63.31%	1.80
Query registry key	4	49.32%	1.90	7	21.89%	1.68
Processes	5	54.79%	1.26	5	51.48%	1.25
GUI elements	6	36.99%	1.74	6	22.49%	2.21
Mutex creation	7	45.89%	1.30	4	56.21%	1.46
Username	8	21.92%	1.31	9	8.88%	1.67
CPU details	9	16.44%	1.00	10	13.02%	1.00
Hostname	10	10.27%	1.40	8	20.71%	1.51

Table 5.8. Types of queries altered in the policy with most activity (counts '#' are averaged over affected samples).

registry values such as ProductID associated with online sandboxes, and searches for GUI and filesystem cues that analysts typically modify by hand (for example, the executable filename or its .exe extension, installation path, working directory, keyboard layout, and presence of particular installed applications). We also encountered checks for Windows features (*e.g.,* Active Setup) and Windows Defender settings—including attempts to tamper with Defender—phenomena that appear underreported in prior surveys [51, 88]. Finally, some checks targeted files or registry keys that a previous-stage payload would be expected to create, indicating multi-stage deployment logic.

Samples in the *Strong additions* class interrogated a broader and more heterogeneous set of environmental aspects, often switching tactics depending on unrelated system conditions or supplemental components. One illustrative pattern was probing for indicators of exploitable configuration (for example, the presence of an acsipc_server named pipe) that would enable privilege escalation or quieter operation. Although many of these binaries also perform classic evasive queries, those checks did not trigger in our baseline environment, so the malware exhibited its malicious functionality from the first run; the fuzzer's mutations then revealed complementary, non-evasion-driven behaviors.

5.5.4 RQ4: Insights from Alternative Paths

RQ4: What alternative behaviors can we observe for a sample when analyzed in PFuzzer?

In this section we analyze, from several perspectives, the alternative execution

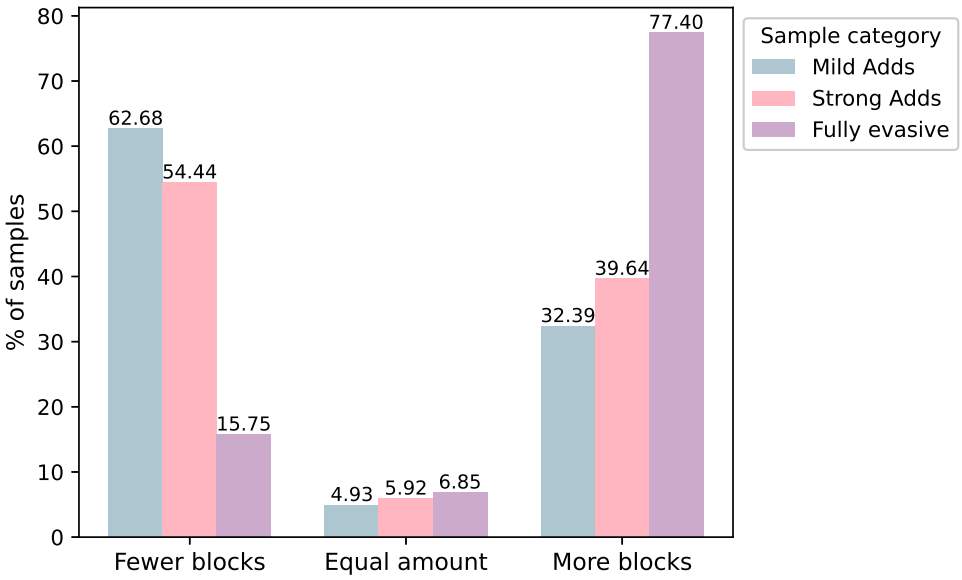


Figure 5.3. Distinct basic blocks that the sample traverses under the first execution policy unveiling new conspicuous activity *w.r.t.* baseline run.

paths that PFuzzer uncovers. Our primary focus remains the 457 samples examined in RQ3 for which PFuzzer revealed additional behaviors; in addition, we separately inspect the 344 samples classified as *No extra activity* in RQ1, since—as we will show—many of those binaries nevertheless conceal dormant evasive branches that can be triggered under different environment conditions.

Code Coverage. To investigate how alternative environment policies change the code exercised by a sample, Figure 5.3 reports the change in unique basic-block coverage between the baseline run and the first fuzzed run that produces novel conspicuous activity. Results are broken down by the three groups studied.

For the *Mild additions* group (142 samples), the run that first reveals novelty traverses fewer distinct basic blocks than the baseline in 62.68% of cases (89 samples). This pattern commonly reflects the sample taking activity alternative to what the baseline exercised, or the fuzzed run interleaving new behavior with baseline actions but timing out before completing all of them. PFuzzer detects these meaningful differences because its novelty judgement depends on the identity of coverage entities (API name plus call site), not merely on raw block counts. In a small number of cases (7 samples) the block count matches the baseline but one or more block identities (or associated dataflow) differ; in the remaining 32.39% (46 samples) the novel run actually covers more basic blocks.

The *Strong additions* cohort (169 samples) shows a similar distribution, though with a somewhat larger fraction of cases where the novel run covers more blocks:

39.64% (67 samples) fall in that category. Our interpretation mirrors the *Mild* group—novel behaviors can either replace baseline behaviors or extend them, producing either fewer or more blocks depending on timing and the runtime decisions taken.

Fully evasive samples (146) display a different pattern: the most common outcome is that the fuzzed run traverses substantially more basic blocks (77.40% of cases). This aligns with our expectation that these binaries often throttle or terminate when they detect an analysis environment in the baseline run; by satisfying environment-dependent checks via mutations, PFuzzer enables the sample to continue into code regions that the baseline never reaches.

Types of Activities. We first catalog the kinds of newly observed actions. For the *Mild additions* cohort, as anticipated in Section 5.5.1, the bulk of the newly surfaced behaviors involve file-system operations (113 samples) and registry modifications (37 samples), with a small number affecting system services (3 samples). These additions typically consist of extra filesystem writes/reads or registry keys being created or altered—side effects that single-run analyses often miss but that nevertheless reflect concrete interactions with the host state.

For the other two groups we collect finer-grained indicators to describe the new activities PFuzzer uncovers. We partition externally observable operations that PFuzzer records into functional categories and count, per category, how many samples exhibit operations of that kind. The three bars per group in Figure 5.2 indicate, for each category, the number of samples where (1) the baseline run already shows operations from the category, (2) the best execution policy reveals new operations from the category, and (3) operations from the category appear in both baseline and fuzzed runs (*i.e.,* they are not exclusive to fuzzing).

The third bar is particularly revealing for the *Strong additions* set. In many of these samples PFuzzer does not merely add completely new categories of activity but uncovers operations within the same category that complement or replace those observed in the baseline run. A single-run analysis would miss these supplementary facets because they depend on environment-specific conditions. For example, we observed a sample that changes its protocol handshake with its C&C server depending on environment cues, and another that chooses between code injection and spawning a new process based on whether the user account has administrative privileges.

Conversely, the samples that contribute only to the fuzzing bar (and not to the "both" bar) demonstrate categories that never surface in the baseline run at all; these are entirely missed by single-execution analyses. This pattern is especially marked for the *Threads* and *Networking* categories, where many more samples exhibit such activity under mutated environments than during the baseline execution.

That dynamic becomes even more pronounced for the *Fully evasive* samples: by design these binaries suppress conspicuous actions when they detect an analysis environment, so only through fuzzing are we able to enable and observe behaviors across the various categories (file manipulations being a partial exception, since many benign programs also touch the filesystem).

To conclude, we manually inspected the additional recorded operations and classified those we could attribute with confidence to canonical malware behaviors. Figure 5.2 summarizes this attribution for the two groups. In the *Strong additions* set the most recurrent malicious behaviors we see are process/service launching, data wiping, and remote communications. In the *Fully evasive* set persistence is the dominant behavior (consistent with evasive logic aimed at protecting conspicuous payloads), followed by launching, remote communications, and code injection — each of which carries a clear malicious footprint.

Paths and Queries. To quantify how many alternative execution paths PFuzzer discovers for a given sample, we convert the recorded external-coverage activity into a graph structure (generally a forest, although in practice often a tree). We construct two variants of this graph: one that includes only actions that effectuate changes to the machine state, and another that also incorporates the observed environmental-information queries.

Using the state-change-only graph, the average number of distinct alternative paths is comparable across the three groups: 7.76 for *Mild additions*, 6.46 for *Strong additions*, and 7.58 for *Highly evasive* samples. When query events are included, the counts diverge: the first two groups remain similar (respectively 12.29 and 9.99), while the *Highly evasive* group shows a much larger figure (22.4). This increase reflects that many samples traverse slightly different query sequences before reaching a state-changing action. The substantially greater number of paths for *Fully/Highly evasive* samples is consistent with multi-stage, environment-sensitive checks: each stage can introduce branching that multiplies feasible execution trajectories. Single-run analysis approaches are likely to miss this rich set of alternative, plausible executions for a single binary.

Samples with No Extra Activity. Finally, we inspect the 344 samples that exhibited conspicuous activity during the baseline run but for which PFuzzer did not surface additional conspicuous behaviors. At first glance these might appear non-evasive; however, for 243 of them (70.64%) our method is able to reveal *silent* evasive branches—*i.e.,* environment-triggered paths that are not taken in the baseline but can be induced by altering specific environmental interactions. The remaining 100 samples are either static-analysis false positives or contain evasive conditions our current experiment did not trigger.

For the 243 samples where silent evasions are present, we observe on average 2.06 baseline accesses to environmental information that, when modified, redirect the execution onto an evasive path. Across the exploration of these samples we see on average 7.43 distinct paths per sample; of those, early termination-type paths occur more frequently (1.39 on average) than explicit stalling strategies (0.88 on average). Considering the baseline executions produce on average 96.02 external-coverage events (queries or system-state changes), the first access whose alteration produced an evasive outcome typically appears after 15.22 such events. This observation is consistent with the view that evasive checks commonly appear early in execution, although they are not restricted to the very beginning of runtime.

Novel Evasive Techniques Found. During manual follow-up of PFuzzer's outputs we identified four evasive techniques that, to our knowledge, were not previously documented publicly; our analyses were contributed to the Unprotect evasions knowledge base [128]. The first technique detects VirtualBox by enumerating network-shared resources rather than using the well-known `WNetGetProviderName` API to obtain conspicuous provider names. The second reveals VirtualBox-attached disk devices by issuing a SMART_RCV_DRIVE_DATA I/O control request to the driver instead of relying on conventional serial-number or model queries that detection systems may easily flag. Two additional techniques were recently accepted and confirmed as new. One detects the presence of the Cuckoo sandbox (a widely used open-source automated malware analysis system) by waiting for a named pipe used for communication between the analysis host and the virtualized guest: the sample polls a pipe with a fixed name (*i.e.,* \\.\PIPE\CUCKOO) and, if the pipe becomes available, concludes that Cuckoo is present. The other technique exploits an unintended use of a mechanism introduced in Windows 8 that allows a process to request lightweight termination on critical failures; by probing for this functionality an attacker can infer whether the running system is Windows 8 (or newer) or an older release such as Windows 7. Attackers still target Windows 7 because it remains unsupported (no standard security updates after Jan 14, 2020 [94]), a measurable number of devices still run it, and legacy systems are both common and hard to upgrade—creating a high-reward, low-cost attack surface exploited in recent zero-day and ransomware campaigns.

This chapter presented PFuzzer, a coverage-guided environment fuzzer that exposes hidden behaviors in environment-sensitive malware. By leveraging dynamic feedback and systematic environmental mutation, pfuzzer enhances behavioral visibility while maintaining practical scalability. The next chapter builds upon this philosophy by addressing a complementary problem: *API hashing*, where malware conceals semantics at the code level.

Chapter 6

Conclusion and Future Directions

Malware analysis has evolved into a race between defensive transparency and adversarial concealment. As analysis systems have grown more sophisticated—combining dynamic instrumentation, virtualization, and large-scale automation—malware has responded with equally sophisticateed *anti-analysis* strategies.

This thesis investigated how anti-analysis mechanisms in malware—traditionally considered obstacles—can instead be leveraged as analytical opportunities. By directly engaging with adversarial logic rather than attempting to remain invisible, we proposed two complementary dynamic analysis methods that broaden behavioral visibility and semantic understanding of evasive malware.

Summary of Contributions

In Chapter 4, we turned to a different but equally evasive challenge: *API hashing*, a code-level anti-static tehcnique that replaces plaintext imports with hash constants to obscure dependencies. Here, we developed a dynamic, slice-guided approach to identify, delimit, and interpret the hashing logic directly within the malwares binary, and we extended it into a *hash oracle* capable of reproducing hash outputs for arbitrary strings. This enables the automated recovery of API mappings without manual reverse engineering of the hashing function, ensuring a one-to-one correspondence between the recovered and original values.

In Chapter 5, we introduced PFuzzer, a novel system for multi-path exploration of environment-sensitive malware. Our key insight was that many evasive samples exhibit additional, semantically distinct behaviors under small environmental perturbations—behaviors typically missed by single-run sandboxing. PFuzzer automatically mutates environmental features (*e.g.,* registry, file system, timing,

system) while using an ad-hoc coverage-guided feedback to prioritize executions that expose new program paths. We demonstrated that PFuzzer can unveil latent malicious activity and recover behaviors obscured by virtualized or sterile sandbox conditions.

Taken together, the two lines of research reinforce a common thesis: dynamic visibility remains the most adaptable weapon against evolving anti-analysis. Rather than relying on full-system transparency—an elusive goal when facing virtualization-aware malware—our work demonstrates the value of *selective* introspection: focusing analytical effort on the regions of code and context most responsible for concealment.

At a conceptual level, PFuzzer reframes environment-sensitive malware not as a nuisance to be hidden from, but as a dynamic system that can be systematically explored. Similarly, the API hashing work treats the malware's own code as an oracle, turning the adversary's obfuscation mechanism into an analytical resource. Both perspectives exemplify a shift from *defensive mimicry* (pretending to be native) to *active engagement* (provoking and exploring evasive logic).

Practical considerations

The techniques developed in this thesis are implemented as dynamic-analysis tools built on top of DynamoRIO and related frameworks. PFuzzer currently targets 32-bit Windows PE samples and is designed to analyze malicious files, which can be directly integrated in a sandbox systems to provide a different type of analysis compared to commercial sandboxes, which only provides scores and results based on a single observation. We think that adding this kind of analysis as a service could improve the analysis process by a lot.

The API-hashing deobfuscation tool similarly can be easily integrated as part of a sandbox system to provide more details on a malicious file. Moreover, since this technique is typically performed at the beginning of the execution, a light-weight version could be integrated on more deep screening processes by AV companies that receives tons of files per-day to distinguish to flag files as possible malicious. Using API-hashing is not directly related to malware, a proprietary software could be interested in hiding its internal functionings, but paired with other scores could provide useful insights of this kind.

Artifact Availablity

PFuzzer code is not publicly available because it is currently owned by a company that has patented it. However, we provide the evaluation material at `https://github.com/Sap4Sec/pfuzzer/`, and we plan to share access upon request with other researchers who wish to experiment with it in a live setting.

Regarding our API-hashing tool, it is still under active development and evaluation, and we therefore do not release it at this time. Once the implementation is complete, we will make the code available at `https://github.com/Sap4Sec/`.

Future Works

While both proposed approaches proved novel and effective, they still face limitations discussed in Chapter 4 and Chapter 5. These constraints indicate that the problems addressed in this thesis are not yet fully solved. Future research may further refine the methodologies discussed here, extend their applicability to new domains, and explore complementary techniques to overcome current limitations or enhance the analysis.

Advancing API Hashing Analysis. Regarding the *API hashing* contribution, a natural next step is to conduct a complete and deeper analysis on our expanded dataset that better captures the diversity of hashing schemes encountered in the wild (see Section 4.4.2). Such large-scale evaluation would enable the generation of comprehensive hash→name mappings across malware families and facilitate systematic characterization of the various facets of this obfuscation technique.

Apart from improving the evaluation by deeply studying a broader set of samples, an improvement to prove our results and provide more insights would be perform a longitudinal evaluation to see how many samples the difference between solving and not solving the analysis statically is due to the API hashing. If samples apply other tricks that still require dynamic inspection, it does not avoid sandbox execution. Another interesting study would be analyze some of these samples with analysts manually, to observe how much time is actually saved compared to the execution of our solution.

Practically speaking, another next step toward producing a complete research on this topic would be enhance our analysis pipeline (*i.e.,* the deobfuscation and analysis tool and our emulator). Based on the insights we gathered by manually analyzing samples we can provide an engineering advancement to the solutions to provide a more solid approach.

Another promising line of research that could enhance the information drawn from these samples, involves a deeper exploration of the hash functions themselves. Beyond deobfuscating API names, analyzing and clustering malware according to their hashing algorithms could reveal relationships between families, variants, and toolchains. Since many samples reuse similar hashing functions with only minor variations (*e.g.,* different seeds or mixing constants), clustering based on hash-function similarity may help track code lineage and identify shared development frameworks.

Recent advances in deep-learning-based (DL) binary similarity analysis provide a compelling foundation for this direction. For instance, Artuso *et al.* [6] demonstrated neural approaches capable of identifying semantically equivalent code fragments even across binaries compiled with different optimizations or symbol names. Such models could be adapted to locate hashing functions automatically, starting from a corpus of known implementations (*e.g.,* from HashDB [103]). By comparing each candidate region within a malware sample to this pool, we could identify the most similar known hash function and subsequently analyze its unique traits. This would enable efficient large-scale identification and classification of API hashing algorithms, supporting both automated reverse engineering and threat intelligence enrichment.

Extending PFuzzer. PFuzzer provides an effective means to uncover environmental characteristics that lead to multiple execution paths in malware. Nevertheless, not all samples responded as expected, often due to factors outside the environmental dimensions that PFuzzer currently manipulates. One promising direction is to investigate additional program-analysis techniques to enhance the way PFuzzer mutates the environment. In particular, the Input-to-State Correspondence (I2S) technique [7], originally proposed to overcome fuzzing roadblocks, could be adapted for this purpose.

The idea is to treat the arguments of system-query functions as inputs and to *color* them using taint-analysis principles [7]. These tainted values can then be tracked at comparison sites to identify which environmental inputs influence execution decisions. The input corresponds to the argument values derived from environmental queries, whereas the state corresponds to the comparison instructions referencing these values. This lightweight form of taint tracking could approximate data-flow reasoning with far less overhead than traditional taint analysis—an important advantage in the presence of anti-analysis constructs [30]. By applying I2S, we could extract informative strings and conditions that act as decision points or divergence triggers within the malware's logic. Incorporating these insights into PFuzzer's mutation engine would make environment fuzzing more targeted, potentially revealing additional hidden behaviors.

A second research direction for PFuzzer lies not in methodological extensions but in broadening its application scope. While this work focused on malicious software, the same principles could be applied to benign programs to identify environmental dependencies that govern legitimate multi-path behavior. Beyond vulnerability discovery, fuzzing-based analyses could illuminate configuration-dependent logic in complex software systems, yielding valuable insights for testing, debugging, and software robustness studies.

Beyond methodological improvements and extensions, a deeper investigation of underexplored aspects of our approach would yield valuable insights for the

community. For example, rigorously evaluating the effectiveness of our coverage metric—and refining it to distinguish a broader range of behaviors during analysis rather than only in post-processing—could improve overall performance. Another promising direction is to analyze the causes of shifts toward new execution paths, which could inform rule design and enable the automatic discovery of new techniques.

Closing Remarks

In summary, this thesis advances the idea that effective malware analysis does not depend solely on achieving perfect transparency, but on strategically engaging with adversarial behavior to turn evasion into evidence. By transforming anti-analysis logic into a measurable signal, the presented methods illustrate how dynamic interaction and code introspection can yield deeper visibility into the adaptive mechanisms of modern threats. Looking forward, integrating these principles with large-scale automation, intelligent prioritization, and cross-domain reasoning could lead to the next generation of resilient analysis frameworks—systems capable not only of observing malware but of understanding and anticipating its strategies. Ultimately, this work contributes to shifting the analytical paradigm from passive observation to active exploration, reinforcing the role of dynamic program analysis as a cornerstone of practical and forward-looking cybersecurity research.

Bibliography

[1] *A Portable Executable Analyzer Tool*. URL: https://github.com/horsicq/Detect-It-Easy.

[2] Acronis. *Avaddon Ransomware Cleans the bin for You*. URL: https://www.acronis.com/en/blog/posts/avaddon-ransomware/.

[3] Anthropic AI. *Claude Code*. URL: https://claude.com/product/claude-code.

[4] Saleh Alzahrani, Yang Xiao, and Wei Sun. "An Analysis of Conti Ransomware Leaked Source Codes". In: *IEEE Access* PP (Jan. 2022), pp. 1–1. DOI: 10.1109/ACCESS.2022.3207757.

[5] Hyrum S. Anderson and Phil Roth. *EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models*. 2018. arXiv: 1804.04637 [cs.CR].

[6] Fiorella Artuso, Marco Mormando, Giuseppe Antonio Di Luna, and Leonardo Querzoni. "BinBert: Binary Code Understanding With a Fine-Tunable and Execution-Aware Transformer". In: *IEEE Transactions on Dependable and Secure Computing* 22.1 (2025), pp. 308–326. DOI: 10.1109/TDSC.2024.3397660.

[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *Proceedings 2019 Network and Distributed System Security Symposium* (2019). URL: https://api.semanticscholar.org/CorpusID:85546717.

[8] Cristian Assaiante, Simone Nicchi, Daniele Cono D'Elia, and Leonardo Querzoni. "Evading Userland API Hooking, Again: Novel Attacks and a Principled Defense Method". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 21st International Conference, DIMVA 2024, Lausanne, Switzerland, July 17–19, 2024, Proceedings*. 2024, pp. 150–173. ISBN: 978-3-031-64170-1. DOI: 10.1007/978-3-031-64171-8_8. URL: https://doi.org/10.1007/978-3-031-64171-8_8.

[9] Mitre ATT&CK. *Obfuscated Files or Information: Dynamic API Resolution.* URL: https://attack.mitre.org/techniques/T1027/007/.

[10] Mitre ATT&CK. *Virtualization/Sandbox Evasion.* URL: https://attack.mitre.org/techniques/T1497/.

[11] *AVClass - A Tool for Malware Labeling.* URL: https://github.com/malicialab/avclass.

[12] 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: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3487–3504. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/avllazagaj.

[13] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. "CodePlan: Repository-Level Coding using LLMs and Planning". In: *Proc. ACM Softw. Eng.* 1.FSE (2024). DOI: 10.1145/3643757. URL: https://doi.org/10.1145/3643757.

[14] Thomas Ball and Susan Horwitz. "Slicing programs with arbitrary control-flow". In: *Automated and Algorithmic Debugging*. Ed. by Peter A. Fritzson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 206–222. ISBN: 978-3-540-48141-6.

[15] 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 Symposium on Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, 2010.

[16] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. "Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge". In: *32st USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023.

[17] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. "TTAnalyze: A Tool for Analyzing Malware". In: 2006. URL: https://api.semanticscholar.org/CorpusID:11273952.

[18] Fabrice Bellard. *QEMU.* URL: https://www.qemu.org/.

[19] Stefano Berlato and Mariano Ceccato. "A large-scale study on the adoption of anti-debugging and anti-tampering protections in android apps". In: *Journal of Information Security and Applications* 52 (2020), p. 102463. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2020.102463. URL: https://www.sciencedirect.com/science/article/pii/S2214212619305976.

[20] Bitdefender. *The Differences Between Static and Dynamic Malware Analysis*. URL: https://www.bitdefender.com/en-us/blog/businessinsights/the-differences-between-static-malware-analysis-and-dynamic-malware-analysis.

[21] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344. ISBN: 9781450349468. DOI: 10.1145/3133956.3134020. URL: https://doi.org/10.1145/3133956.3134020.

[22] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing as Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, 1032–1043. ISBN: 9781450341394. DOI: 10.1145/2976749.2978428. URL: https://doi.org/10.1145/2976749.2978428.

[23] Marcus Botacin. "Fuzzing and Symbolic Execution for Multipath Malware Tracing: Bridging Theory and Practice via Survey and Experiments". In: *Digital Threats* 5.4 (Dec. 2024). DOI: 10.1145/3700147. URL: https://doi.org/10.1145/3700147.

[24] 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: *2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2025, pp. 1121–1139. DOI: 10.1109/EuroSP63326.2025.00068. URL: https://doi.ieeecomputersociety.org/10.1109/EuroSP63326.2025.00068.

[25] 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: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Manuel Egele, Veelasha Moonsamy,

Daniel Gruss, and Michele Carminati. Cham: Springer Nature Switzerland, 2025, pp. 287–293. ISBN: 978-3-031-97620-9.

[26] Doug Brubacher. "Detours: Binary Interception of Win32 Functions". In: *Windows NT 3rd Symposium (Windows NT 3rd Symposium)*. Seattle, WA: USENIX Association, July 1999. URL: https://www.usenix.org/conference/windows-nt-3rd-symposium/detours-binary-interception-win32-functions.

[27] Derek Bruening, Qin Zhao, and Saman Amarasinghe. "Transparent dynamic instrumentation". In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE '12. London, England, UK: Association for Computing Machinery, 2012, 133–144. ISBN: 9781450311762. DOI: 10.1145/2151024.2151043. URL: https://doi.org/10.1145/2151024.2151043.

[28] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. "Automatically Identifying Trigger-based Behavior in Malware". In: *Botnet Detection: Countering the Largest Security Threat*. Ed. by Wenke Lee, Cliff Wang, and David Dagon. Boston, MA: Springer US, 2008, pp. 65–88. ISBN: 978-0-387-68768-1. DOI: 10.1007/978-0-387-68768-1_4. URL: https://doi.org/10.1007/978-0-387-68768-1_4.

[29] Gianluca Capozzi, Tong Tang, Jie Wan, Ziqi Yang, Daniele Cono D'Elia, Giuseppe Antonio Di Luna, Lorenzo Cavallaro, and Leonardo Querzoni. "On the Lack of Robustness of Binary Function Similarity Systems". In: *10th IEEE European Symposium on Security and Privacy, EuroS&P 2025, Venice, Italy, June 30 - July 4, 2025*. IEEE, 2025, pp. 980–1001. DOI: 10.1109/EUROSP63326.2025.00060. URL: https://doi.org/10.1109/EuroSP63326.2025.00060.

[30] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. "On the Limits of Information Flow Techniques for Malware Analysis and Containment". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Diego Zamboni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 143–163. ISBN: 978-3-540-70542-0.

[31] Luca Caviglione, Michał Choraś, Igino Corona, Artur Janicki, Wojciech Mazurczyk, Marek Pawlicki, and Katarzyna Wasielewska. "Tight Arms Race: Overview of Current Malware Threats and Trends in Their Detection". In: *IEEE Access* 9 (2021), pp. 5371–5396. DOI: 10.1109/ACCESS.2020.3048319.

[32] ChannelE2E. *Cyber Threat Trends in 2024: The Landscape According to Top Industry Reports*. URL: https://www.channele2e.com/native/cyber-threat-trends-in-2024-the-landscape-according-to-top-industry-reports.

[33] Efstratios Chatzoglou, Georgios Karopoulos, Georgios Kambourakis, and Zisis Tsiatsikas. *Bypassing antivirus detection: old-school malware, new tricks*. 2023. arXiv: 2305.04149 [cs.CR]. URL: https://arxiv.org/abs/2305.04149.

[34] CheckPoint. *Anti-Debug Tricks*. URL: https://anti-debug.checkpoint.com/.

[35] Checkpoint. *Evasion techniques*. URL: https://evasions.checkpoint.com/.

[36] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. "Obfuscation-Resilient Executable Payload Extraction From Packed Malware". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3451–3468. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/cheng-binlin.

[37] *Cuckoo Sandbox*. URL: https://github.com/cuckoosandbox/cuckoo.

[38] Cyphere. *Malware Statistics to Be Taken Seriously*. URL: https://thecyphere.com/blog/malware-statistics/.

[39] Control D. *100 Chilling Malware Statistics & Trends (2023-2025)*. URL: https://controld.com/blog/malware-statistics-trends/.

[40] 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 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS '19. Auckland, New Zealand: Association for Computing Machinery, 2019, 15–27. ISBN: 9781450367523. DOI: 10.1145/3321705.3329819. URL: https://doi.org/10.1145/3321705.3329819.

[41] Daniele Cono D'Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. "On the Dissection of Evasive Malware". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 2750–2765. DOI: 10.1109/TIFS.2020.2976559.

[42] Daniele Cono D'Elia, Lorenzo Invidia, Federico Palmaro, and Leonardo Querzoni. "Evaluating Dynamic Binary Instrumentation Systems for Conspicuous Features and Artifacts". In: *Digital Threats* 3.2 (Feb. 2022). DOI: 10.1145/3478520. URL: https://doi.org/10.1145/3478520.

[43] Daniele Cono D'Elia, Simone Nicchi, Matteo Mariani, Matteo Marini, and Federico Palmaro. "Designing Robust API Monitoring Solutions". In: *IEEE Transactions on Dependable and Secure Computing* 20.1 (2023), pp. 392–406. DOI: 10.1109/TDSC.2021.3133729.

[44] 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*. ASIA CCS '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1600–1616. ISBN: 9798400714108. DOI: 10.1145/3708821.3736206. URL: https://doi.org/10.1145/3708821.3736206.

[45] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. "Ether: malware analysis via hardware virtualization extensions". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: Association for Computing Machinery, 2008, pp. 51–62. ISBN: 9781595938107. DOI: 10.1145/1455770.1455779. URL: https://doi.org/10.1145/1455770.1455779.

[46] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. "Dynamic spyware analysis". In: *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*. ATC'07. Santa Clara, CA: USENIX Association, 2007. ISBN: 9998888776.

[47] Mojtaba Eskandari, Zeinab Khorshidpour, and Sattar Hashemi. "HDM-Analyser: A hybrid analysis approach based on data mining techniques for malware detection". In: *Journal of Computer Virology and Hacking Techniques* 9 (May 2013). DOI: 10.1007/s11416-013-0181-8.

[48] Francisco Falcon and Nahuel Riva. "Dynamic Binary Instrumentation Frameworks: I know you're there spying on me." In: *Recon*. 2012.

[49] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. "AFL++: combining incremental steps of fuzzing research". In: *Proceedings of the 14th USENIX Conference on Offensive Technologies*. WOOT'20. USA: USENIX Association, 2020.

[50] Brian Gallagher. *The Malware Arms Race: Why Traditional Defenses Are Failing*. URL: https://www.linkedin.com/pulse/malware-arms-race-why-traditional-defenses-failing-brian-gallagher-695le/.

[51] Nicola Galloro, Mario Polino, Michele Carminati, Andrea Continella, and Stefano Zanero. "A Systematical and longitudinal study of evasive behaviors in windows malware". In: *Comput. Secur.* 113.C (Feb. 2022). ISSN: 0167-4048. DOI: 10.1016/j.cose.2021.102550. URL: https://doi.org/10.1016/j.cose.2021.102550.

[52] Jiaxuan Geng, Junfeng Wang, Zhiyang Fang, Yingjie Zhou, Di Wu, and Wenhan Ge. "A survey of strategy-driven evasion methods for PE malware: Transformation, concealment, and attack". In: *Computers & Security* 137 (2024), p. 103595. ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2023.103595. URL: https://www.sciencedirect.com/science/article/pii/S0167404823005059.

[53] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." In: *NDSS*. The Internet Society, 2008.

[54] Google. *AFL*. URL: https://github.com/google/AFL.

[55] Google. *Honggfuzz*. URL: https://github.com/google/honggfuzz.

[56] Google. *WinAFL*. URL: https://github.com/googleprojectzero/winafl.

[57] Cosmin Gorgovan. *Escaping DynamoRIO and Pin*. 2014. URL: https://github.com/lgeek/dynamorio_pin_escape.

[58] 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. Rome, Italy: Association for Computing Machinery, 2023, pp. 8–14. ISBN: 9798400700859. DOI: 10.1145/3578357.3589455. URL: https://doi.org/10.1145/3578357.3589455.

[59] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. "AntiFuzz: Impeding Fuzzing Audits of Binary Executables". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1931–1947. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/guler.

[60] Nikhil Gupta. *API Hashing - Why Malware Loves (And You Should Care)*. URL: https://securitymaven.medium.com/api-hashing-why-malware-loves-and-you-should-care-77c5135d9aaa.

[61] Mahmoud Hammad, Joshua Garcia, and Sam Malek. "A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing

Machinery, 2018, 421–431. ISBN: 9781450356381. DOI: 10.1145/3180155.3180228. URL: https://doi.org/10.1145/3180155.3180228.

[62] *Hybrid Analysis*. URL: https://hybrid-analysis.com/.

[63] Infosec. *Malware Anti-Analysis Techniques and Ways to Bypass Them*. URL: https://www.infosecinstitute.com/resources/malware-analysis/malware-anti-analysis-techniques-ways-bypass/.

[64] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. "Fuzzification: Anti-Fuzzing Techniques". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1913–1930. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/jung.

[65] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. "Emulating emulation-resistant malware". In: *Proceedings of the 1st ACM Workshop on Virtual Machine Security*. VMSec '09. Chicago, Illinois, USA: Association for Computing Machinery, 2009, 11–22. ISBN: 9781605587806. DOI: 10.1145/1655148.1655151. URL: https://doi.org/10.1145/1655148.1655151.

[66] Mohammad Sina Karvandi, MohammadHosein Gholamrezaei, Saleh Khalaj Monfared, Soroush Meghdadizanjani, Behrooz Abbassi, Ali Amini, Reza Mortazavi, Saeid Gorgin, Dara Rahmati, and Michael Schwarz. "HyperDbg: Reinventing Hardware-Assisted Debugging". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1709–1723. ISBN: 9781450394505. DOI: 10.1145/3548606.3560649. URL: https://doi.org/10.1145/3548606.3560649.

[67] Yuhei Kawakoya, Makoto Iwamura, and Jun Miyoshi. "Taint-assisted IAT Reconstruction against Position Obfuscation". In: *Journal of Information Processing* 26 (2018), pp. 813–824. DOI: 10.2197/ipsjjip.26.813.

[68] Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu. "API Chaser: Anti-analysis Resistant Malware Analyzer". In: *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 8145*. RAID 2013. Rodney Bay, St. Lucia: Springer-Verlag, 2013, 123–143. ISBN: 9783642412837. DOI: 10.1007/978-3-642-41284-4_7. URL: https://doi.org/10.1007/978-3-642-41284-4_7.

[69] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. "libdft: practical dynamic data flow tracking for commodity systems". In: *SIGPLAN Not.* 47.7 (Mar. 2012), 121–132. ISSN: 0362-1340. DOI: 10.1145/2365864.2151042. URL: https://doi.org/10.1145/2365864.2151042.

[70] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. "Barecloud: bare-metal analysis-based evasive malware detection". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, pp. 287–301. ISBN: 9781931971157.

[71] Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel. "PwIN - Pwning Intel piN: Why DBI is Unsuitable for Security Applications". In: *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I*. Barcelona, Spain: Springer-Verlag, 2018, pp. 363–382. ISBN: 978-3-319-99072-9. DOI: 10.1007/978-3-319-99073-6_18. URL: https://doi.org/10.1007/978-3-319-99073-6_18.

[72] KnowBe4. *AV-TEST: There are now 12 million new malware variants per month*. URL: https://blog.knowbe4.com/av-test-there-are-now-12-million-new-malware-variants-per-month.

[73] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. "The power of procrastination: detection and mitigation of execution-stalling malicious code". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 285–296. ISBN: 9781450309486. DOI: 10.1145/2046707.2046740. URL: https://doi.org/10.1145/2046707.2046740.

[74] Marek Krčál, Ondřej Švec, Martin Bálek, and Otakar Jašek. *Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only*. 2018. URL: https://openreview.net/forum?id=HkHrmM1PM.

[75] Alexander Kuechler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. "Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes". In: *Proceedings 2021 Network and Distributed System Security Symposium* (2021). URL: https://api.semanticscholar.org/CorpusID:231798972.

[76] James R Larus and Satish Chandra. *Using tracing and dynamic slicing to tune compilers*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1993.

[77] Gwangyeol Lee, Minho Kim, Jeong Hyun Yi, and Haehyun Cho. "Pinicorn: Towards Automated Dynamic Analysis for Unpacking 32-Bit PE Malware". In: *Electronics* 13.11 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13112081. URL: https://www.mdpi.com/2079-9292/13/11/2081.

[78] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1 (Dec. 2018). DOI: 10.1186/s42400-018-0002-y.

[79] Qiang Li, Yunan Zhang, Liya Su, Yang Wu, Xinjian Ma, and Zeming Yang. "An Improved Method to Unveil Malware's Hidden Behavior". In: *Information Security and Cryptology*. Ed. by Xiaofeng Chen, Dongdai Lin, and Moti Yung. Cham: Springer International Publishing, 2018, pp. 362–382. ISBN: 978-3-319-75160-3.

[80] Xiaoning Li and Kang Li. "Defeating the Transparency Features of Dynamic Binary Instrumentation." In: *BlackHat USA*. 2014.

[81] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. "Detecting Environment-Sensitive Malware". In: *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Menlo Park, CA, USA, 2011. DOI: 10.1007/978-3-642-23644-0_18.

[82] *List of APIs used in malicious actions*. URL: https://malapi.io/.

[83] *List of Hash & Name values for BlackMatter*. URL: https://gist.github.com/jgru/c58851bde4ee4778d83c84babb2f69d1.

[84] LLVM. *libFuzzer*. URL: https://llvm.org/docs/LibFuzzer.html.

[85] LMNTRIX. *The Problem With "Traditional" Antivirus Software*. URL: https://lmntrix.com/blog/the-problem-with-traditional-antivirus-software/.

[86] 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 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, 190–200. ISBN: 1595930566. DOI: 10.1145/1065010.1065034. URL: https://doi.org/10.1145/1065010.1065034.

[87] Gustav Lundsgård and Victor Nedström. "Bypassing modern sandbox technologies An experiment on sandbox evasion techniques". In: 2016. URL: https://api.semanticscholar.org/CorpusID:189805343.

[88] Lorenzo Maffia, Dario Nisi, Platon Kotzias, Giovanni Lagorio, Simone Aonzo, and Davide Balzarotti. "Longitudinal Study of the Prevalence of Malware Evasive Techniques". In: *CoRR* abs/2112.11289 (2021). arXiv: 2112.11289. URL: https://arxiv.org/abs/2112.11289.

[89] Malwation. *Static Malware Analysis vs Dynamic Malware Analysis - Comparison Chart*. URL: https://www.malwation.com/blog/static-malware-analysis-vs-dynamic-malware-analysis-comparison-chart.

[90] Mandiant. *Tracking Malware with Import Hashing*. URL: https://cloud.google.com/blog/topics/threat-intelligence/tracking-malware-import-hashing/.

[91] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "Fuzzing: Art, Science, and Engineering". In: *CoRR* abs/1812.00140 (2018). arXiv: 1812.00140. URL: http://arxiv.org/abs/1812.00140.

[92] Microsoft. *Microsoft Detours*. URL: https://github.com/microsoft/Detours.

[93] Microsoft. *Microsoft Technical documentation*. URL: https://learn.microsoft.com/en-us/docs/.

[94] Microsoft. *Support for Windows 7 product has ended*. URL: https://learn.microsoft.com/en-us/lifecycle/products/windows-7.

[95] *Microsoft Export Table*. URL: https://www.aldeid.com/wiki/PE-Portable-executable#Export_Table.

[96] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. "Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts". In: *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 1009–1024. URL: https://api.semanticscholar.org/CorpusID:12665057.

[97] MoneyWeek. *Cybersecurity stocks: why now might be the time to buy*. URL: https://moneyweek.com/investments/tech-stocks/buy-cybersecurity-stocks.

[98] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Exploring Multiple Execution Paths for Malware Analysis". In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. 2007, pp. 231–245. DOI: 10.1109/SP.2007.17.

[99] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. 3rd ed. Hoboken and N.J: John Wiley & Sons, 2012.

[100] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavy-weight dynamic binary instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: https://doi.org/10.1145/1273442.1250746.

[101] James Newsome and Dawn Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software." In: Feb. 2005.

[102] Nullteilerfrei. *API-Hashing in the Sodinokibi/REvil Ransomware - Why and How?* URL: https://blag.nullteilerfrei.de/2019/11/09/api-hashing-why-and-how/.

[103] OALabs. *Hash DB*. URL: https://github.com/OALabs/hashdb.

[104] OALabs. *HashDB IDA Pro*. URL: https://github.com/OALabs/hashdb-ida/.

[105] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. "Obfuscation: where are we in anti-DSE protections? (a first attempt)". In: *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*. SSPREW9 '19. San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019. ISBN: 9781450377461. DOI: 10.1145/3371307.3371309. URL: https://doi.org/10.1145/3371307.3371309.

[106] OpenAI. *ChatGPT*. URL: https://chatgpt.com/.

[107] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. "X-Force: Force-Executing Binary Programs for Security Applications". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 829–844. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/peng.

[108] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. "Measuring and Defeating Anti-Instrumentation-Equipped Malware". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Michalis Polychronakis and Michael Meier. Cham: Springer International Publishing, 2017, pp. 73–96. ISBN: 978-3-319-60876-1.

[109] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. "Measuring and Defeating Anti-Instrumentation-Equipped Malware". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Michalis Polychronakis and

Michael Meier. Cham: Springer International Publishing, 2017, pp. 73–96. ISBN: 978-3-319-60876-1.

[110] Giacomo Priamo, Daniele Cono D'Elia, Mathias Payer, and Leonardo Querzoni. "Towards Path-aware Coverage-guided Fuzzing". In: *Proceedings of the 2026 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '26. Sydney, Australia: IEEE Press, 2026.

[111] *Qiling*. URL: https://github.com/qilingframework/qiling.

[112] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. *Malware Detection by Eating a Whole EXE*. 2017. arXiv: 1710.09435 [stat.ML]. URL: https://arxiv.org/abs/1710.09435.

[113] Hex Rays. *Calculating API hashes with IDA Pro*. URL: https://hex-rays.com/blog/calculating-api-hashes-with-ida-pro.

[114] Red Team Notes. *Windows API Hashing in Malware*. URL: https://www.ired.team/offensive-security/defense-evasion/windows-api-hashing-in-malware.

[115] Christian Rossow, Christian Dietrich, and Herbert Bos. "Large-Scale analysis of malware downloaders". In: *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA'12. Heraklion, Crete, Greece: Springer-Verlag, 2012, 42–61. ISBN: 9783642372995. DOI: 10.1007/978-3-642-37300-8_3. URL: https://doi.org/10.1007/978-3-642-37300-8_3.

[116] Italo Santos, Cleyton Magalhaes, and Ronnie de Souza Santos. *Model-Assisted and Human-Guided: Perceptions and Practices of Software Professionals Using LLMs for Coding*. 2025. arXiv: 2510.09058 [cs.SE]. URL: https://arxiv.org/abs/2510.09058.

[117] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)". In: *Proceedings of the 2010 IEEE Symposium on Security and Privacy*. SP '10. USA: IEEE Computer Society, 2010, pp. 317–331. ISBN: 9780769540351. DOI: 10.1109/SP.2010.26. URL: https://doi.org/10.1109/SP.2010.26.

[118] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. "AVclass: A Tool for Massive Malware Labeling". In: *International Symposium on Recent Advances in Intrusion Detection*. 2016. URL: https://api.semanticscholar.org/CorpusID:7715150.

[119] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. "AddressSanitizer: a fast address sanity checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 28.

[120] Andrii Shalaginov, Sergii Banin, Ali Dehghantanha, and Katrin Franke. "Machine Learning Aided Static Malware Analysis: A Survey and Tutorial". In: *CoRR* abs/1808.01201 (2018). arXiv: 1808.01201. URL: http://arxiv.org/abs/1808.01201.

[121] Rami Sihwail, Khairuddin Omar, and Khairul Akram Zainol Ariffin. "A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis". In: *International Journal on Advanced Science Engineering and Information Technology* 8 (Sept. 2018), p. 1662. DOI: 10.18517/ijaseit.8.4-2.6827.

[122] Rami Sihwail, Khairuddin Bin Omar, and Khairul Akram Zainol Ariffin. "A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis". In: *International Journal on Advanced Science, Engineering and Information Technology* (2018). URL: https://api.semanticscholar.org/CorpusID:54017419.

[123] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st. USA: No Starch Press, 2012. ISBN: 1593272901.

[124] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. "Dynamic program analysis of Microsoft Windows applications". In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 2010, pp. 2–12. DOI: 10.1109/ISPASS.2010.5452079.

[125] Spacelift. *50+ Malware Statistics for 2025*. URL: https://spacelift.io/blog/malware-statistics.

[126] Ke Sun, Xiaoning Li, and Ya Ou. "Break Out of the Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation." In: *BlackHat Asia*. 2016.

[127] Positive Technologies. *Sandbox Detection and Evasion Techniques. How Malware Hides from Sandboxes*. URL: https://global.ptsecurity.com/en/research/analytics/antisandbox-techniques/.

[128] Jean-Pierre Lesueur Thomas Roccia. *Unprotect Project*. URL: https://unprotect.it/.

[129] Christoph Treude and Margaret-Anne Storey. *Generative AI and Empirical Software Engineering: A Paradigm Shift*. 2025. arXiv: 2502.08108 [cs.SE]. URL: https://arxiv.org/abs/2502.08108.

[130] *Understanding BlackMatter's API Hashing*. URL: https://blog.digital-investigations.info/2021-08-05-understanding-blackmatters-api-hashing.html.

[131] *Understanding BlackMatter's API Hashing*. URL: https://blog.digital-investigations.info/2021-08-05-understanding-blackmatters-api-hashing.html.

[132] *Unicorn*. URL: https://github.com/unicorn-engine/unicorn.

[133] Unit42. *Defeating BazarLoader Anti-Analysis Techniques*. URL: https://unit42.paloaltonetworks.com/bazarloader-anti-analysis-techniques/.

[134] UnpackMe. *Malware Trends: Yearly 2023*. URL: https://blog.unpac.me/2024/01/30/malware-trends-yearly/.

[135] Reini Urban. *Smhasher/murmurhash2*. URL: https://github.com/rurban/smhasher/tree/4db9ed2dc78a162788d0b86a91cff3fa5a700f7c/MurmurHash2.cpp.

[136] A. Vasudevan and R. Yerraballi. "Cobra: fine-grained malware analysis using stealth localized-executions". In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. 2006, 15 pp.–279. DOI: 10.1109/SP.2006.9.

[137] VirusTotal Victor Alvarez. *Anti-Debugging and Anti-Virtualization YARA Rules*. URL: https://github.com/Yara-Rules/rules/blob/master/antidebug_antivm/antidebug_antivm.yar.

[138] Vasilis Vouvoutsis, Fran Casino, and Constantinos Patsakis. "On the effectiveness of binary emulation in malware classification". In: *Journal of Information Security and Applications* 68 (2022), p. 103258. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2022.103258. URL: https://www.sciencedirect.com/science/article/pii/S2214212622001223.

[139] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. "Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 1–15. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/wang.

[140] Thomas Weber, Maximilian Brandmaier, Albrecht Schmidt, and Sven Mayer. "Significant Productivity Gains through Programming with Large Language Models". In: *Proc. ACM Hum.-Comput. Interact.* 8.EICS (2024). DOI: 10.1145/3661145. URL: https://doi.org/10.1145/3661145.

[141] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. "GoldenEye: Efficiently and Effectively Unveiling Malware's Targeted Environment". In: *Research in Attacks, Intrusions and Defenses*. Ed. by Angelos Stavrou, Herbert Bos, and Georgios Portokalidis. Cham: Springer International Publishing, 2014, pp. 22–45. ISBN: 978-3-319-11379-1.

[142] 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: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1121–1138. DOI: 10.1109/SP40000.2020.00035.