



THE ARCHITECTURAL EVOLUTION, ADVERSARIAL MECHANICS, AND DEFENSIVE ENGINEERING OF ENDPOINT DETECTION AND RESPONSE

www.front-code.com

SEP 2026

The Architectural Evolution, Adversarial Mechanics, and Defensive Engineering of Next-Generation Endpoint Detection and Response Systems

[Front-code] • [contact@front-code.com]

ABSTRACT

Endpoint Detection and Response (EDR) has progressed from signature-bound antivirus and user-mode API interception to kernel-resident telemetry, cloud-scale correlation, and hardware-assisted control-flow validation. This article reviews that evolution and analyzes the adversarial techniques that shaped it, including direct and indirect system calls, dynamic-library remapping, call-stack spoofing, sleep obfuscation, reflective loading, and process impersonation. It then evaluates defensive countermeasures spanning ETW Threat Intelligence, eBPF and BPF-LSM, whole-system provenance graphs, graph neural networks, security data lakehouses, Intel Processor Trace, Control-Flow Enforcement Technology, and agentic security-operations workflows. The resulting synthesis shows a continuing migration of trust from mutable user space toward kernel, hardware, and causally correlated telemetry, while emphasizing the need for deterministic validation and human oversight in autonomous response.

Keywords: EDR; XDR; endpoint telemetry; ETW-Ti; eBPF; BPF-LSM; provenance graphs; system-call evasion; agentic SOC; Intel CET

1. Technological Foundations and the Evolutionary Trajectory of Host Telemetry

Historical Transition from Reactive Antivirus to Behavioral Telemetry

The paradigm of endpoint security has evolved over three decades from static, artifact-bound verification to dynamic, distributed behavioral analysis and causal provenance tracking¹. The earliest protective mechanisms, commonly classified as first-generation Antivirus (AV) platforms, relied on deterministic signature matching and heuristic hashing algorithms¹. These systems scanned files residing on secondary storage, computing cryptographic hashes (such as MD5 or SHA-256) of portable executable (PE) files and comparing them against flat repositories of known malicious indicators of compromise (IOCs)¹. While computationally efficient and characterized by near-zero false positive rates for known binaries, signature-based AV proved fundamentally incapable of countering polymorphic malware, runtime binary packers, memory-only execution primitives, and zero-day vulnerabilities that altered syntactic binary representations while preserving malicious semantics¹.

To address these visibility gaps, Host Intrusion Prevention Systems (HIPS) emerged, introducing rule-based inspection of local operating system activities². HIPS solutions monitored local process activities, registry modifications, and file access attempts using static heuristic rules². However, the absence of persistent contextual state across distinct execution phases, coupled with rigid matching rules, resulted in severe operational challenges: organizations experienced unsustainable alert fatigue, high system brittleness, and frequent system lockups caused by policy overreach².

The modern category of Endpoint Detection and Response (EDR) marked a major operational and conceptual shift: rather than attempting to prevent every anomalous interaction pre-execution, defensive architectures prioritized continuous, fine-grained telemetry collection, cross-event correlation, behavioral profiling, and automated post-compromise containment¹. Early commercial EDR implementations relied primarily on user-mode API hooking and operating system event notification routines¹. In Microsoft Windows environments, EDR agents

injected a proprietary dynamic-link library (DLL) into the virtual address space of newly spawned user-mode processes⁷. This injected module installed inline function hooks (typically 5-byte relative JMP instructions or indirect trampolines) across native APIs exported by ntdll.dll and standard subsystems like kernel32.dll⁷. When an application invoked security-sensitive Win32 or Native APIs—such as NtAllocateVirtualMemory, NtWriteVirtualMemory, or NtCreateThreadEx—execution was diverted to the EDR sensor engine, which inspected the parameters against behavioral heuristics before returning execution to the native system call stub⁷.

Kernel-Space Callback Subsystems and Filesystem Mini-Filters

Concurrently, EDR architectures integrated kernel-mode drivers to capture lower-level operating system events that could not be intercepted reliably from user space⁷. By utilizing documented Windows kernel callback routines, defensive drivers established telemetry collection pipelines across core operating system primitives⁷:

PsSetCreateProcessNotifyRoutineEx: Audited the process lifecycle, supplying parent-child process relationships, command-line arguments, and file image identifiers prior to initial thread execution⁷.

PsSetCreateThreadNotifyRoutine: Monitored thread creation across local and remote processes, enabling the detection of cross-process injection vectors⁷.

PsSetLoadImageNotifyRoutine: Monitored executable modules and drivers mapped into memory spaces, supporting DLL sideloading and unbacked module detection⁷.

ObRegisterCallbacks: Mediated process and thread handle creation and duplication, allowing the stripping of dangerous access rights (such as PROCESS_VM_WRITE or PROCESS_CREATE_THREAD) requested against protected processes⁷.

These telemetry feeds were supplemented by File System Minifilter drivers registering pre-operation and post-operation callbacks through the Filter Manager to inspect I/O request packets (IRPs), and Windows Filtering Platform (WFP) callout drivers monitoring socket bindings, connection requests, and network transport streams⁷.

Architectural Bottlenecks of Early-Generation EDR

Despite these advancements, first- and second-generation EDR architectures encountered acute operational bottlenecks¹. The volume of raw telemetry saturated network bandwidth and overwhelmed centralized Security Operations Centers (SOCs)⁵. Security analysts were inundated with tens of thousands of disconnected alerts daily, lacking causal connection and business context⁵.

Crucially, the asymmetry between user-mode monitoring agents and unprivileged adversaries became an attack vector¹³. Because

the EDR's inspection logic often resided within the same unprivileged ring 3 (user-space) memory space as untrusted application code, adversaries developed evasion primitives to bypass, corrupt, or blind these user-mode interception pipelines⁷. This structural limitation prompted the development of modern EDR architectures, defined by kernel-space provenance tracking, hardware-assisted telemetry, decoupled cloud-native analytical pipelines, and automated machine-learning-driven investigation engines¹.

2. Next-Generation Enterprise Architectural Paradigms

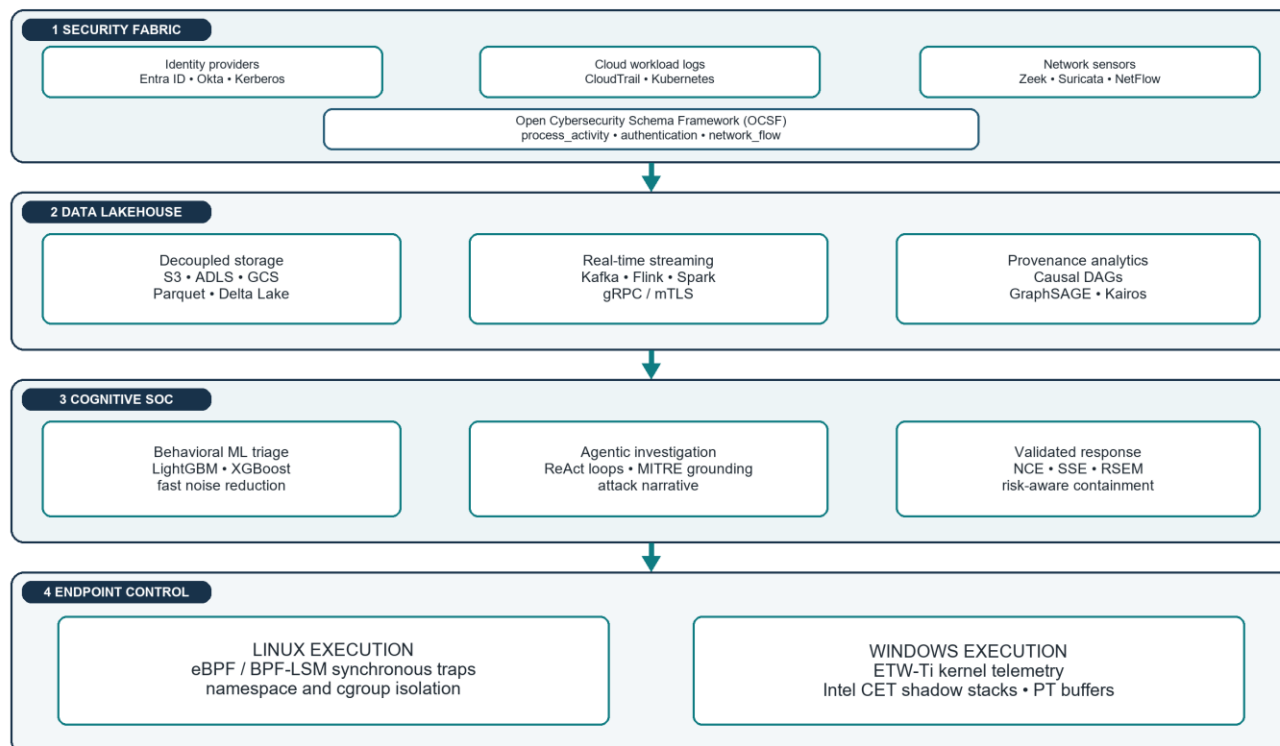


Figure 1. Reference architecture for next-generation EDR/XDR telemetry, analytics, investigation, and endpoint control.

Convergence with Extended Detection and Response and Enterprise Security Data Lakes

The host-centric boundary of traditional EDR has expanded through convergence with Extended Detection and Response (XDR) architectures and enterprise Security Data Lakes¹⁸. While legacy EDR platforms analyzed endpoint events in isolation, advanced threat actors increasingly exploit distributed enterprise vectors—combining compromised identity credentials, cloud infrastructure control-plane modifications, API-driven software-as-a-service (SaaS) exfiltration, and lateral network movements that generate minimal footprints on any individual endpoint¹².

XDR architectures break these telemetry silos by ingesting, normalizing, and correlating heterogeneous data streams across endpoints, identity management platforms (such as Okta and Microsoft Entra ID), network detection and response (NDR) sensors, and cloud workload infrastructures (such as AWS CloudTrail and Kubernetes audit logs)¹⁸.

A critical enabler of this unification is the rapid industry-wide adoption of the Open Cybersecurity Schema Framework (OCSF)¹⁹. Historically, security organizations bore a heavy data normalization tax, devoting engineering resources to

maintaining ad-hoc, brittle parsers to align incompatible log structures across diverse vendors¹⁹. OCSF establishes an open, vendor-agnostic, extensible schema governed by defined semantic taxonomies¹⁹. By structuring diverse events into standardized categories—such as process_activity, authentication, network_flow, and security_finding—downstream analytical engines can execute portable threat-detection rules and unified graph queries across disparate operational systems²¹.

Concurrently, data storage architectures have shifted away from traditional, proprietary Security Information and Event Management (SIEM) indexing backends toward open, columnar Security Data Lakes built on formats like Apache Parquet and Delta Lake¹⁸. By physically decoupling distributed storage from compute capabilities, enterprise data lakehouses allow organizations to retain petabytes of raw, full-fidelity endpoint telemetry over long retention horizons at lower operational costs¹⁸. Cloud-native query engines (such as Amazon Athena, Apache Spark, Snowflake, and Databricks) query this structured, OCSF-compliant telemetry in place¹⁸. This eliminates the need to truncate or discard endpoint events, preserving the temporal context required to uncover long-dwell Advanced Persistent Threat (APT) campaigns¹⁷.

Cognitive Triage, Graph Neural Networks, and Autonomous Security Operations Centers

Modern enterprise security centers face significant telemetry scale: an average enterprise SOC processes between 10,000 and 11,000 alerts every twenty-four hours, with false-positive rates frequently exceeding 70% to 90%¹¹.

Empirical research reveals that approximately 28% to 40% of generated alerts are never reviewed by human operators due to acute alert fatigue and resource constraints, creating blind spots where adversaries operate undetected¹¹.

To overcome this operational challenge, next-generation architectures deploy multi-tiered cognitive triage pipelines that combine specialized machine learning models with Large Language Model (LLM) reasoning agents¹¹.

At the initial telemetry ingest layer, static and behavioral machine learning classifiers perform low-latency filtering⁷. Gradient Boosted Decision Trees (GBDTs), such as XGBoost and LightGBM models trained on structural PE headers, dynamic import features, and static entropy distributions, execute pre-execution artifact evaluations in sub-millisecond windows⁷. Downstream runtime event streams are continuously evaluated by sequential models (e.g., Long Short-Term Memory networks and Transformer-based event encoders) trained to identify deviations from established baseline execution behaviors⁷.

Beyond individual event classification, modern detection architectures employ Graph Neural Networks (GNNs) operating on Whole-System Provenance Graphs¹⁷. Conventional alert correlation engines treat operating system events as isolated points in time, obscuring complex multi-stage attack chains¹⁷.

In contrast, Provenance-based Intrusion Detection Systems (PIDS) continuously construct directed acyclic graphs (DAGs) representing causal dependencies between system entities (processes, files, threads, network sockets) interconnected by system-call-derived edges (e.g., fork, execve, read, write, connect)¹⁷.

Inductive Graph Neural Networks, such as tailored GraphSAGE architectures and temporal-spatial encoder-decoders (e.g., Kairos, ThreaTrace), map the structural evolution of these graphs over time¹⁷. By evaluating node-level and edge-level structural embeddings, GNN architectures flag causal anomalies while computing graph substructure similarity metrics¹⁷.

To mitigate the historic challenge of provenance graph dependency explosion, modern systems apply graph reduction algorithms based on Jaccard distance clustering:

$$D(s, t) = 1 - \frac{|N(s) \cap N(t)|}{|N(s) \cup N(t)|}$$

Equation (1). Jaccard-distance formulation used for provenance-graph reduction.

This mathematical formulation collapses redundant cyclic system operations while preserving causal forensic paths³⁰.

At the apex of this architectural framework sit agentic autonomous SOC frameworks (such as AgentSOC and hybrid ReAct reasoning systems)¹¹. Unlike legacy Security Orchestration, Automation, and Response (SOAR) playbooks—which operate as rigid, deterministic scripts that fail silently when encountering unexpected attacker tradecraft—agentic AI platforms leverage LLMs as reasoning engines embedded with strict validation guardrails¹².

These systems operate via dynamic planner-executor loops: upon receiving a telemetry anomaly or provenance subgraph alert, an agent formulates investigation hypotheses grounded in the MITRE ATT&CK knowledge base¹¹.

The agent queries endpoints for cross-corroborating evidence (such as parent process lineage, memory state artifacts, or network socket destinations), updates its hypothesis state machine, and reconstructs the attack narrative²⁵.

To prevent model hallucinations from triggering catastrophic remediation actions (such as isolating critical enterprise domain controllers or database clusters), advanced cognitive SOC frameworks employ dual-layer validation systems¹²:

A Narrative Counterfactual Engine (NCE) synthesizes potential threat explanations¹².

A deterministic Structural Simulation Engine (SSE) cross-checks these hypotheses against verified network topologies, active identity graphs, and system state invariants¹².

Finally, a Risk Scoring and Evaluation Module (RSEM) calculates an objective risk vector balancing remediation efficacy against business service interruption¹².

This layered approach automates the closure of 70% to 85% of benign, non-critical false-positive alerts, reducing mean-time-to-respond (MTTR) metrics from several hours down to sub-two-minute intervals, while routing structured, auditable evidence trails to senior human analysts for high-tier containment decisions¹¹.

Transition to Cloud-Native Streaming Architectures

The underlying engineering infrastructure of endpoint detection has transformed, moving from localized, monolithic client-server architectures to globally distributed, cloud-native streaming architectures². Early EDR deployments relied on periodical polling mechanisms, batching log files locally on the endpoint and transmitting them across relational database management systems or on-premises indexers². This model incurred structural delays—often introducing telemetry latencies ranging from several minutes to hours—and created single points of failure when centralized infrastructure faced traffic spikes⁵.

Modern EDR platforms leverage distributed, event-driven streaming pipelines built on technologies like Apache Kafka, Apache Pulsar, and gRPC over HTTP/2 with mutual TLS (mTLS) cryptographic handshakes⁶. Lightweight endpoint agents run asynchronous ring buffers within kernel- and low-level user-space modules⁷. These buffers stream telemetry directly to globally distributed edge ingestion points, ensuring that critical process execution, thread creation, and module loading events are ingested, parsed, and evaluated in the analytical backend within seconds⁶.

Furthermore, next-generation architectures enforce a functional split between edge-based computation and cloud-native centralized analytics². Resource-constrained endpoint environments cannot execute heavy behavioral graph correlations or large-scale machine learning models without degrading user experience or introducing significant CPU and memory overhead². Consequently, modern edge agents limit local processing to real-time deterministic signature evaluation, critical kernel callback telemetry collection, lightweight anomaly heuristics, and high-speed policy enforcement (e.g., immediate process suspension or network packet drops via WFP or eBPF)⁷.

Complex stateful operations—such as multi-host provenance graph construction, long-term behavioural drift tracking, and cross-source identity correlation—are offloaded to auto-scaling cloud microservices running on elastic container orchestration frameworks².

3. Adversarial Mechanics and EDR Circumvention Methodologies

User-Mode Hook Evasion: Direct and Indirect System Calls

The reliance of commercial EDR agents on user-mode inline API hooking fostered a dedicated offensive research domain focused on evading user-space telemetry taps⁷.

When an EDR hooks a function such as `NtProtectVirtualMemory` in `ntdll.dll`, it typically overwrites the initial byte sequence of the function stub—the classic 5-byte prologue: `mov r10, rcx mov eax, eax`, with an unconditional jump (`JMP rel32 / E9` opcode, or an indirect absolute jump `FF 25`) pointing to the EDR's monitoring DLL⁷.

To circumvent this interception, adversaries developed the technique of Direct System Calls¹⁵. Instead of invoking high-level subsystems (`kernel32!VirtualProtect`) or jumping directly into `ntdll.dll`, the attacker embeds an assembly stub within their own malicious binary that reconstructs the native kernel transition: `mov r10, rcx mov eax, syscall ret`. This routine triggers the CPU instruction `syscall`, transitioning the processor from unprivileged ring 3 execution directly into ring 0 via the kernel dispatcher `KiSystemCall64`, bypassing the EDR's user-mode hooks entirely⁹.

However, direct `syscall` implementations encountered an operating system barrier: System Service Numbers (SSNs) are dynamic across Windows releases and minor build updates⁸. This challenge led to the evolution of dynamic runtime SSN resolution algorithms¹¹:

Hell's Gate: This algorithm parsed the dynamic export directory of the loaded `ntdll.dll` in memory to locate the target native API stub, subsequently parsing its opcodes to extract the immediate SSN value from the `mov eax, <imm32>` instruction (`4C 8B D1 B8 ?? 00 00 00`)⁸. However, when an EDR hooked the function, the initial opcodes were overwritten with an `E9` or `FF 25` jump, corrupting the expected pattern and causing Hell's Gate to fail⁸.

Halo's Gate: To counter hook-induced corruption, Halo's Gate implemented an iterative neighborhood displacement search⁸. If the target function stub was hooked, the engine scanned upward and downward within `ntdll.dll`'s `.text` address space to locate adjacent, unhooked sibling stubs⁸. Because Windows NT compilers generate `syscall` stubs sequentially, Halo's Gate calculated the target SSN by adding or subtracting the function distance offset from an unhooked neighbor⁸.

Tartarus' Gate: This technique extended Halo's Gate by identifying secondary hooking variations, such as indirect memory jumps and hooks positioned slightly deeper within the stub (e.g., after the initial `mov r10, rcx` sequence, where opcodes show `4C 8B D1 E9`), following the trampoline pointers to resolve the target SSN⁷.

FreshyCalls: Recognizing that inspecting in-memory opcode bytes is inherently vulnerable to hook interference, FreshyCalls introduced address-sorting resolution⁸. FreshyCalls iterates over the Export Address Table (EAT) of `ntdll.dll`, filters all exports prefixed with `Nt` or `Zw`, and sorts these functions in ascending

order based on their Virtual Addresses (VAs)⁸. Because the Windows PE linker allocates system call stubs sequentially in memory matching their assigned SSN values, the sorted index of the target function corresponds to its runtime System Service Number⁸. This approach operates reliably without reading modified stub bytes⁸.

RecycledGate: Combining the export address sorting of FreshyCalls with the opcode inspection of Hell's Gate, RecycledGate cross-validates the candidate SSN across multiple parameters, establishing high reliability in aggressively instrumented operating systems⁸.

Hardware Breakpoint Extraction: This method configures CPU Debug Registers (DR0–DR3) to set hardware execution breakpoints on target `ntdll` stubs coupled with a Vectored Exception Handler (VEH)⁸. When execution hits the breakpoint prior to the `syscall` instruction, the VEH catches the `EXCEPTION_SINGLE_STEP` interrupt, extracts the active SSN directly from the EAX hardware register, clears the debug flag, and resumes thread execution³⁹.

Despite solving the SSN discovery problem, basic Direct System Calls introduced distinct telemetry anomalies⁴⁰. When a processor executes the `syscall` instruction, the instruction pointer (RIP) points to the location of that instruction⁴².

In a legitimate execution flow, the `syscall` occurs within the boundaries of the mapped `ntdll.dll` code section⁴⁰.

When executing direct `syscalls`, the RIP at the moment of kernel transition points inside the unbacked or non-module private memory of the calling executable or injected shellcode⁴⁰.

EDR telemetry providers, operating via kernel tracing, inspect the calling RIP and the user-mode call stack; encountering a system call whose origin is outside `ntdll.dll` serves as a high-fidelity indicator of malicious activity⁷.

To counter this detection vector, the offensive research community developed Indirect System Calls¹⁵. In an indirect `syscall` framework (e.g., `SysWhispers2/3/4`, `HookChain`), the attacker configures the SSN in the EAX register within their own stub, but avoids executing the `syscall` instruction locally¹⁵.

Instead, the stub jumps to an existing, valid `syscall`; `ret` instruction gadget located within the body of a legitimate function in `ntdll.dll`⁴³.

Consequently, when the kernel dispatcher inspects the execution context, the system call originated from an authorized memory range inside `ntdll.dll`, satisfying basic RIP-origin validation heuristics¹⁵.

Dynamic Library Remapping and User-Space API Unhooking

An alternative evasion strategy avoids manual `syscall` generation entirely, focusing instead on neutralizing the EDR's hooks directly within memory—a methodology known as user-space API unhooking¹⁴. Because user-mode hooks reside in volatile process memory, an attacker with write access to their own address space can restore the modified functions to their clean, original states³⁹.

The primary vector for API unhooking involves manual library remapping⁸. The malware retrieves a clean copy of `ntdll.dll` directly from the host filesystem (e.g., reading `\SystemRoot\System32\ntdll.dll` via raw disk I/O) or maps the clean image from the kernel's internal `\KnownDlls\` object directory⁸.

Once mapped into private memory, the malware parses the clean PE section headers, identifies the boundaries of the .text executable code section, and uses VirtualProtect (or an equivalent native call) to change the memory protection of the active process's hooked ntdll.dll .text section to PAGE_EXECUTE_READWRITE³⁹.

The malware then copies the pristine code bytes over the hooked memory regions, overwriting the EDR's jump trampolines and restoring the original operating system function stubs³⁹.

A refined iteration of this principle is the Perun's Fart evasion technique⁴⁶. In this technique, the adversary does not read the binary from disk, bypassing filesystem filter drivers and anti-tampering heuristics⁵¹.

Instead, the malware invokes CreateProcess (or NtCreateProcessEx) with flags such as CREATE_SUSPENDED to spawn a legitimate decoy process⁵¹.

In Windows, process creation initializations execute in stages: while the kernel loads the core image and maps ntdll.dll, third-party EDR software modules have not yet injected their user-mode inspection DLLs or applied inline hooks⁵¹.

The malware reads the pristine .text section directly from the suspended child process's address space via ReadProcessMemory (or NtReadVirtualMemory), maps those bytes over its own corrupted memory, and terminates the decoy process⁵¹.

With the EDR's inline hooks removed, the process can execute native Windows APIs without triggering user-space interception logic⁵¹.

Memory Manipulation, Call Stack Spoofing, and Sleep Obfuscation

As detection engines shifted from static opcode verification to deeper context-based telemetry, security tools integrated dynamic memory scanning and call stack unwinding into their evaluation routines⁷. Defensive monitors scan running process heaps and private memory allocations for anomalous characteristics, such as unbacked memory (executable memory regions marked PAGE_EXECUTE_READWRITE that are not mapped from a valid PE image on the disk) and suspicious call stack chains³⁰.

When a thread triggers an operating system event, EDR sensors trace the user-mode execution chain backward through the stack frame using functions like RtlVirtualUnwind and RtlCaptureStackBackTrace⁷.

Under normal application behavior, the call stack exhibits a hierarchical trail of return addresses leading from native stubs, through subsystems like kernel32.dll, down to the executable's image base⁵⁷.

If an attacker executes an indirect system call or invokes APIs from injected shellcode, the unwound call stack reveals an anomaly: a return address pointing into unbacked, dynamically allocated private memory, exposing the presence of an unauthorized execution bridge⁴².

To circumvent call stack inspection, adversaries developed Call Stack Spoofing techniques⁵⁷. Early iterations pushed arbitrary return addresses onto the stack frame to mislead frame-pointer walking heuristics⁵⁸.

However, modern 64-bit operating systems enforce structured exception handling (SEH) unwinding based on .pdata and runtime function tables, rendering crude address modifications prone to system crashes or detection¹⁰.

Advanced call stack spoofing frameworks (such as SilentMoonwalk, VulcanRaven, and Draugr) fabricate synthetic call stacks⁷:

These frameworks locate valid call-site gadgets within legitimate modules⁵⁸.

They align the stack pointer (RSP) and construct valid unwind records that mimic typical Windows execution patterns (e.g., simulating standard thread creation sequences originating from ntdll!RtlUserThreadStart to kernel32!BaseThreadInitThunk)⁵⁷.

Recent evasion research introduced Bring Your Own Unwind Data (BYOUD) and gap exploitation⁵⁹. By analyzing the internal mechanics of RtlVirtualUnwind, researchers discovered that when the Windows unwinder encounters an instruction address lacking a corresponding RUNTIME_FUNCTION entry in the module's .pdata section, it defaults to treating the frame as a leaf function, advancing RSP by exactly 8 bytes without raising an exception⁵⁹.

Adversaries exploit these unwinder fallback states, chaining ghost frames and utilizing specialized x64 machine frame opcodes (such as opcode 10, UWOP_PUSH_MACHFRAME in KiUserExceptionDispatcher) to achieve stack pointer manipulation, hiding malicious execution frames from dynamic unwinding engines⁵⁹.

Concurrently, memory-based detection led to the emergence of Sleep Obfuscation primitives⁷. Traditional Command-and-Control (C2) implants remain idle in memory between beacon check-ins by calling standard sleeping APIs (e.g., Sleep or WaitForSingleObject)⁶³. During these extended idle intervals, the implant's payload resides in memory in decrypted plaintext, leaving it vulnerable to periodic memory sweeps from forensic scanners like PE-sieve or Moneta⁶⁶.

Sleep obfuscation frameworks—exemplified by Ekko, FOLIAGE, Zilean, and DreamWalkers—evade these periodic memory scans by encrypting the implant during idle intervals⁷.

The implementation details of this tradecraft follow a structured sequence:

The implant creates a Waitable Timer or leverages an asynchronous Timer Queue (CreateTimerQueueTimer) configured to fire after the desired idle period³⁹.

The implant prepares a sequence of Return-Oriented Programming (ROP) gadgets and structures a set of Windows CONTEXT records³⁹.

Through APIs such as NtContinue, the thread execution context is shifted without triggering conventional thread monitoring callbacks³⁹.

The chained timers invoke VirtualProtect (or NtProtectVirtualMemory) to strip executable permissions from the implant's memory space, downgrading it to PAGE_READWRITE to hide it from scanners auditing executable memory³⁹.

A system cryptographic primitive—such as SystemFunction032 or an internal RC4/AES algorithm—is triggered to encrypt the implant's code and data sections in place³⁹.

The thread enters an alertable sleep state (NtDelayExecution)³⁹.

When the timer expires, an asynchronous procedure call (APC) executes the reverse sequence: decrypting the memory region, restoring PAGE_EXECUTE_READWRITE permissions, and resuming active payload execution³⁹.

To defensive memory scanning tools, the dormant thread appears non-executable, showing no exposed heuristic signatures or suspicious executable section attributes⁵⁶.

Fileless Exploitation, Memory Injection, and Process Impersonation

Adversaries have moved away from writing compiled binaries directly to disk, pivoting to fileless execution strategies, in-memory reflective loading, and process impersonation attacks⁵².

By operating within volatile memory, malicious payloads minimize the forensic footprints that trigger traditional filesystem scanning engines⁵³.

Reflective DLL Injection (rDLL) bypasses the native Windows operating system loader (LdrLoadDll)¹⁰. Instead of writing a library to the storage medium and calling LoadLibrary, the malware allocates private memory within a target process, copies the sections of the DLL into that space, parses and applies base relocation fix-ups, binds the Import Address Table (IAT) by resolving exported module addresses, and jumps to the DLL's entry point (DllMain)¹⁰.

To evade detection heuristics that flag unbacked executable allocations created via VirtualAlloc, operators employ Module Stomping (also termed DLL Hollowing)⁵⁷.

In this technique, the attacker loads a legitimate, signed, benign DLL from the system directory, changes the page permissions of its .text section, and overwrites the legitimate code with malicious shellcode⁵⁷.

Because the memory region remains formally backed on disk by a valid file handle, basic memory scanning tools evaluate the memory space as legitimate module data⁵⁷.

Beyond payload execution, adversaries exploit structural timing gaps within the Windows process initialization architecture, manipulating the execution metadata visible to endpoint security products⁶⁶:

Process Hollowing: The adversary spawns a legitimate operating system binary (e.g., svchost.exe) in a suspended state, unmaps its primary executable section using NtUnmapViewOfSection, allocates replacement memory at the same base address, writes a malicious payload, realigns thread contexts, and resumes execution⁵². While historically effective, modern EDRs detect hollowing by monitoring unmapping APIs and identifying mismatches between the Process Environment Block (PEB) image base and the underlying file mappings⁵².

Process Doppelgänger: Disclosed to exploit the Windows Transactional NTFS (TxFS) subsystem, this technique allows an attacker to initiate an NTFS transaction, write a malicious executable into the transacted file, create an image section object (SEC_IMAGE) from the transaction handle, and subsequently roll back the transaction⁶⁹. The malicious payload is never committed to the filesystem, leaving no file trace on disk, yet the operating system creates an execution process from the transacted section⁶⁹.

Process Herpaderping: This evasion primitive exploits the temporal gap between process creation and the activation of kernel-mode notifications⁷⁰. In Windows, when a process is

created via native APIs (NtCreateSection and NtCreateProcessEx), the operating system maps the file into memory as an image section⁷⁰. Crucially, kernel process creation callbacks (PsSetCreateProcessNotifyRoutineEx) do not fire when the process object is created; rather, they are triggered when the initial thread is inserted into the process via NtCreateThreadEx⁵². In a Herpaderping attack, the adversary writes malicious bytes to a file on disk, maps the file as an image section, creates the process object, and then, using the still-open writable file handle, overwrites the payload on disk with benign data (or replaces it with a clean binary such as cmd.exe or calc.exe)⁷³. Only then does the adversary call NtCreateThreadEx to spawn the initial thread⁷³. When the EDR's kernel callback fires, the security sensor inspects the file path on disk to compute cryptographic hashes and run heuristic checks, evaluating the benign, overwritten file rather than the malicious binary executing in memory⁵².

Process Ghosting: Building upon the structural limitations exploited by Doppelgänger and Herpaderping, Process Ghosting circumvents the security mechanisms that lock active binaries⁵². Under normal conditions, Windows prevents programs from modifying or overwriting an executable file that is currently executing⁶⁶. However, Process Ghosting exploits the delete-pending file disposition state⁵². The malware creates a new temporary file on disk and puts it into a delete-pending state by invoking NtSetInformationFile with the FileDispositionInformation flag set to true⁶⁶. The malicious payload is written into this delete-pending file, and an image section (SEC_IMAGE) is created from the file handle⁶⁶. The malware then closes the file handle; because the file is marked for deletion and has no remaining active file handles, the operating system deletes the file from the filesystem index⁵². Crucially, the section object remains valid in kernel memory, maintaining an internal reference to the underlying storage blocks⁵². The malware instantiates a process from this section object and inserts a thread⁶⁶. The process executes from an orphaned section backed by a file that no longer exists on disk, depriving filesystem filters and endpoint scanners of an accessible file target to scan⁵².

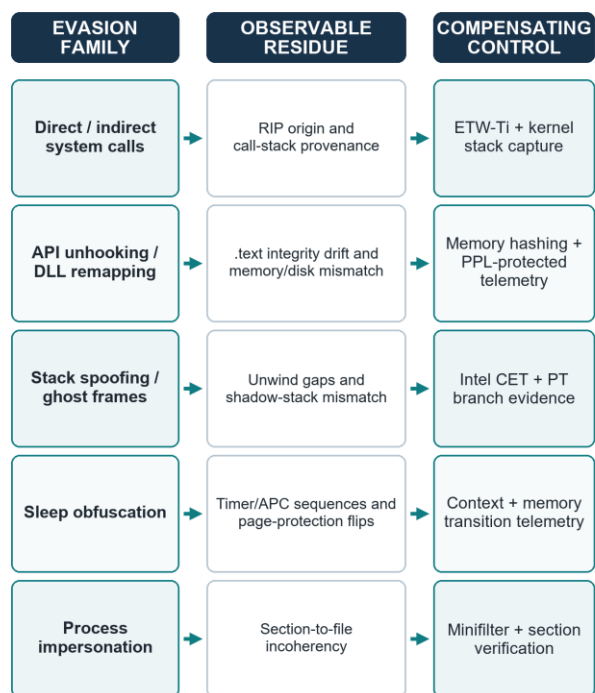


Figure 2. Observable residue and compensating controls across major EDR evasion families.

4. Next-Generation Defensive Countermeasures and Kernel Observability

Deep Kernel Instrumentation: ETW-Ti and System Call Auditing

Recognizing that user-mode hooking is fundamentally vulnerable to evasion and that Kernel Patch Protection (PatchGuard) prohibits third-party security vendors from modifying the System Service Descriptor Table (SSDT) to inspect kernel routines directly, operating system developers introduced dedicated kernel telemetry frameworks⁷. In the Windows ecosystem, the Microsoft-Windows-Threat-Intelligence (ETW-Ti) provider serves as the foundational telemetry source for next-generation endpoint security⁷.

ETW-Ti is integrated directly into the kernel image (ntoskrnl.exe), operating beneath user-mode controls⁷. Unlike standard user-mode Event Tracing for Windows (ETW) providers—which adversaries can disable or blind by patching the user-mode function `ntdll!EtwEventWrite`—ETW-Ti events are generated and dispatched within the ring 0 kernel path itself⁷.

When critical native operations occur, internal kernel routines fire dedicated logging callbacks (e.g., `nt!EtwTiLogAllocExecVm` on executable memory allocations, `nt!EtwTiLogSetContextThread` on thread context modifications, and `nt!EtwTiLogReadWriteVm` on cross-process memory operations)⁷.

To protect this telemetry stream from unauthorized tampering, ETW-Ti restricts event consumption: consumer processes must execute under the Protected Process Light (PPL) security architecture, specifically bearing the `PsProtectedSignerAntimalware` certificate⁷. Standard administrative or `NT AUTHORITY\SYSTEM` accounts cannot attach to or terminate these telemetry sessions⁷.

The critical capability of ETW-Ti lies in its automated, kernel-driven stack tracing⁵⁸. When an ETW-Ti event fires, the kernel captures a snapshot of the execution call stack directly from kernel space⁷.

Even if an adversary utilizes direct system calls to bypass user-mode hooks, the kernel captures the transition and traces the return addresses back into user space⁴². If the stack unwinds into an unbacked private memory allocation rather than a verified disk-backed PE section, the EDR flags the execution anomaly⁴².

Nevertheless, security research has identified limitations in ETW-Ti telemetry collection mechanisms⁴⁵:

Reverse engineering of internal ETW-Ti dispatch loops reveals that certain stack collection events are handled via asynchronous User-mode Asynchronous Procedure Calls (User APCs) rather than synchronous Kernel APCs⁵⁹.

A User APC delivers only when the executing thread enters an alertable wait state (`SleepEx`, `WaitForSingleObjectEx`), introducing an exploitable timing window between event logging and stack capture⁵⁹.

Furthermore, adversaries can configure hardware breakpoints through native APIs like `NtContinue` rather than `NtSetContextThread`⁴⁵. Because `NtContinue` was designed for exception return flows, it does not trigger the specific `nt!EtwTiLogSetContextThread` telemetry callback, allowing attackers to manipulate debug registers while minimizing ETW-Ti audit footprints⁴⁵.

In response, modern EDR kernels combine ETW-Ti telemetry with kernel-level memory hashing, thread pooling audits, and verified return address validation at the kernel boundary⁷.

Linux Kernel Security: eBPF and BPF-LSM in Cloud-Native Workloads

In Linux and containerized cloud-native environments, endpoint security architectures have evolved along a parallel trajectory³⁷. Traditional host security architectures relied either on periodic user-space polling (e.g., inspecting the `/proc` filesystem), audit systems (e.g., `auditd`, which imposes substantial performance degradation under high syscall throughput), or rigid Mandatory Access Control (MAC) systems such as `SELinux` and `AppArmor`³⁶.

Modern multi-tenant environments—governed by shared-kernel architectures like `Docker` and `Kubernetes`—face distinct vulnerabilities³⁷. Flaws in container runtimes (e.g., the `runc` file descriptor leak in `CVE-2024-21626`, or time-of-check-to-time-of-use race conditions) allow adversaries to break namespace isolation, achieve container escapes, and compromise host kernels³⁷.

To protect dynamic containerized environments without destabilizing operational systems, modern security tools use Extended Berkeley Packet Filter (eBPF) and BPF-LSM (Linux Security Module) technologies³⁷. eBPF provides an in-kernel virtual machine that executes verified, event-driven bytecode safely within the kernel address space without modifying kernel source code or loading unstable third-party kernel modules³⁷.

An in-kernel verifier validates all eBPF programs prior to execution—confirming that the bytecode contains bounded loops, satisfies memory safety bounds, avoids uninitialized register reads, and terminates safely—preventing kernel panics and instability³⁶.

Advanced cloud-native EDR platforms (e.g., Cilium Tetragon, Aqua Tracee, and KubeArmor) attach eBPF programs to diverse kernel hooks, including dynamic kprobes, lightweight function boundaries (fentry/fexit), tracepoints, and socket buffers³⁷.

The primary technological advancement in this domain is BPF-LSM⁷⁴. Historically, eBPF was primarily an observability framework: programs attached to tracepoints or kprobes could inspect system arguments and asynchronously stream telemetry to user space, but could not block malicious actions³⁸.

By integrating eBPF with the centralized Linux Security Module (LSM) framework (supported in mainline Linux kernels), BPF-LSM programs attach directly to security mediation hooks (such as `bprm_check_security`, `file_open`, and `socket_connect`)³⁸.

When a container escape payload attempts to exploit a kernel flaw or execute an unauthorized binary, the attached BPF-LSM hook executes synchronously in kernel space³⁷. The program interrogates runtime parameters—such as container namespace identifiers, process control groups (cgroups), executable path hashes, and active system capabilities—against behavioral policies³⁷.

If an isolation boundary violation is identified, the BPF-LSM hook returns an access denial error code (e.g., `-EPERM`) or immediately issues a synchronous kernel signal (`SIGKILL`) to terminate the offending thread before the system call reaches the operational subsystem³⁷.

Empirical benchmarks demonstrate that this approach incurs minimal overhead—typically under 1% to 2% throughput degradation—providing in-kernel enforcement against container breakouts and privilege escalation without requiring static per-application allowlists³⁷.

Hardware-Assisted Telemetry and Control-Flow Integrity

To counter user-space hook circumvention, stack spoofing, and low-level evasion, modern endpoint defenses increasingly incorporate hardware-assisted execution telemetry, shifting validation directly into physical CPU execution pipelines⁴⁵.

A prominent hardware technology is Intel Processor Trace (Intel PT)⁸⁰. Intel PT captures fine-grained, non-intrusive branch trace telemetry directly from physical processor execution units with low performance overhead (typically 1% to 3%)⁸⁰.

Unlike software tracing tools that rely on binary rewriting, debugger traps, or software hooks, Intel PT logs control-flow instructions—such as conditional branches, indirect branches, function calls, and returns—directly to dedicated physical memory buffers using compact, packetized hardware records (e.g., Taken/Not-Taken (TNT) packets, Target IP (TIP) packets, and Flow Update Packets (FUP))⁸⁰.

Security architectures utilize Intel PT buffers to reconstruct execution flows and validate control-flow integrity⁸⁰. When a thread executes a system call or triggers a critical kernel callback, defensive modules parse the Intel PT trace buffer to reconstruct the preceding instructions⁸⁰.

If the hardware branch trace reveals that execution jumped directly into an internal function without executing the documented entry prologue, or transitioned through an unbacked ROP gadget chain, the system detects the anomaly regardless of whether the software call stack was spoofed⁷.

A complementary hardware security mechanism is Intel Control-Flow Enforcement Technology (CET)⁷⁹. Intel CET provides silicon-level protection against Return-Oriented Programming (ROP) and call stack spoofing through Hardware-Enforced Shadow Stacks⁷⁹.

When CET is active, the CPU maintains two distinct stack representations: the conventional, software-managed call stack (governed by the traditional RSP register) and an internal, isolated Shadow Stack (governed by the CPU's internal Shadow Stack Pointer, or SSP), which can only be modified through dedicated, hardware-restricted call and return instructions⁷⁹.

When an application invokes a `CALL` instruction, the CPU pushes the return address onto both the data stack and the shadow stack⁷⁹. When the corresponding `RET` instruction executes, the physical processor verifies that the return address on the data stack matches the return address stored on the shadow stack⁷⁹.

If an attacker employs synthetic call stack spoofing, ROP gadget chaining, or overwrites stack frames to deceive EDR stack walkers, a discrepancy arises between the corrupted user stack and the immutable hardware shadow stack⁷⁹.

The CPU immediately intercepts this mismatch, triggering a silicon-level control protection exception (`#CP`), terminating the malicious thread before it can execute further instructions⁷⁹.

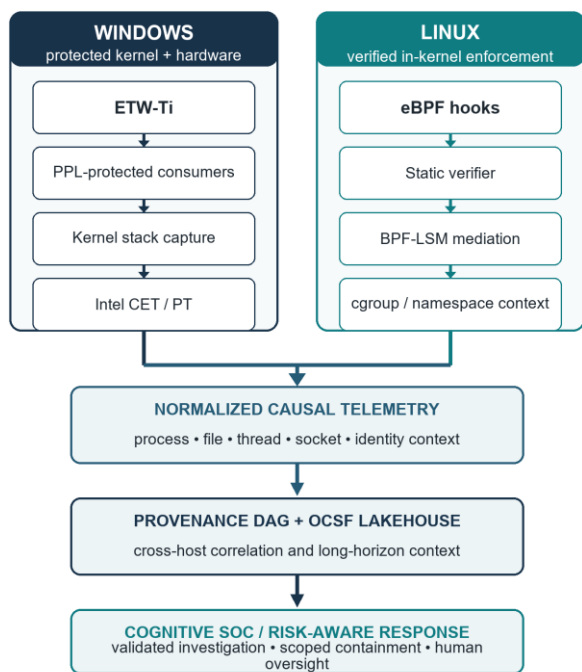


Figure 3. Cross-platform kernel observability converging into provenance-aware XDR analytics.

5. Comparative Structural Matrices and Tactical Syntheses

Table 1. Comprehensive Architectural Evolution Across EDR Generations

Dimension	First-Generation EDR (2013–2016)	Second-Generation EDR (2017–2020)	Third-Generation EDR / XDR (2021–2024)	Next-Generation Cognitive EDR (2025–Present)
Primary Detection Paradigm	Post-execution IOC matching, basic rule-based process telemetry ¹	Static ML, early behavioral sequencing, isolated process anomalies ¹	Dynamic XDR correlation, whole-system data provenance, MITRE ATT&CK mapping ¹⁸	Inductive GNNs, Temporal-Spatial Provenance, Autonomous Agentic SOC loops ¹²
Interception Layer	Ring 3 API hooks (inline JMP in ntdll.dll), basic OS event logs ⁷	Ring 3 hooks, Minifilters, Kernel Callbacks (PsSetCreateProcessNotifyRoutine Ex) ⁷	Kernel ETW-Ti, BPF-LSM, PPL protection, Call-stack unwinding verification ⁷	In-Kernel BPF-LSM, Hardware-Assisted Intel PT/CET, Kernel-Provenance DAGs ³⁷
Analytics Architecture	Centralized on-premises relational databases, batch polling ²	Hosted SaaS, proprietary SIEM indexers, periodic agent log uploads ⁸	Cloud-Native streaming (Kafka/gRPC), Security Data Lakes, OCSF schema alignment ¹⁸	Decoupled Data Lakehouses, Substructure Graph Reduction, Hybrid LLM Agents ¹¹
Average Threat Latency	Minutes to hours; high dependency on local agent batch sweeps ⁴	Tens of seconds to minutes; manual analyst queue triage ⁸	Real-time event streaming; minutes to hours for cross-source investigation ¹²	Sub-second edge blocking; sub-two-minute full cognitive triage and response ²⁸
Resilience to Evasion	Critically Low; bypassed by direct syscalls and basic memory injection ⁹	Low; vulnerable to unhooking (Perun's Fart), sleep masking, and stack manipulation ⁵⁵	Moderate; detects direct syscalls and unbacked memory, challenged by ghost frames ⁵⁹	High; silicon-enforced CET, kernel ETW-Ti, synchronous in-kernel BPF-LSM ⁷⁴

Table 2. User-Mode Syscall Resolution and Invocation Paradigms

Technique / Framework	SSN Resolution Mechanism	Kernel Invocation Pattern	Hook Bypass Capability	Residual Telemetry Artifacts / Detection Vectors
Hell's Gate	In-memory pattern scanning of ntdll.dll function stub opcodes ⁸	Direct user-mode syscall stub embedded in payload binary ⁸	Bypasses unhooked functions; fails completely if target stub is hooked ⁸	RIP points to unbacked executable memory; abnormal call stack lacks legitimate frames ²⁰
Halo's Gate	Iterative neighborhood displacement search scanning adjacent stubs ⁸	Direct user-mode syscall stub embedded in payload binary ⁸	Evades basic inline hooks; fails if large sequential stub clusters are hooked ⁸	Syscall origin validation alerts; unbacked private memory traces in thread context ⁴²
Tartarus' Gate	Scans for indirect trampolines (FF 25) and displaced instructions ⁷	Direct user-mode syscall stub embedded in payload binary ⁸	Resolves complex trampolines and non-standard hook jump patterns ⁷	Ring 0 stack unwind exposes return address inside unbacked allocation ⁴²
FreshyCalls	Ascending export Virtual Address (VA) sorting across ntdll exports ⁸	Direct or Indirect syscall; ret gadget execution ⁸	Immune to opcode modification; resolves SSNs regardless of hook state ⁸	EAT enumeration footprints; detected by ETW-Ti kernel call stack tracing if direct ⁴³
RecycledGate	Cross-validates FreshyCalls VA sort index against opcode verification ⁸	Indirect invocation through benign ntdll gadget addresses ⁸	Highly resilient; detects reordered stubs or hooked export table entries ⁸	Requires stack spoofing to defeat deep ETW-Ti call-stack unwind analysis ³⁹
Hardware Breakpoints	Configures DR0–DR3 debug registers and captures SSN in VEH from EAX [8, 39, 45]	Indirect invocation through benign ntdll gadget addresses ⁸	Bypasses reading potentially monitored function opcode bytes ⁸	Setting thread context triggers nt!EtwTiLogSetContextThread unless masked via NtContinue [45]

Table 3. Process Impersonation and Fileless Execution Mechanics

Attack Vector	Primitive Mechanics	Operating System Artifacts	Primary EDR Interception Point	Countermeasure Evasion Factor
Process Hollowing	Spawns suspended process, calls <code>NtUnmapViewOfSection</code> , maps malicious PE ⁵²	Suspended process, unmapped image section, replacement private pages ⁵²	<code>NtUnmapViewOfSection</code> telemetry, PEB image base mismatch against VAD tree ⁵²	Easily detected by modern EDR memory scanners comparing disk PE to memory ⁵²
Process Doppelgänger	Creates transacted file via <code>TxF</code> , writes PE, creates <code>SEC_IMAGE</code> , rolls back ⁵³	Transacted NTFS objects, anonymous section handles, uncommitted files ⁵³	Minifilter monitoring of <code>IRP_MJ_FILE_SYSTEM_CONTR</code> OL transacted operations ⁵²	Bypasses basic on-access filesystem scanning; no malicious artifact touches permanent disk ⁵³
Process Herpaderping	Maps image section from file, creates process, overwrites on-disk file with decoy ⁵⁴	Discrepancy between mapped section bytes and on-disk file hash ⁵⁴	Kernel minifilter post-cleanup verification; section-to-file byte coherency checks ⁵²	Exploits callback latency; kernel notify routine fires on thread creation, viewing clean disk bytes ⁵⁴
Process Ghosting	Sets <code>FileDispositionInformation</code> (delete-pending), writes PE, maps section, deletes ⁵²	Process running from deleted file; section object referencing unlinked cluster ⁵²	Kernel callback monitoring section creation from delete-pending file handles ⁵²	Leaves zero accessible file objects on disk; filesystem scanners fail when attempting to open path ⁵²
Module Stomping	Loads legitimate signed DLL, overwrites <code>.text</code> with shellcode via <code>VirtualProtect</code> [57, 64]	Legitimate DLL mapped, but <code>.text</code> section modified to <code>PAGE_EXECUTE_READWRITE</code> [57, 64]	Memory scanner integrity check comparing in-memory <code>.text</code> against clean disk image ⁵²	Code appears backed by a verified disk module, defeating naive unbacked memory heuristics ⁵⁷
Sleep Obfuscation (Ekko)	Timer Queue invokes ROP chain: encrypts memory, sets <code>PAGE_READWRITE</code> , sleeps ⁵⁵	Non-executable memory pages, waitable timer objects, modified thread contexts ⁵⁵	Periodic thread context inspection; identifying sleeping threads pointing to non-executable pages ⁵⁵	Payload resides entirely encrypted during idle intervals; evades point-in-time memory sweeps ⁵⁵

Table 4. Modern Defense Countermeasures vs. Advanced Evasion Techniques

Defensive System	Operational Execution Realm	Monitored Telemetry Artifacts	Target Adversarial Tradecraft	Architectural Resilience & Boundary Guarantees
ETW-Ti (Threat Intel)	Ring 0 Kernel (<code>ntoskrnl.exe</code>) ⁷	Native API calls, cross-process read/write, thread contexts, kernel stack frames ⁷	Direct System Calls, User-Mode Unhooking, APC Injection ⁴²	Protected by PPL; unmodifiable by user-mode hooks; vulnerable to APC timing gaps ⁷
BPF-LSM (Linux)	Ring 0 Kernel (LSM Framework) ³⁸	In-kernel security hooks (<code>bprm_check</code> , <code>file_open</code>), cgroup IDs, namespaces ³⁷	Container breakouts (runc CVEs), privilege escalation, unauthorized binaries ³⁷	Statically verified, synchronous execution; blocks malicious syscalls with minimal userspace delay ³⁷
Provenance-based EDR (P-EDR)	Cloud Analytical Fabric / Lakehouse ¹⁷	Causal directed acyclic graphs of process, file, thread, and socket dependencies ¹⁷	Living-off-the-Land (LotL), multi-stage APTs, lateral spread ¹⁷	Resolves dependency explosion via Jaccard graph reduction; tracks causality across processes ¹⁷
Intel CET (Shadow Stack)	Physical Hardware Execution Unit ¹⁹	Hardware Shadow Stack Pointer (SSP), return address validation on <code>CALL/RET</code> [79]	Call Stack Spoofing, ROP gadget exploitation, execution hijacking ⁷⁹	Hardware-enforced; cannot be bypassed by software memory manipulation or spoofing ⁷⁹
Intel PT (Processor Trace)	Physical Hardware Execution Unit ⁶⁰	Packetized hardware control flow, Taken/Not-Taken branch execution buffers ⁶⁰	Syscall trampolines, code injection, JIT shellcode, unbacked execution bridges ⁷	Silicon-level tracing; operates independently of OS software hooks ⁶⁰
Agentic Cognitive SOC	Cloud Distributed Microservices ¹²	Normalized OCSF telemetry, alert graphs, threat intelligence, identity models ¹²	Alert fatigue, fragmented attacks across domains, complex evasion techniques ¹²	Replaces rigid SOAR; combines deterministic simulation engines with LLM reasoning ¹²

Table 5. Comparison of Provenance-Based Intrusion Detection Architectures

System Architecture	Ingestion Data Source	Analytical / Machine Learning Paradigm	Temporal & Dynamic Awareness	Graph Reduction Strategy	Kill-Chain & ATT&CK Alignment
SLEUTH ³²	Linux audit logs, BSD audit frameworks	Dynamic provenance tracking via causal tag propagation and dependency parsing	Coarse temporal tracking; reconstructs historic causal graphs offline	Static dependency pruning based on trust boundaries and domain tags	Implicit; tracks confidentiality and integrity tag corruption paths
UNICORN ¹⁷	Linux audit framework, CamFlow provenance capture	Streaming graph kernel evaluation; compact fixed-size histogram sketching	Continuous runtime analysis; captures long-running execution states	Dynamic graph decay; summarizes system behavior into structural histograms	Unsupervised anomaly detection; identifies divergence from normal system execution
POIROT ¹⁷	Host system audit records, enterprise endpoints	Inexact subgraph matching over Cyber Threat Intelligence (CTI) graphs	Evaluates spatial relationships across pre-recorded audit logs	CTI-guided filtering; searches for structural indicator patterns	Explicit mapping to CTI graphs and adversary attack specifications
HOLMES ³²	Auditd system call streams, operating system events	High-Level Scenario Graph (HSG); maps low-level events to high-level abstractions	Stateful kill-chain phase progression modeling in near-real-time	Filters noise by aggregating micro-events into macro-level scenario nodes	Explicit MITRE ATT&CK alignment across attack kill-chain phases

System Architecture	Ingestion Data Source	Analytical / Machine Learning Paradigm	Temporal & Dynamic Awareness	Graph Reduction Strategy	Kill-Chain & ATT&CK Alignment
Kairos ¹⁷	DARPA TC datasets, whole-system kernel provenance	Temporal-spatial Graph Neural Network (GNN) encoder-decoder architecture	Fine-grained temporal graph learning over streaming dynamic graphs	Edge-level anomaly scoring; extracts compact summary attack footprints	Causal footprint reconstruction; maps multi-host enterprise intrusions
ThreaTrace ³³	Whole-system kernel logs (Windows, Linux, macOS)	Inductive GNN using tailored GraphSAGE over system entity nodes	Edge-by-edge real-time classification as events are logged	Evaluates node neighborhood roles without storing full historic graphs	Entity-level anomaly detection; identifies specific compromised objects
Sylph ³³	System audit logs, provenance graph transformations	Unsupervised graph embeddings over time-sliced dynamic subgraphs	Fine event granularity with temporal subgraph segmentation	Time-sliced subgraph partitioning; reduces memory occupation at runtime	Localizes APT intrusions to specific events and entity sub-graphs

6. Strategic Synthesis and Future Trajectories

The evolution of endpoint detection and response over the past decade reflects a persistent competition between evasion tradecraft and defensive observability¹. The historical model—characterized by unprivileged user-mode API hooking and isolated heuristic evaluation—has proven structurally inadequate against adversaries utilizing direct and indirect system calls, dynamic library remapping, synthetic call stack spoofing, sleep-state memory encryption, and process metadata manipulation¹⁵.

In response, modern defensive architectures have executed three fundamental structural shifts:

First, interception authority has shifted downward from unprivileged ring 3 user space into ring 0 operating system kernels and physical CPU hardware⁷. Software hooks in user memory have been superseded by kernel telemetry frameworks like Microsoft-Windows-Threat-Intelligence (ETW-Ti) on Windows and synchronous in-kernel enforcement engines like BPF-LSM on Linux⁷. By embedding security validation directly into native system call dispatch and security module routines, operating systems ensure that evasive user-mode mechanics cannot circumvent recording or mitigation⁷. Furthermore, software telemetry is increasingly anchored in physical silicon⁷⁹. Hardware mechanisms like Intel Control-Flow Enforcement Technology (CET) provide hardware-enforced shadow stacks that mitigate call stack spoofing and ROP execution at the CPU level, while Intel Processor Trace (Intel PT) provides branch execution tracing that verifies code provenance without relying on software hooks⁷⁹.

Second, the operational data fabric has transitioned from isolated endpoint log stores toward federated XDR architectures and enterprise Security Data Lakes¹⁸. By standardizing on the Open Cybersecurity Schema Framework (OCSF) and decoupling storage from compute using columnar formats like Apache Parquet and Delta Lake, modern enterprises eliminate data normalization overhead and can retain petabytes of full-fidelity telemetry over multi-year observation windows¹⁸. This architectural shift enables defenders to correlate disparate signals—linking endpoint process creation with identity provider anomalies, cloud infrastructure modifications, and network egress flows—to identify complex, multi-stage intrusions¹⁸.

Third, threat analysis is transitioning from reactive, human-constrained queue management to automated cognitive investigation frameworks¹¹. By combining Graph Neural Networks (which evaluate whole-system causal provenance graphs while resolving dependency explosion through

mathematical clustering) with multi-agent cognitive triage systems (such as AgentSOC), security operations can mitigate alert fatigue¹². Integrating deterministic structural simulation engines with Large Language Model reasoning loops enables these cognitive platforms to safely filter out over 80% of benign false positives, construct validated causal attack narratives, and execute targeted, sub-two-minute containment actions¹¹.

The future of endpoint resilience rests on this integrated architecture: a zero-trust endpoint framework where physical hardware guarantees execution integrity, in-kernel modules maintain visibility over system transitions, cloud data lakes preserve causal context, and autonomous cognitive engines coordinate defenses against sophisticated adversarial techniques⁵.

References

- Kaur, H., Sanjaiy, D., Naveen, K., et al. "Evolution of Endpoint Detection and Response (EDR) in Cyber Security: A Comprehensive Review." *E3S Web of Conferences*, vol. 556, 2024, article 01006.¹
- Arfeen, A., Ahmed, S., & Jafri, S. A. H. "Endpoint Detection & Response: A Malware Identification Solution." *Proceedings of the International Conference on Cyber Warfare and Security (ICWS)*, 2021.²
- Kumar, N., Kasbekar, G. S., & Manjunath, D. "Application of Data Collected by Endpoint Detection and Response Systems for Implementation of a Network Security System based on Zero Trust Principles and the EigenTrust Algorithm." *arXiv preprint arXiv:2203.09325*, 2022.⁵
- Lee, S.-J., Jeon, S., & Lee, I.-G. "A Machine Learning-Enhanced Endpoint Detection and Response Framework for Fast and Proactive Defense Against Advanced Cyber Attacks." *Soft Computing*, 2022.²
- Smith, J., & Johnson, A. "A Survey of Endpoint Detection and Response (EDR) Systems." *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 1, pp. 120–133, 2020.⁸⁵
- SafeGuard Research Consortium. "Lightweight Three-Tier Client-Server Architecture for Real-Time Endpoint Monitoring and Administrative Response." *arXiv preprint arXiv:2607.10027*, 2026.⁶
- Industry Expert Consortium. "Evaluating the Efficacy and Operating Costs of Provenance-Based Endpoint Detection and Response (P-EDR) Systems." *arXiv preprint arXiv:2307.08349*, 2023.¹⁶
- ShellForge Research Group. "Benchmarking EDR Behavioral Detection Robustness Against Genetic Algorithm Mutated Payloads." *arXiv preprint arXiv:2607.07191*, 2026.³

- Júnior, H. "HookChain: Advanced EDR Bypass via Unhooked Neighboring Indirect System Calls." *Red Team Leaders Research Disclosures*, 2023.¹⁵
- EDR Evasion Research Labs. "Direct and Indirect Syscalls: Overcoming Modern Windows Telemetry." *arXiv preprint arXiv:2608.01639*, 2026.¹³
- Vaadata Security Research. "Analysis of Modern Antivirus and EDR Bypass Techniques." *Technical Disclosures*, 2023.⁴⁰
- Santos, J. A. "SysWhispers4: Advanced System Service Number Resolution and Invocation Strategies." *Offensive Security Tool Documentation & Source Release*, 2024.⁸
- Praetorian Engineering. "Bypassing ETW Threat Intelligence Telemetry via Hardware Breakpoint Context Manipulation with NtContinue." *Praetorian Security Research*, 2023.⁴⁵
- Valhguard Systems. "Call Stack Spoofing: How Spoofers Beat the Unwinder and Where the Truth Still Lives." *Technical Research Series*, 2026.⁶⁰
- Nighthawk Development Team. "API Call Stack Masking and Kernel ETW-Ti Unwind Correlation." *Nighthawk Technical Whitepapers*, 2023.⁶⁴
- Smoky Security. "In-Direct System Calls and Call Stack Memory Tracing in Modern EDR." *Smoky Security Disclosures*, 2023.⁴²
- Oxdbgman Research Group. "EDR Internals: Kernel Callbacks, Minifilters, ETW-Ti Architecture, and Dynamic Bypass Mechanics." *Oxdbgman Technical Publications*, 2024.⁷
- Avantguard Research. "Threadless Ops II: Synthetic Stack Spoofing and Module Stomping Telemetry Evasion." *Avantguard Technical Reports*, 2024.³⁷
- Maz, Ox. "The LACUNA Chain: Ghost Frames and Exploiting Undocumented Leaf-Function Fallbacks in RtlVirtualUnwind." *Offensive Security Monographs*, 2024.⁵⁹
- Binary Defense Labs. "Understanding Sleep Obfuscation: ROP-Chains, Memory Scanners, and Ekko Telemetry Evasion." *Binary Defense Intelligence Reports*, 2023.⁵⁶
- Cobalt Strike Security Team. "Behind the Mask: Spoofing Call Stacks Dynamically with Timers." *Cobalt Strike Technical Disclosures*, 2023.⁸¹
- CovertSwarm Research. "A Comprehensive Timeline of Modern EDR Bypass Techniques." *CovertSwarm Whitepapers*, 2024.⁶³
- Sektor7 Institute. "Perun's Fart: Restoring Unhooked System Call DLLs from Suspended Decoy Processes." *Sektor7 Research Reports*, 2021.⁵⁵
- Landau, G. "Process Ghosting: Bypassing Antimalware Scan Buffers Using Delete-Pending Executable Section References." *Elastic Security Research Disclosures*, 2021.⁵²
- Microsoft Defender Research Team. "Using Process Creation Properties and Execution Sequences to Detect Process Herpaderping, Doppelgänger, and Ghosting." *Microsoft Security Blog*, 2022.⁷⁰
- CrowdStrike Falcon Intelligence. "Process Herpaderping: Exploiting the Temporal Gap Between Image Section Mapping and Thread Creation Callbacks." *CrowdStrike Research Publications*, 2020.⁷³
- Bitdefender Labs. "Process Impersonation and Image Incoherency Detection via GravityZone Process Protection." *Bitdefender Technology Explainers*, 2026.⁵²
- Open Cybersecurity Schema Framework Steering Committee. "Open Cybersecurity Schema Framework (OCSF) Specification and Reference Implementation." *Linux Foundation Technical Reports*, 2023–2024.¹⁸
- Intel Corporation. "Intel 64 and IA-32 Architectures Software Developer's Manual: Hardware-Enforced Control-Flow Enforcement Technology (CET) and Intel Processor Trace (Intel PT)." *Intel Technical Specifications*, 2021–2023.⁷⁹
- Patel, K., et al. "KRAKENGUARD: Statically Verified In-Kernel Security and Multi-Tenant Isolation for eBPF Programs." *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2026.⁷⁶
- Isovalent Research Group. "Synchronous In-Kernel Isolation Enforcement and Container Breakout Mitigation via Tetragon and BPF-LSM." *Isovalent & CNCF Whitepapers*, 2024–2025.³⁷
- Eunomia eBPF Community. "The Technical Progress of the eBPF Ecosystem: BPF-LSM, BTF User-Space Unwinding, and Runtime Enforcement." *Eunomia Deep Dives*, 2025.⁷⁷
- Cheng, L., Lv, C., et al. "Kairos: Practical Intrusion Detection and Investigation Using Whole-System Provenance." *arXiv preprint arXiv:2308.05034*, 2023.¹⁷
- Han, W., et al. "UNICORN: Runtime Provenance-Based Detector for Advanced Persistent Threats." *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2020.¹⁷
- Milajerdi, S. M., et al. "POIROT: Aligning Cyber Threat Intelligence with System Provenance Graphs for APT Detection." *Proceedings of the USENIX Security Symposium*, 2019.¹⁷
- Milajerdi, S. M., et al. "HOLMES: Real-Time APT Detection through Correlation of Anomalous Behaviors into High-Level Scenario Graphs." *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019.³²
- Wang, K., et al. "ThreaTrace: Detecting and Tracing Advanced Persistent Threats without Prior Knowledge via GraphSAGE." *IEEE Transactions on Information Forensics and Security*, 2022.³³
- Zhang, Y., et al. "Sylph: A Lightweight Unsupervised APT Detection Method Based on Time-Sliced Provenance Graph Partitioning." *Information*, vol. 16, no. 7, 2025, article 566.⁸³
- AgentSOC Research Group. "AgentSOC: Autonomous Security Operations Centers via Anticipatory Narrative Reasoning and Structural Simulation." *arXiv preprint arXiv:2604.20134*, 2026.¹²
- Schonberg, Z. "Why Large Language Models Alone Are Insufficient for AI SOC Triage: The Critical Need for Hybrid Deterministic Architecture." *Intezer Security Research*, 2025.²⁹
- D3 Security. "Autonomous Reasoning Graphs: Eliminating Playbook Stagnation in Agentic SOC Architectures." *D3 Security Whitepapers*, 2026.²⁶
- Simbian Engineering. "The Agentic SOC: Resolving Alert Fatigue via Autonomous Evidence Gathering and MTTR Reduction." *Simbian AI Publications*, 2025.²⁵
- UnderDefense Security Operations. "Autonomous SOC Implementation Framework: Data Pipelines, Feedback Loops, and Decision Intelligence." *UnderDefense Whitepapers*, 2025.¹¹
- Vectra AI. "Practitioner Reality vs. Agentic Automation: Telemetry Overload, ViB Survey Findings, and Human-in-the-Loop Boundaries." *Vectra AI Strategy Papers*, 2026.²⁷
- Torq Automation Labs. "Autonomous Triage: Combining Deterministic Workflow Integrations with ReAct Agentic Reasoning." *Torq Security Engineering*, 2025.³⁵
- Meacci, L. "Bypassing EDR in a Crystal Clear Way: YARA Scanners, Memory Unbacking, and Draugr Call Stack Synthesis." *Offensive Security Research Disclosures*, 2024.¹⁰

O'Neill, T. "UnwindRaven: Constructing Fully Synthetic Call Stacks via Unwind Codes and Machine Frames." *Offensive Security GitHub Publications*, 2024.⁶¹

KlezVirus. "SysWhispers Is Dead, Long Live SysWhispers: User-Mode Hook Evasion and System Call Architecture." *KlezVirus Blog Series*, 2022.⁸⁹

ProvX Research Team. "ProvX: Explainable Graph Neural Networks for Advanced Persistent Threat Detection on Whole-System Provenance." *arXiv preprint arXiv:2508.06073*, 2025.³¹

Tidal Systems. "Addressing Concept Drift in Provenance-Based Intrusion Detection Systems." *OASICS Proceedings of NINEs*, vol. 139, 2026.⁸⁷

Supplementary Source Links

Evolution of Endpoint Detection and Response (EDR) in Cyber. [Access source](#)
Endpoint Detection & Response: A Malware Identification Solution. [Access source](#)

Evaluating Endpoint Detection Robustness Against Genetic ... - arXiv. [Access source](#)

Evaluating Endpoint Detection Robustness Against Genetic ... - arXiv. [Access source](#)

Application of Data Collected by Endpoint Detection and Response. [Access source](#)

SafeGuard - Time Endpoint Threat Detection and Response - arXiv. [Access source](#)

EDR Tradecraft: Internals, Detection, Evasion & Advanced Research. [Access source](#)

SSN Resolution Methods - SysWhispers4. [Access source](#)

Introduction to SysWhispers4. [Access source](#)

Bypassing EDR in a Crystal Clear Way - Lorenzo Meacci. [Access source](#)

Autonomous SOC Guide: AI Alert Triage, Agentic Response, Vendor. [Access source](#)

AgentSOC: A Multi-Layer Agentic AI Framework for Security ... - arXiv. [Access source](#)

Autonomous Endpoint Evasion via Knowledge-Driven Multi-Agent. [Access source](#)

(PDF) Proactive Detection of Malware Evasion Strategies Using. [Access source](#)

Hookchain Technique Introduction by Helvio Júnior (M4v3r1ck). [Access source](#)

Are we there yet? An Industrial Viewpoint on Provenance-based. [Access source](#)

Kairos: Practical Intrusion Detection and Investigation using Whole. [Access source](#)

What is Security Data Lakes - Bitdefender InfoZone. [Access source](#)

OCSF Celebrates First Anniversary with the Launch of a New Open. [Access source](#)

Amazon Security Lake is now generally available - Stellar Cyber. [Access source](#)

Open Cybersecurity Schema Framework (OCSF) - Deepwatch. [Access source](#)

What is an Intelligent Unification Layer (IUL)? - Anomali. [Access source](#)

Utilizing NETSCOUT Deep Packet Inspection Technology to Enrich. [Access source](#)

Optimizing Delta Lake Security Data Lakehouses - Query.AI. [Access source](#)

What Are AI Agents in SOC? How Agentic SOC Outpace Traditional. [Access source](#)

Agentic SOC Platform | The Future of Autonomous Security Operations. [Access source](#)

Agentic SOC: definition, autonomy claims, and the evidence - Vectra AI. [Access source](#)

AI Agents for Threat Triage, Response, and Orchestration. [Access source](#)

Why the best LLMs are not enough for the AI SOC - Intezer. [Access source](#)

Provenance Graph-Based Deep Learning Framework for APT. [Access source](#)

ProvX: Generating Counterfactual-Driven Attack Explanations ... - arXiv. [Access source](#)

A Review of Provenance Graph based APT Attack Detection. [Access source](#)

Kairos: Practical Intrusion Detection and Investigation using Whole. [Access source](#)

MGDA: A provenance graph-based framework for threat detection. [Access source](#)

How Agentic AI Security Is Shaping the Future of Cybersecurity - Torq. [Access source](#)

Aquila: Efficient In-Kernel System Call Telemetry for Cloud-Native. [Access source](#)

Using ebpf-based tools for detection and mitigation of container. [Access source](#)

BPFGuard: Multi-Granularity Container Runtime Mandatory Access. [Access source](#)

JoasASantos/SysWhispers4: AV/EDR evasion via direct ... - GitHub. [Access source](#)

Antivirus and EDR Bypass Techniques Explained - Vaadata. [Access source](#)

SysWhispers4 - SysWhispers4. [Access source](#)

(In)-Direct System Calls - | Smoky | Blog. [Access source](#)

FreshyCalls - SysWhispers4. [Access source](#)

README.md - gmh5225/rust-mordor-rs - GitHub. [Access source](#)

ETW Threat Intelligence and Hardware Breakpoints | Praetorian. [Access source](#)

EDR Evasion Techniques with Syscalls | PDF | Malware - Scribd. [Access source](#)

HookChain: A Deep Dive into Advanced EDR Bypass Techniques. [Access source](#)

sysplant — AI Skill for Claude Code | SkillTip. [Access source](#)

EDR Unhooking Techniques Explained | PDF | Malware - Scribd. [Access source](#)

Windows Malware Development & Evasion Course | Khubab Ahmed. [Access source](#)

Security Controls: Modern EDR & Windows Protection Bypass. [Access source](#)

What is Process Impersonation – Bitdefender TechZone. [Access source](#)

Ghosts in the Endpoint: How Attackers Evade Modern EDR Solutions. [Access source](#)

Unhook Antivirus Hooks | Evasion Technique Perun Farts - Medium. [Access source](#)

Perun's Fart - yet another unhooking method. [Access source](#)

Understanding Sleep Obfuscation - Binary Defense. [Access source](#)

Threadless Ops II – Enhanced Evasion - avantguard cyber security. [Access source](#)

Call Stack Spoofing - SysWhispers4. [Access source](#)

Ghost Frames — defeats all EDR layers of call-stack-based detection. [Access source](#)

Call Stack Spoofing: x64 Unwinding & Detection - Valhguard. [Access source](#)

GitHub - toneillcodes/UnwindRaven. [Access source](#)

stack-spoofing-dev | Skills Marketplace - LobeHub. [Access source](#)

The Evolution of EDR Bypasses: A Historical Timeline - CovertSwarm. [Access source](#)

Nighthawk 0.2.6 – Three Wise Monkeys. [Access source](#)

Endpoint Detection and Response | zerohuntlabs. [Access source](#)

Process Ghosting | Pentest Laboratories. [Access source](#)

GitHub - thomasxm/BOAZ_beta: Multilayered AV/EDR Evasion. [Access source](#)

Claude-Red/Skills/infrastructure/offensive-edr-evasion/SKILL.md at. [Access source](#)

Process Injection Explained: Techniques, Evasion, and Defense. [Access source](#)

Using process creation properties to catch evasion techniques. [Access source](#)

Process Ghosting Explained - Hackercool Magazine. [Access source](#)

Hiding in plain sight: Developing EDR evasion strategies. [Access source](#)

Herpaderping: Security risk or unintended behavior? - CrowdStrike. [Access source](#)

Container Instrumentation and Enforcement System for Runtime. [Access source](#)

Leveraging eBPF to Produce Container Audit Data for the Purpose. [Access source](#)

KraKenGuard: Towards Fine-Grained eBPF Isolation - USENIX. [Access source](#)

eBPF Ecosystem Progress in 2024–2025: A Technical Deep Dive. [Access source](#)

An Abnormal File Access Detection Model for Containers Based on. [Access source](#)

SindriKit Evasion Framework: Bypassing EDR Telemetry. [Access source](#)

Efficient data flow tracking through hw-assisted parallelization - DTIC. [Access source](#)

Behind the Mask: Spoofing Call Stacks Dynamically with Timers. [Access source](#)

How to Endpoint Detection Evasion - Security Scientist. [Access source](#)

Sylph: An Unsupervised APT Detection System Based on the ... - MDPI. [Access source](#)

Efficient Large-Scale Data Provenance Tracking ... - Thomas Pasquier. [Access source](#)

Evolution of Endpoint Detection and Response (EDR) in Cyber. [Access source](#)

Offensive Tradecraft - Night Walker - Thomas Marques. [Access source](#)

Tidal: Tackling Concept Drift in Provenance-Based Advanced. [Access source](#)

[PDF] Kairos: Practical Intrusion Detection and Investigation using. [Access source](#)

SysWhispers is dead, long live SysWhispers! - klezVirus. [Access source](#)