

**ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ**



**ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS**

Department of Informatics

MSc in Information Systems Development and Security

MSc Thesis:

Evading and crashing anti-malware solutions via data collection
overloading during analysis serialization

Εχμετάλλευση και Κατάρρευση Λύσεων Αντι-Ιομορφικού Λογισμικού
μέσω Υπερφόρτωσης της Συλλογής Δεδομένων κατά τη Σειριοποίηση
της Ανάλυσης

Evgenios Gkritis

f3312306

Thesis Supervisor:

Constantinos Patsakis

Professor at University of Piraeus

Athens, January 2026



Contents

1	Introduction	2
2	Concept and methodology	3
2.1	Threat model and assumptions	5
2.2	Formal Model	5
3	Experimental process and setup	7
3.1	Setup	7
3.2	Goals and evaluation criteria	8
3.3	Test environments and controls	9
4	Results	9
4.1	Failed attacks on EDRs	10
4.2	Wazuh	10
4.3	Undetected malicious behavior from malware sandboxes	11
4.4	Velociraptor	11
4.5	CAPEv2	11
4.6	Triage	12
4.7	Cuckoo	13
5	Discussion	13
5.1	Scalability and Generalization	13
5.2	Root cause analysis and mitigation measures	14
5.3	Limitations	14
6	Related work	15
6.1	Attacks on EDRs	15
6.2	Attacks on malware sandboxes	15
6.3	Denial-of-Service attacks on security solutions	16
6.4	Positioning	16
7	Conclusions	16
A	Ethical Considerations	16
A.1	Stakeholders and potential impacts. . .	16
A.2	Risk mitigation in experimentation. . . .	17
A.3	Use of public/third-party services. . . .	17
A.4	Responsible and coordinated disclosure.	17
B	UI/analysis degradation and process count stress test	17



Abstract

Anti-malware systems rely on sandboxes, hooks, and telemetry pipelines like collection agents, serializers, and database backends, to monitor program and system behavior. We show that these data-handling components constitute an exploitable attack surface that can lead to denial-of-analysis (DoA) states without disabling sensors or requiring elevated privileges. We present Telemetry Complexity Attacks (TCAs), a new class of vulnerabilities that exploit mismatches between unbounded collection mechanisms and bounded processing capabilities. Our method recursively spawns child processes to generate deeply nested and oversized objects that stress serialization and storage boundaries, as well as visualization layers, e.g., JSON/BSON depth and size limits. Depending on the product, this leads to truncated or missing behavioral reports, rejected database inserts, serializer recursion and size errors, and unresponsive dashboards, with some cases also exhibiting normal malicious execution that was not recorded or presented to analysts. We evaluate our technique against 13 commercial and open-source malware analysis platforms and endpoint detection and response (EDR) solutions. Seven products fail at different stages of the telemetry pipeline; three CVE identifiers have been assigned (CVE-2025-61301, CVE-2025-61303, and CVE-2025-67221); one more is pending; and others have issued patches or configuration changes. We discuss root causes and propose mitigation strategies to prevent DoA attacks triggered by adversarial telemetry.

1 Introduction

Security reports estimate that hundreds of thousands of new malicious programs emerge daily, contributing to a corpus exceeding one billion malware samples [6]. The economic impact is similarly stark: IBM reports an average data-breach cost of \$4.4m, with a substantial share attributable to malware-driven incidents. Ransomwhere [14], which catalogs cryptocurrency addresses used by ransomware groups, indicates that total tracked ransomware payments surpass \$1bn. CrowdStrike further attributes a 22% reduction in average breakout time to infostealers [18]. Together, these figures underscore the financial consequences of modern malware, even before accounting for activity by advanced persistent threat (APT) groups and other threat actors.

Defenders often rely on telemetry-centric pipelines to detect malware. Endpoint detection and response (EDR) agents collect behavioral signals (e.g., process, network, file, and registry activity) for detection and investigation, while dynamic-analysis sandboxes detonate artifacts in isolated environments to observe behavior and generate

reports with behavioral indicators [1, 12, 15, 16, 21, 34, 40, 41]. These systems typically combine OS-level event providers (e.g., process creation monitoring via API hooking or ETW/Sysmon instrumentation), user-space collectors, sandbox execution, and backend aggregation. For interoperability and storage, telemetry is commonly serialized into structured logs (often JSON) as it flows through the pipeline; for instance, Wazuh serializes endpoint events as JSON records (e.g., alerts.json) prior to indexing [54].

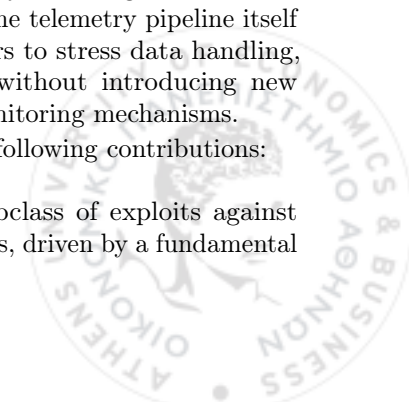
Such pipelines implicitly assume end-to-end resilience: once malicious activity executes in a monitored environment, its corresponding events should appear in the final report. In practice, malware actively undermines this assumption via anti-analysis techniques that detect sandboxed or instrumented settings and then alter, delay, or suppress behavior. Common strategies include checking for VM artifacts, timing and resource probes, and user-interaction tests, and have grown increasingly sophisticated [2]. These techniques primarily aim to evade observation by minimizing telemetry generation, often by making malware appear benign and exploiting defenders' dependence on novelty-based detection (e.g., new signatures or behavior models).

Our work departs from this paradigm. Telemetry pipelines strive to capture rich detail (event logs, process trees, memory snapshots), yet the components that serialize, store, and render this information have practical limits, including maximum JSON nesting depth, document size caps, database insertion constraints, and user interface (UI) rendering capacity. We present a class of attacks that intentionally generate pathological yet valid telemetry to push these components beyond their limits.

We identify a critical design assumption: telemetry is treated as benign input to be processed, rather than as an attack surface. Defensive tools scrutinize program behavior but largely trust the infrastructure that records and transmits that behavior. Unlike traditional evasive malware that hides its actions, our attacks do not conceal malicious behavior; instead, they overload the pipeline's ability to represent it. The result is telemetry that is sufficiently malformed or voluminous that the security product fails to record it correctly or present it to an analyst—making the activity effectively invisible not by avoiding the observer, but by breaking it. We show that an attacker can exploit the telemetry pipeline itself using perfectly legal behaviors to stress data handling, achieving detection bypass without introducing new evasion of the underlying monitoring mechanisms.

Contributions. We make the following contributions:

1. We introduce a new subclass of exploits against malware analysis pipelines, driven by a fundamental



mismatch between unbounded data collection and bounded data processing.

2. We formalize Telemetry Complexity Attacks (TCAs) as a subset of resource-exhaustion attacks with telemetry-specific failure modes. Our formal model enables accurate estimations of when and where the mechanisms will fail.
3. We propose a process-spawning methodology that generates high-volume telemetry, stressing serialization and storage boundaries and inducing analysis failure or denial-of-service in downstream components.
4. We target the post-collection pipeline (serializers, databases, renderers) rather than the collection mechanism itself or the host OS.
5. We achieve a silent visibility loss rather than a denial-of-service attack. The monitored system continues operating normally; only the defender’s view is compromised. While this may sound like a limitation, it allows the attacks using malware that follow this approach to pass under the radar.
6. Due to our formal estimation of where the failure is expected, we may operate within resource bounds that avoid triggering fork-bomb detections, requiring careful calibration of depth and rate parameters.
7. We provide an analysis and a binary proof-of-concept that reaches dynamic analysis engines (sandboxes and EDR telemetry pipelines) while avoiding static classification as malware. Rather than evading instrumentation or disabling hooks, we (i) manipulate monitored data to construct telemetry records that trigger ingestion/storage failures, and (ii) abuse container formats and serialization boundaries to provoke faults across stages (ingestion, transformation, storage), with outcomes from missing fields to pipeline collapse.
8. We evaluate the technique against multiple anti-malware solutions, resulting in vendor patches and the publication of three CVEs, with one more pending.

Our results show that the telemetry pipeline itself is a critical attack surface. To our knowledge, this work is the first to systematically demonstrate, across multiple deployed security products, that adversarial but valid telemetry structures can induce analyst-visible denial-of-analysis outcomes through serialization, storage, and rendering boundaries,

even when the monitoring sensors remain intact. Thus, our contributions include identifying the telemetry pipeline as a cross-product attack surface, empirically characterizing failure modes across diverse architectures, and validating the real-world impact through coordinated disclosures and assigned CVEs.

Responsible disclosure. We coordinated responsible disclosure with affected vendors, providing at least 90 days for them to patch their solutions, leading to patches and configuration hardening across multiple EDR and sandbox products. At the time of writing, three issues have been assigned CVE IDs, namely CVE-2025-61301 (CAPEv2), CVE-2025-61303 (Triage), and CVE-2025-67221 (orjson serialization library), with one additional CVE pending for Cuckoo. Proof-of-concept artifacts for the two CVEs are available on NIST’s website (GitHub repositories can be found there) [37–39].

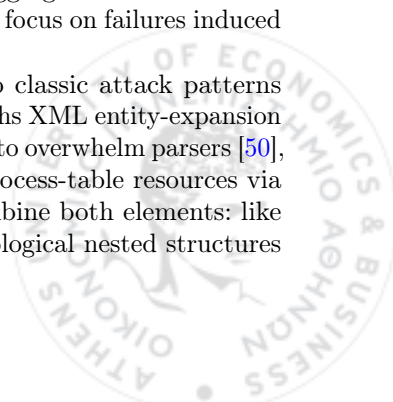
Overall, TCAs offer a new evasion perspective: by leveraging data serialization and pipeline processing limits, attackers can induce silent analysis failures without defeating the monitoring mechanisms themselves.

Structure. The rest of this work is structured as follows. In the next section, we describe our methodology, the concept behind each attack, and a formal model. Section 3 presents our experimental process and how we created our test environments. Next, Section 4 details our experimental results in various open source and proprietary malware monitoring and analysis environments. Then, in Section 5, we first discuss how our results can scale and be generalized. We also perform a root cause analysis of our findings and propose mitigation strategies. Finally, we conclude the article, summarizing our contributions and suggesting ideas for future work.

2 Concept and methodology

Our objective is to stress the telemetry pipeline rather than disable sensors. Many malware-analysis systems adopt greedy collection because they cannot predict which actions a program will take and thus lack clear collection thresholds. While endpoint products may block execution via static analysis, sandboxes are typically expected to record all observable actions of a submitted binary; however, permissive logging can backfire. Since some filtering already exists, we focus on failures induced by data representation.

Our approach draws on two classic attack patterns from literature. The billion laughs XML entity-expansion attack uses recursive structures to overwhelm parsers [50], and the fork bomb exhausts process-table resources via rapid spawning [8]. TCAs combine both elements: like billion laughs, we induce pathological nested structures



that stress serialization and storage; like fork bombs, we use process creation as the generator. The target, however, is different. Billion laughs primarily targets the parser, while fork bombs target the OS scheduler. TCAs target downstream telemetry components (serializers, databases, and dashboards) while the monitored host and the program continue to run. The outcome is typically not a host crash but a silent loss of visibility: malware executes, yet defenders cannot reliably observe what occurred.

Trivial Example. Consider a sandboxed binary whose execution triggers process activity that is timestamped and linked via parent-child relationships, and may also produce artifacts such as memory dumps. This forms a tree, but interoperability and storage often use a nested JSON encoding. Arbitrary nesting, however, is not uniformly supported across libraries and backends.

A sample that recursively spawns processes in a tree is valid behavior: modern operating systems permit thousands of processes, and sensors will log each creation event. The pipeline must then serialize a potentially massive process tree. A JSON encoder may emit a deeply nested object (a parent containing children, each with their own children, etc.), yet many JSON libraries and document stores impose nesting-depth limits (often ~64–128 levels) and document-size caps. Similarly, dashboards may fail to render extremely large trees due to DOM growth or timeout constraints. In our experiments, recursive process spawning readily breached these limits, causing serialization exceptions or truncation and making web-based analysis UIs unresponsive. This mismatch between what sensors can collect and what backends can process yields exploitable failure modes at multiple stages of the pipeline.

We demonstrate, through extensive experiments, that this vulnerability affects multiple real-world anti-malware products and services. Across deployments, we observe: (i) missing or partial behavioral reports, (ii) rejected database inserts due to size or depth limits, (iii) serializer crashes from recursion or memory exhaustion, and (iv) stalled or unresponsive dashboards. These effects require no elevated privileges and do not tamper with hooks, sensors, or agents; they arise solely from adversarially structured telemetry that is valid but pathological.

To verify that the attack does not impede malware functionality, the payload triggers a malicious action at a chosen depth (e.g., launching a shell). This enables systematic stress testing of pipeline components across commercial and open-source sandboxes and EDR environments. In all cases, the attack operates in user mode without code injection or interference with kernel hooks or sensor processes. Under our crafted telemetry load, we observe dropped or incomplete logs, insertion

errors, serialization failures, and dashboard instability; critically, the malicious action is often absent from reports, so products fail to detect or alert. This form of evasion can be more damaging than a crash because it can convince defenders that no threat is present.

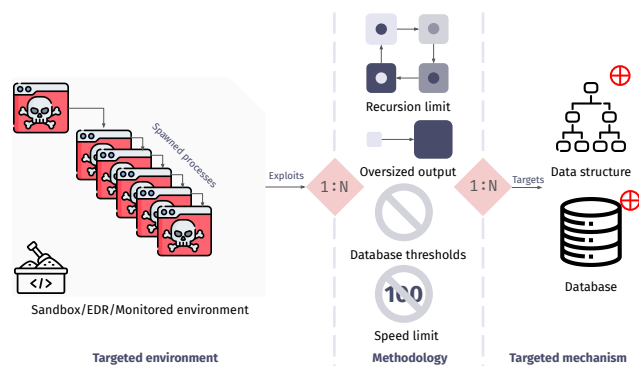


Figure 1: High-level overview of attack methodology

Our attack uses controlled recursion using a crafted binary that repeatedly launches an embedded copy of itself, increments a depth counter each iteration, and, at a predetermined maximum depth, executes a lightweight payload (in our tests, a PowerShell reverse shell) to confirm end-to-end execution. Depth is carried across processes via environment variables. The implementation evolved from a minimal C/C++ proof-of-concept into a hardened test binary with multiple stages of XOR-based encryption to reduce trivial static detection and ensure the relevant behavior manifests only at runtime.

Figure 1 summarizes the method’s workflow and the primary failure modes exercised. On the left are the targeted environments (sandboxes and endpoint agents) in which a proof-of-concept binary was executed. The central column represents the operational limits we triggered in our experiments: recursion limit refers to code-level or timeout protections that abort deep traversal; oversized output captures large JSON or BSON blobs that exceed serializer or database limits; database thresholds indicate document size and nesting caps or insert limits that cause errors; and speed limit symbolizes UI rate caps that produce agent queue overflows or unresponsive dashboards. The panel to the right shows the telemetry mechanisms we targeted (process and call-argument data structures and persistent storage), where failures manifest as truncated or missing process trees, corrupted or absent behavioral reports, and UI stalls. Together, the elements illustrate how a simple recursive load can propagate through the telemetry pipeline and produce denial-of-analysis conditions.

2.1 Threat model and assumptions

We detail our threat model and assumptions below. Thus, we list all the parameters that are relevant for our attack, what is controlled and by whom, but also what we expect to achieve. Execution context. We assume that an adversary can store an unprivileged user-mode binary on either (i) an endpoint that is instrumented by sandboxing or EDR/DFIR tooling, or (ii) at a cloud solution, by uploading the binary through common user interfaces. The binary recursively spawns child processes up to a chosen depth and rate. Execution occurs under typical analysis conditions (e.g., detonation VMs, lab hosts, or enterprise-managed endpoints) where process creation is permitted, and environment variables propagate to child processes.

Telemetry. We assume that monitoring stacks capture process execution (e.g., via user-mode hooking and/or kernel/event providers), serialize these events into structured records (e.g., JSON/BSON or comparable encodings), and store them to a database-backed backend that exposes dashboards or APIs to analysts. Default configurations of such tools (e.g., document size caps, recursion/nesting limits, per-index write limits, dashboard rate caps, and analysis timeouts) are used as-is and not tuned to any specific workload. Also, the adversary cannot modify or disable sensors, hooks, or ETW/Sysmon providers; load kernel drivers; or gain elevated privileges, directly access or reconfigure serialization libraries, databases, or dashboards.

We assume the telemetry pipeline ingests data derived from untrusted program behavior (e.g., deep process trees, long command-line arguments, large argument arrays, and nested object graphs), and that serialization and storage components treat these data as valid inputs rather than as isolation boundaries.

Resource bounds. We assume that recursive spawners use a maximum depth chosen to avoid gross host-level denial of service (e.g., kernel scheduler collapse). We do not rely on exhausting OS resources (e.g., CPU, RAM, PIDs) as the primary failure mechanism. This was proven true in all tested cases.

Network and isolation. According to our evaluation methodology, execution can occur in offline or network-restricted environments. A network is not required for the core effect of the presented methodology.

Non-assumptions. The attack does not assume administrative privileges, kernel drivers, agent tampering, or vendor-specific vulnerabilities. It does not require disabling sensors, unhooking instrumentation, or modifying backend configurations.

Non-goals. This work does not aim to achieve persistence, privilege escalation, data exfiltration, or evasion of dynamic behavior capture. We do not

claim cryptographic bypasses, supply-chain compromise, or exploitation of memory corruption. Likewise, we did not introduce obfuscation mechanisms that would completely blind the payload, but only the necessary ones to prevent EDRs from blocking execution from static analysis. As such, we focus only on introducing operational failures in telemetry pipelines by using valid but manipulated inputs generated from the data sources they are programmed to collect.

Scope note (EDR vs sandbox/DFIR). Our strongest results apply to platforms whose goal is to retain and present high-fidelity behavioral telemetry for analyst review, such as malware sandboxes and DFIR pipelines. Many endpoint protection systems mitigate this class of attack by terminating fork-bomb-like behavior early; in those cases, the attack fails because execution is stopped, not because telemetry handling is robust. We therefore report EDR outcomes separately and interpret them as a boundary on applicability, rather than a negative result about telemetry pipelines.

2.2 Formal Model

Definition 1 (Telemetry pipeline). We define a telemetry pipeline as a tuple $\Pi = (\mathcal{A}, \mathcal{E}, \mathcal{S}, \mathcal{R})$ where the agent/collector \mathcal{A} observes system events $e \in \Sigma$ and produces a stream $X = (e_1, e_2, \dots)$; the encoder \mathcal{E} serializes batches $B \subseteq X$ into objects $o = \mathcal{E}(B)$ belonging to an object space O (e.g., JSON or BSON documents); the store \mathcal{S} persists objects subject to capacity constraints; and the renderer/API \mathcal{R} exposes persisted telemetry to analysts.

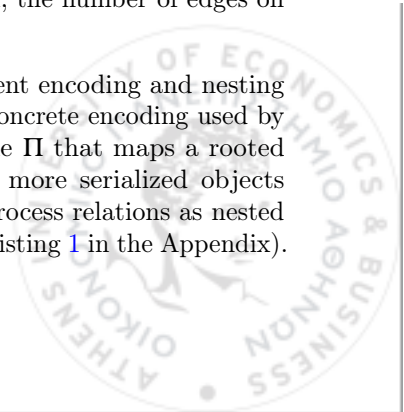
Definition 2 (Telemetry graph and depth). Execution induces a directed telemetry graph $G = (V, E, D)$ where vertices $v \in V$ are entities (e.g., processes, threads, sockets), edges E represent relations (e.g., spawned, communicated-with), and D stores per-vertex/edge metadata (arguments, timestamps, command line).

For any directed graph G , we define its (reachable) depth from a distinguished root vertex $r \in V$ as

$$\text{depth}(G, r) = \max_{v \in V \text{ reachable from } r} (\text{dist}(r, v)),$$

where $\text{dist}(r, v)$ is the length (number of edges) of a shortest directed path from r to v . For a rooted tree, this coincides with the usual notion, the number of edges on the longest root-to-leaf path.

Definition 3 (Graph-to-document encoding and nesting depth). Let Enc_Π denote the concrete encoding used by a platform's telemetry pipeline Π that maps a rooted telemetry graph G to one or more serialized objects $o \in O$. Many systems encode process relations as nested objects or nested arrays (e.g., Listing 1 in the Appendix).



We define the encoding nesting depth $d_{\text{enc}}(G)$ as the maximum bracket, object, or array nesting depth in the serialized representation produced by $\text{Enc}_{\Pi}(G)$.

Note that $d_{\text{enc}}(G)$ can equal the graph depth $d(G)$ for tree encodings, but can also differ if the platform flattens edges into tables, performs chunking, or uses multiple documents. In our evaluation, we attribute nesting-limit failures to violations of bounds on $d_{\text{enc}}(G)$, not necessarily on $d(G)$.

Definition 4 (Admission predicates). Each stage $Y \in \{\mathcal{E}, \mathcal{S}, \mathcal{R}\}$ implements a Boolean admission predicate

$$\text{admit}_Y : O \times C_Y \rightarrow \{\text{true}, \text{false}\}$$

which returns **true** iff object $o \in O$ can be accepted under capacity constraints C_Y .

Definition 5 (Capacity model). Each pipeline stage $Y \in \{\mathcal{E}, \mathcal{S}, \mathcal{R}\}$ has a capacity vector

- Encoder limits: $C_{\mathcal{E}} = (d_{\text{max}}^{\text{nest}}, s_{\text{max}}^{\text{obj}}, r_{\text{max}}^{\text{stack}})$ for maximum nesting depth, per-object size, and recursion/stack depth.
- Store limits: $C_{\mathcal{S}} = (s_{\text{max}}^{\text{doc}}, d_{\text{max}}^{\text{store}}, q_{\text{max}})$ for document size, nesting depth (e.g., BSON), and insert quota.
- Renderer/UI limits: $C_{\mathcal{R}} = (m_{\text{max}}^{\text{mem}}, n_{\text{max}}^{\text{DOM}}, e_{\text{max}}^{\text{msg}}, q_{\text{max}}^{\text{buf}}, d_{\text{max}}^{\text{UI}}, \mu_{\mathcal{R}})$, capturing (in order) memory, UI element count, maximum message size, event-queue buffer capacity, maximum renderable tree depth under the UI's DOM/memory constraints, and service rate.

A stage accepts an object o (e.g., an encoded event, stored document, or rendered UI node) iff $\text{admit}_Y(o, C_Y) = \text{true}$; otherwise it truncates or rejects the input.

Definition 6 (Rendered view). Let Render_{Π} denote the platform's renderer that maps stored/decoded telemetry into the analyst-visible UI view. We define $\text{Rendered}_{\Pi}(G) \subseteq V$ as the subset of vertices whose associated telemetry is ultimately presented to the analyst after encoding, storage, and rendering (including any truncation or drops).

Definition 7 (Stage-aware complexity). Let $G = (V, E, D)$ be a telemetry graph rooted at r . Moreover, let (i) $d(G) = \text{depth}(G, r)$, (ii) $S(G) =$ serialized size of the encoded object(s), (iii) $n(G) = |\text{Rendered}_{\Pi}(G)|$, the number of analyst-visible entities (and, when rendered as a tree, a proxy for UI node count), and (iv) $\lambda(G)$ represents the rate at which telemetry events are

generated, controlled by the adversary's spawn rate. We collect these into a stage-aware complexity vector

$$\vec{C}(G) = (d(G), S(G), n(G), \lambda(G)).$$

Each pipeline stage $Y \in \{\mathcal{E}, \mathcal{S}, \mathcal{R}\}$ induces thresholds on the relevant components of $\vec{C}(G)$ via its capacity vector C_Y . The first three components (d, S, n) govern static structural limits at the encoder, store, and renderer, respectively, while λ governs dynamic throughput limits and is used to model queue-based failures (see Theorem 3)

Definition 8 (Visibility). Let $\mathcal{M} \subseteq V$ denote nodes representing malicious activity. Moreover, let $\text{Rendered}_{\Pi}(G) \subseteq V$ denote the subset of vertices whose associated telemetry successfully passes through all pipeline stages and is presented to the analyst. We define the visibility of the pipeline as

$$\text{Vis}_{\Pi}(G) = \frac{|\mathcal{M} \cap \text{Rendered}_{\Pi}(G)|}{|\mathcal{M}|} \in [0, 1],$$

Definition 9 (Denial-of-analysis (DoA)). Given a policy threshold $\tau \in (0, 1]$, a telemetry complexity attack (TCA) succeeds when $\text{Vis}_{\Pi}(G) < \tau$, while the malicious computation still executes normally.

Adversary model The adversary \mathcal{Adv} controls the shape of the telemetry graph G via their program behavior, without elevated privileges or tampering with sensors. Thus, the adversary selects parameters: $\text{spawn}(b, d, k)$ where b is branching factor, d is depth, and k controls metadata size (e.g., command-line argument length in bytes).

Definition 10 (Serialization overhead). Let $\alpha > 0$ denote the structural serialization overhead, that is, the fixed number of bytes added per node by the encoding format, independent of node metadata. This includes field delimiters, keys, brackets, and type markers. For JSON encoding of a process node, typical values are $\alpha \in [50, 200]$ bytes depending on the schema.

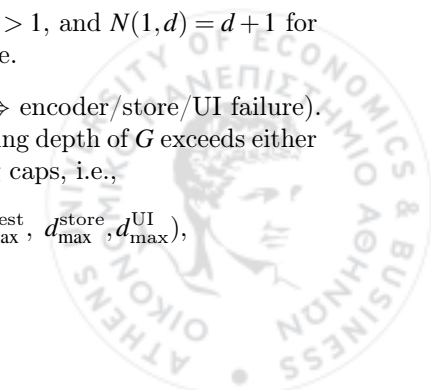
The encoded object for a full b -ary tree of depth d with per-node metadata of size k has approximate serialized size

$$S(b, d, k) \approx N(b, d) \cdot (\alpha + k),$$

where $N(b, d) = \frac{b^{d+1} - 1}{b - 1}$ for $b > 1$, and $N(1, d) = d + 1$ for the unary (linear chain) case.

Theorem 1 (Depth bound \Rightarrow encoder/store/UI failure). If the effective encoding nesting depth of G exceeds either the encoder or store nesting caps, i.e.,

$$d_{\text{enc}}(G) > \min(d_{\text{max}}^{\text{nest}}, d_{\text{max}}^{\text{store}}, d_{\text{max}}^{\text{UI}}),$$



then for objects o encoding the corresponding subtree,

$\text{admit}_{\mathcal{E}}(o, C_{\mathcal{E}}) = \text{false}$ or $\text{admit}_{\mathcal{S}}(o, C_{\mathcal{S}}) = \text{false}$, or

$\text{admit}_{\mathcal{R}}(o, C_{\mathcal{R}}) = \text{false}$

and telemetry associated with the pruned subtree becomes invisible.

Note that d_{\max}^{UI} depends on browser memory limits and DOM complexity.

Theorem 2 (Size bound \Rightarrow store failure). Let $M = s_{\max}^{\text{doc}}$ be the per-document limit. If $S(b, d, k) > M$, the store rejects the document. Let d^* denote the minimum depth guaranteed to trigger a size-bound failure. Then,

$$d^* \geq \left\lceil \log_b \left(1 + (b-1) \frac{M}{\alpha + k} \right) \right\rceil - 1.$$

Theorem 3 (Burst/queue bound \Rightarrow renderer/API failure). Let $B = B_{\mathcal{R}}$ and $\mu = \mu_{\mathcal{R}}$ be the queue buffer capacity and service rate from the renderer capacity vector $C_{\mathcal{R}}$. We model the renderer as an $M/M/1/B$ queue with event arrival rate $\lambda = \lambda(G)$. If $\lambda > \mu$, buffer overflow occurs in expected time

$$T_{\text{overflow}} \approx \frac{B}{\lambda - \mu}.$$

Additionally, if the renderer must materialize $N(b, d)$ DOM nodes and $N(b, d) > n_{\max}^{\text{DOM}}$, or if the transport message size exceeds e_{\max}^{msg} , the UI or API fails immediately.

Definition 11 (Successful TCA). Using the stage-aware complexity vector

$$\vec{C}(G) = (d(G), S(G), n(G), \lambda(G)),$$

a TCA succeeds iff at least one pipeline constraint is violated and visibility drops:

$$\exists i : \vec{C}(G)_i > \vec{C}_{\max, i} \text{ and } \text{Vis}_{\Pi}(G) < \tau.$$

Therefore, we target implicit constraints of the underlying implementations for each telemetry component. More precisely: First, we target serialization as implementations impose recursion, stack, and object size limits that can be triggered by nested or high-volume telemetry structures. Then, we target storage since backends impose document, request, or record limits that may reject inserts or truncate stored telemetry. For instance, when a platform uses MongoDB-like document stores, per-document size and nesting constraints can produce missing subtrees or empty reports. Finally, we target the rendering engines. Web UIs and APIs can fail due to oversized responses or when asked to materialize large hierarchical datasets; practical failures often manifest as timeouts, unresponsive dashboards, or incomplete report rendering.

Based on the above, we can assume a successful attack on a MongoDB-dependent solution as follows.

Table 1: Estimated capacity parameters by platform. Notation: [†]Python default recursion limit. [‡]Size-bound failure predicted; UI failure occurs first. [§]Buffer limit, not serialization depth.

Platform	d_{\max}^{nest}	s_{\max}^{doc}	α (bytes)	d_{pred}^*	d_{obs}^*
CAPEv2	100	16 MB	180	95	100
Cuckoo	1000 [†]	50 MB	210	238k [‡]	180
Velociraptor	-	4 MB	150	26.7k	200 [§]

Corollary 1 (MongoDB-like store limits). Consider a deployment where $s_{\max}^{\text{doc}} = 16\text{MB}$ and $d_{\max}^{\text{store}} = 100$ (following MongoDB’s limits [35]). Let α be the structural serialization overhead. A sufficient condition to violate at least one store constraint is:

$$d > 100 \text{ or } d \geq \left\lceil \log_b \left(1 + \frac{(b-1) \cdot 16 \times 10^6}{\alpha + k} \right) \right\rceil - 1.$$

Therefore, forcing a solution to exceed any of these thresholds individually, we expect an analysis failure (e.g., truncation, rejection, timeouts, UI hangs, or crashes).

Using the above formalisation, we can interpret Figure 2 as a visual representation of Definition 11. The recursive spawner controls the complexity vector $\vec{C}(G)$, and each failure mode corresponds to a violated capacity constraint in the encoder, store, or renderer. Moreover, our formal model enables us to estimate capacity parameters through documentation review and incremental probing.

Indeed, using Theorem 1 and Theorem 2, we predict failure thresholds:

For CAPEv2 with MongoDB ($d_{\max}^{\text{store}} = 100$):

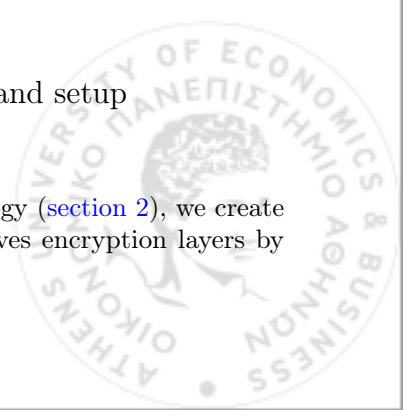
$$d_{\text{depth}}^* = d_{\max}^{\text{store}} = 100 \quad (1)$$

For Cuckoo, the formal model predicts size-bound failure (see Section 4) at $d \approx 238,000$ (assuming $s_{\max}^{\text{doc}} = 50\text{MB}$, $\alpha = 210$). However, the observed failure occurs at $d = 180$ because renderer/UI limits dominate in practice, the UI cannot materialize the resulting view once $N(b, d) > n_{\max}^{\text{DOM}}$. This highlights that platform-specific renderer constraints can trigger analysis failure well before store-size bounds. Nevertheless, the model provides useful bounds for assessing vulnerability exposure.

3 Experimental process and setup

3.1 Setup

As described in our methodology (section 2), we create a binary that gradually removes encryption layers by



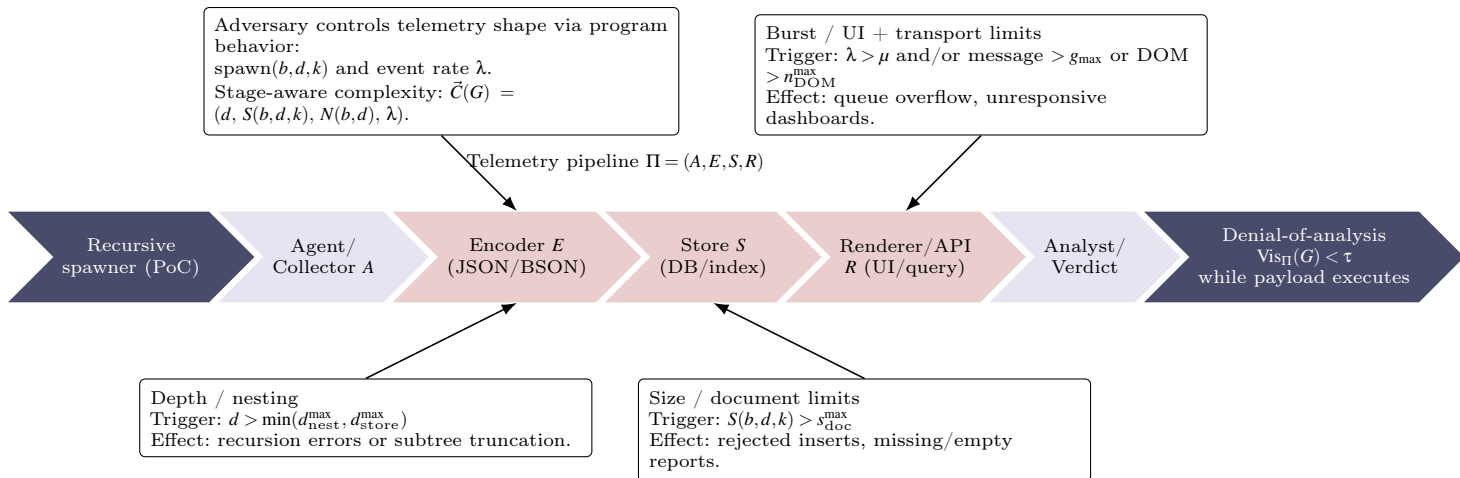


Figure 2: A recursive process spawner induces a telemetry graph whose depth, size, or %event rate exceeds implicit limits in the encoder, store, or renderer, yielding missing or unusable analyst-visible telemetry.

spawning a decrypted version of itself. As such, in each execution, the binary keeps a counter of the spawned processes to determine when to stop. At that step, our binary would open a reverse shell to a host under our control. Thus, we intended to create a structure (similar to Listing 1 in the Appendix), where deep nesting would create storage or processing issues. Minor changes to the binaries were introduced to explore the potential problems as we evaluated each software solution.

The experiments followed a progressive strategy. We started with open-source sandboxing platforms to iterate quickly and observe internal failures. After validating the concept on those systems, we moved to open-source EDR and DFIR tools and locally deployed trial instances of commercial products. Finally, sanitized samples were submitted to public, cloud-based analysis services to validate behavior with online providers. All local testing was performed in isolated, offline laboratory networks using virtual machines to prevent telemetry leakage and to ensure safe handling of test artifacts. Each sandbox or EDR instance was installed with default or documented configurations; where supported, we varied timeouts, process depth limits, memory dump options, and report compression settings to exercise different code paths. Tests were repeated multiple times to ensure reproducibility; observed failures were cross-validated across independent environments when possible. Logging and monitoring of host resource usage (CPU, memory, disk I/O) were performed for each run to correlate system load with observed telemetry failures. The isolation of the experimental environment from the internet was introduced to prevent telemetry from communicating with software vendors, who could then prevent the

execution of our binaries by adding static checks or patching their solutions on the fly.

3.2 Goals and evaluation criteria

The primary objective of the experimental phase was to validate whether recursive process spawning could cause denial-of-analysis or otherwise impair the behavioral visibility of sandbox and endpoint detection systems. We designed the experiments to observe three main outcomes: (i) denial or corruption of analysis reports, (ii) partial or total loss of behavioral telemetry, and (iii) degradation of analysis performance or stability leading to timeouts or crashes.

Each local test environment, whether sandbox, EDR, or DFIR platform, was evaluated using identical input parameters, including recursion depth and number of spawned processes. The consistency of execution ensured that the differences in outcomes reflected genuine architectural or implementation differences between the platforms rather than sample variance. The evaluation was carried out on five main pillars: (i) Behavioral completeness: The degree to which each platform successfully captured and recorded all executed behaviors (process creation, PowerShell invocation, network connections). Missing or truncated telemetry indicated a partial failure. (ii) Detection verdict: The final classification or confidence score assigned to the sample (e.g., clean, suspicious, malicious). A low confidence score despite active payload execution was considered an analysis failure. (iii) System stability: Observable signs of engine instability, including crashes, unresponsive interfaces, incomplete reports, or unprocessed data queues. (iv) Report integrity: Verification that the

JSON, BSON, or database-backed output was correctly generated and rendered without corruption, truncation, or serialization errors. (v) Performance overhead: Comparative measurement of processing time, report generation delay, and resource utilization under recursive load compared to baseline runs with benign samples.

We considered a test successful if the sample prevented the behavioral engine from generating a complete or accurate report or caused instability in the telemetry or storage pipeline. Conversely, a platform was considered resilient if it fully recorded the recursive activity and correctly labeled the behavior as malicious without disruption.

3.3 Test environments and controls

Our methodology included testing across a variety of platforms to exercise different telemetry and persistence architectures:

1. Open-source sandboxes. Cuckoo and CAPEv2 were used to evaluate behavior logging and process-tree handling. These platforms rely heavily on JSON-based reporting and were instrumental in identifying truncation, serialization, and database size limitations.
2. Open source EDRs and monitoring tools. Wazuh was used to analyse agent-based monitoring under high-volume recursive process creation (truncated alerts, Sysmon event loss, memory pressure). Velociraptor evaluated forensic and live-response visibility under recursive load.
3. Commercial trial and local deployments. Tools such as Bitdefender GravityZone, Kaspersky EDR, WatchGuard EDPR, and Comodo Valkyrie were deployed in offline virtual machines to assess detection resilience without external telemetry leakage.
4. Online sandboxes. After offline validation, sanitized samples were submitted to public services including VirusTotal, AnyRun, Hybrid-Analysis, Malwation Threat Zone, Recorded Future Triage, OPSWAT Filescan.io, and Comodo Valkyrie to verify behavior across cloud providers.

All test systems were instrumented to capture system events, agent logs, exported reports, and backend errors. Where available, raw JSON/BSON dumps, database error logs, and UI traces were retained for post-mortem analysis. Proof-of-concept artifacts and any reproduction materials were handled in accordance with institutional disclosure policies and stored on isolated lab infrastructure.

To reduce the likelihood of early static detection from EDRs, the sample employs a multi-layer “onion” encryption scheme. Each runtime stage decrypts only the subsequent stage. This prevents the full payload or final command-and-control instruction from being visible during static inspection. The obfuscation is intentionally lightweight, sufficient to bypass basic static heuristics while maintaining reproducibility across testbeds.

For the experiments, we used different samples, all based on the same logic. The samples had variants based on the following:

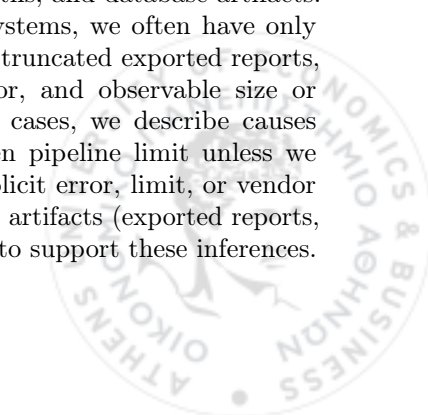
- Payload. We used three different payloads. The simplest payload simply dumped the EICAR test file to trigger the anti-malware solutions. The second one created a reverse shell, using a default and widely used payload, known to trigger most anti-malware solutions. Finally, we had one payload with a keylogger.
- Build architecture. We compiled the code for both x86 and x64 architectures.
- Depth. Since each sample contains an encrypted version of another one in an onion-like approach, to experiment with different depths, we created various samples. The depths that were tested were 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, and 15000.

Thus, ignoring the various testing malware that we developed to assess the efficacy of our approach, 66 samples were tested with each anti-malware solution. So, the total experiments reported in Section 4 are 792.

4 Results

Due to variability in results across products, we detail the experimental results in the following paragraphs, categorizing them by attack success and product, and discussing the root causes for each product where appropriate. Table 2 summarizes the results and the severity of the failures observed.

For open-source platforms, we confirm failure modes using internal logs, code paths, and database artifacts. For proprietary or cloud systems, we often have only black-box evidence, such as truncated exported reports, API responses, UI behavior, and observable size or depth thresholds. In those cases, we describe causes as “consistent with” a given pipeline limit unless we can directly observe an explicit error, limit, or vendor confirmation. We retain raw artifacts (exported reports, error codes, and UI traces) to support these inferences.



Environment	Availability	Type	Attack	CVE
Bitdefender GravityZone EDR	Proprietary	EDR	✗	
Bitdefender GravityZone Sandbox	Proprietary	Malware sandbox	✓	
Malwation Threat Zone Sandbox	Proprietary	Malware sandbox	✓	
CAPEv2	Open source	Malware sandbox	✓	CVE-2025-61301
Cuckoo	Open source	Malware sandbox	✓	Pending
Kaspersky EDR	Proprietary	EDR	✗	
Microsoft Defender	Proprietary	EDR	✗	
Recorded Future Triage	Proprietary/Cloud	Malware sandbox	✓	CVE-2025-61303
Velociraptor	Open source	Digital forensic & incident response	✓	
VirusTotal	Proprietary/Cloud	Multiscan aggregator	✓	
WatchGuard EDPR	Proprietary	EDPR	✗	
Wazuh	Open source	EDR	✓	
Hybrid-analysis	Proprietary/Cloud	Malware sandbox	✗	
orjson (library)	Open source	Serialization library	✓	CVE-2025-67221

Table 2: Overview of our results. Notation: ✓= achieved DoA/visibility loss while payload executed; ✗= blocked/fully logged

4.1 Failed attacks on EDRs

Most EDRs we tested terminated the corresponding processes relatively early, whereas others completed the decryption and terminated the process once a connection to the attacking machine was established. Based on feedback from some vendors, they implemented this measure because their products detected an excessive number of spawned processes within a short time; therefore, they were flagged as fork bombs, and the corresponding processes were terminated. Clearly, EDRs do not have to record all interactions of a binary and wait for a final verdict. Once a binary triggers specific alerts, the EDR immediately kills the process, so our attack did not manage to succeed in most EDR solutions

4.2 Wazuh

A special case of EDRs is Wazuh. Wazuh’s pipeline is heavily dependent on JSON: Agents serialize endpoint events into JSON artifacts (e.g., ossec-alerts.json) and forward them for indexing and correlation. In our tests, adversarially structured telemetry caused those JSON streams and alert files to balloon (we observed ossec-alerts.json and related logs exceeding 100-200 MB), flooding the reporting and correlation pipeline. The result was not an obvious crash, but a silent loss of visibility; critical malicious activity was not logged or surfaced; many alerts were delayed or dropped, and numerous Windows Error Reporting files indicated local instability (stack-overflow exceptions) on the monitored hosts.

From an analyst’s point of view, this is a denial-of-analysis. Wazuh continued to run but failed to produce useful, actionable telemetry, creating dangerous

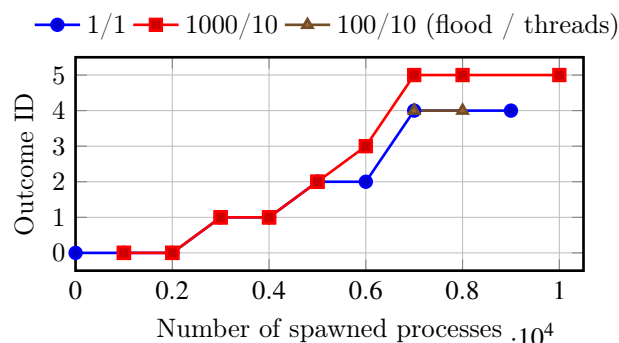


Figure 3: Agent-side stress outcomes versus process count under different flooding intensities.

- (0) Flagged
- (1) Flagged + Event queue flooded
- (2) NOT Flagged + Event queue flooded
- (3) NOT Flagged + Wazuh Error alert
- (4) NOT Flagged + Not enough memory
- (5) same as (4) with anomaly marker in raw logs (* or /*)

blind spots and false confidence. Therefore, the finding suggests a systemic risk for JSON-based EDR solutions, which can be exploited to hide part of their actions.

As seen in Fig. 3, early runs are flagged despite successful shell execution; beyond ~5,000 processes, shell execution persists while detections disappear and the pipeline transitions to event-queue flooding and memory/resource errors. This experiment captures a qualitative phase change in defender visibility. At low to moderate loads (up to 2,000 processes), the shell executes and the activity is flagged. As load increases, the agent/pipeline degrades; around 3,000–6,000 processes we observe explicit event-queue

flooding and, importantly, loss of flagging even though shell execution continues. Past ~7,000 processes, the dominant failure mode becomes resource exhaustion (“not enough memory resources”), with additional anomalous cases (marked in the raw notes) under high-flood configurations, consistent with TCAs shifting detection from semantic analysis to capacity limits in telemetry handling.

4.3 Undetected malicious behavior from malware sandboxes

In the case of some malware sandboxes, e.g., Bitdefender GravityZone Sandbox and Threat Zone, even when there is a verdict, the solution does not detect what the malware was doing or report it to the analyst. In fact, in some cases, the solution might not have broken, e.g., Threat Zone, but the verdict was that the sample was benign.

4.4 Velociraptor

Velociraptor presented a noticeably different failure mode compared to the malware sandboxes. The problem we observed was not a silent database rejection or a library exception, but rather an operational breakdown in Velociraptor’s collection and rendering pipeline when asked to materialize very large process trees. Two related reproducible issues were identified during our testing.

First, the `Generic.System.Pstree` artifact attempts to return the entire process tree as a single JSON row. Under our recursive-load experiments, this row becomes extremely large and, during transmission or queuing, exhausts the client’s memory buffers. The client process is then terminated by the local OS, producing the following observable behavior: the flow aborts with the message:

```
Flow not known - maybe the client crashed?
```

and the result folder for the collection contains only a tiny (4 KB) compressed artifact, indicating severe truncation of the expected output.

Velociraptor’s engineering team confirmed the root cause; the artifact encoded the entire tree in a single JSON row. A patch was merged (PR # 4454, Sep 17, 2025) [53] with the commit message:

```
Bugfix: Limit number of entries emitted into
process_tracker_tree(). The Generic.System.Pstree
artifact puts the entire tree into a single JSON row
which can exceed buffer size in some cases. This causes
problems in deeply nested trees.
```

Second, even when the backend successfully receives the raw event stream, for example, when `Windows.Events.TrackProcesses` is configured to forward updates, the Velociraptor UI, and gRPC transport hit practical limits. In our tests, the raw sysmon-derived events did contain all spawned processes, but attempting to display or transmit that dataset produced the following gRPC error:

```
received message larger than max (8732115 vs.
4194304)
```

causing the interactive view to fail and the analyst to lose visibility. Therefore, although the events were captured, it was practically invisible to the operator, who had to perform low-level interactions to collect them.

4.5 CAPEv2

For CAPEv2, we used the commit 52e4b43 from 2025-05-17. CAPEv2 uses MongoDB for storage and relies heavily on JSON/BSON serialization in Python. The above introduces various constraints. For instance, using MongoDB implies two distinct quotas. First, when storing analysis reports in MongoDB, there is a 16 MB BSON document size limit. Similarly, MongoDB enforces a maximum nesting depth of 100 levels. Therefore, if reports exceed the 16 MB quota or the nesting exceeds 100, the database rejects the insert, causing report generation to fail, regardless of whether the corresponding collector worked correctly.

Setting aside the underlying database, CAPEv2 also relies on Python’s `orjson` library to serialize results into JSON. However, the latter also has its own limits regarding deeply nested or recursive structures. Therefore, this is another potential vulnerability condition, as properly crafted samples may cause the serializer to exceed Python’s recursion limit, leading to serialization failures and aborted report generation. Note that through our investigation of CAPEv2, we identified a related vulnerability in `orjson` itself: the `orjson.dumps()` function does not enforce recursion limits when serializing deeply nested data structures, leading to stack exhaustion and crashes. This issue affects `orjson` versions through 3.11.4 and has been assigned CVE-2025-67221 [39]. Since `orjson` is widely used in Python applications for high-performance JSON serialization, this vulnerability extends beyond CAPEv2 to any application that serializes untrusted nested data using `orjson`.

We believe that some of these limitations were known to the developers of CAPEv2, as there are integrated mechanisms to mitigate them, e.g., by progressively pruning large parts of the report by removing child nodes

Outcome class	Timeout range	Observed trigger / notes
Behavior OK	60-120s (most runs)	Reporting succeeds; report grows with logged processes
No behavior	≥180s (some runs)	Store insertion fails (MongoDB Code 15) despite execution
Failed reporting	200s, 300s (with heavy settings)	Encoder recursion limit + MongoDB Code 15; report missing

Table 3: CAPEv2: failure thresholds at a glance.

or entire process subtrees. However, even this pruning is insufficient when behavior complexity surpasses thresholds, resulting in repeated failures to save the report.

Our tests resulted in a High-Severity (CVSS 7.5) denial-of-analysis vulnerability affecting reporting/mongodb.py and reporting/jsondump.py in CAPEv2 that allows attackers who can submit samples to cause incomplete or missing behavioral analysis reports by generating deeply nested or oversized behavior data that trigger MongoDB BSON limits or orjson recursion errors when the sample executes in the sandbox. Note that many systems depend on CAPEv2 dynamic analysis, e.g., VirusTotal uses it as one of its sandboxes. This issue has been assigned CVE-2025-61301 [37].

From a system perspective, system logs show MongoDB OperationFailure Code 15 and “Recursion limit reached” errors during JSON serialization. Moreover, final reports often contain no behavioral data or show “failed reporting”, falsely making the sample appear benign or inactive. Table 4 shows evidence collected across multiple analysis runs (report sizes, logged processes, and errors). Table 3 shows that as telemetry volume increases, runs transition from normal reporting to loss of analyst-visible behavior (“No behavior”) and, ultimately, to reporting failure (missing/empty report) under recursion and storage constraints.

Table 4 shows that “No behavior” and “Failed reporting” correlate with MongoDB insert failures (Code 15), and with recursion-limit errors under heavier logging profiles.

Report size is a practical proxy for telemetry volume. For CAPEv2, failures cluster at higher volumes and longer timeouts above 180s (see Figure 4); heavy profiles can fail reporting even when a timeout is fixed.

4.6 Triage

Similar to our findings in CAPEv2, a critical (CVSS 9.8) denial-of-analysis vulnerability was found in Triage [45], a widely used commercial malware sandbox. Given that the platform is proprietary and the vendor did not disclose further details, we provide our own speculations. Nevertheless, this issue has been assigned

T (s)	Prof	Dur (s)	Proc	BSON (files/KB)	Report (KB)	Result
60	L	413	24	25/126	613	OK
60	B	252	38	39/200	923	OK
60	H	230	42	43/230	1054	OK
100	H	330	61	62/318	1482	OK
100	B	277	81	82/493	2032	OK
120	H	340	72	73/383	1793	OK
120	B	-	88	89/475	2228	OK
180	H	414	90	91/492	2356	No beh. (MongoDB 15)
180	B	323	88	89/475	2228	OK
200	B	413	119	120/645	3174	No beh. (MongoDB 15)
200	M	-	-	144/726	-	Fail rpt. (Rec.+Mongo)
300	B	513	146	147/779	3676	No beh. (MongoDB 15)
300	H	-	-	167/892	-	Fail rpt. (Rec.+Mongo)

Profiles (Prof): B=baseline (no custom settings); L=enforce-timeout; M=full-proc-memory-dumps+import-reconstruction+enforce-timeout; H=M + syscall.
 Abbrev: OK=Behavior OK; No beh.=report indicates no behavior; Fail rpt.=failed reporting; Rec.=recursion limit reached.

Table 4: CAPEv2: compact run-level evidence.

CVE-2025-61303 [38].

Using the same concept as in CAPEv2, we submitted a sample to Triage for assessment. The sample generates a very high process volume and deeply nested activity. Using, for instance, Windows 10 v2004 and Windows 10 LTSC 2021 images to test our sample, a denial-of-analysis condition is performed, allowing a submitted sample to exhaust process-tracking and logging resources by recursively spawning large numbers of child processes. The result is truncated or missing telemetry; the PowerShell execution and reverse-shell stages that execute on the host are not recorded or reported. In fact, the initial behavioral output reported a score of 1/10, which marks it actually as benign, despite the sample executing PowerShell reverse-shell stages and other malicious actions on the host.

Reproducible public detonations are available on Triage [46], which we used to validate the behavior across the provider’s Windows images. Based on the Triage API, we know that the behavior analysis report is a JSON, which, for the processes, uses a nested format. Therefore, we assume that the same process is followed when collecting the events, leading to a crash that truncates the output. We do not consider the issue to be in the presentation layer, as a score (1/10) is fetched and part of the process tree is reported.

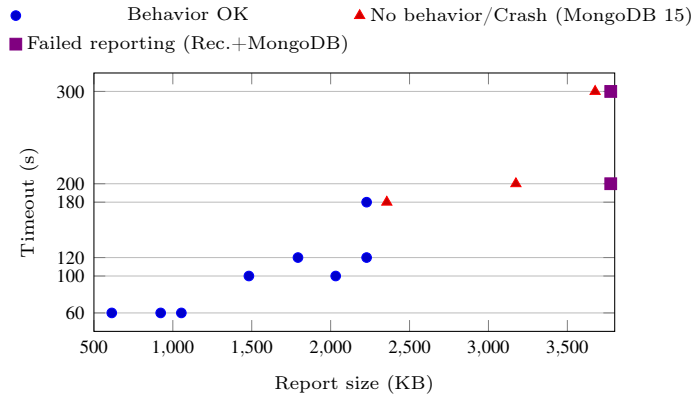


Figure 4: CAPEv2 outcomes vs. report growth.

Phase	Timeout	Total	UI outcome
OK	30s-60s	2.9-7.7 MB	No lag
Degraded	100s-120s	20-22 MB	Lags (10-20 s load)
DoA	180s	34 MB	Tab unresponsive
DoA	240s	35 MB	Unresponsive from start
DoA	300s	46 MB	Unresponsive from start
Crash	400s	51 MB	Tab crashed

Table 5: Cuckoo UI failure thresholds under telemetry growth.

4.7 Cuckoo

For Cuckoo, our tests resulted in a denial-of-analysis in the behavior-logging and report-generation pipeline of CERT-EE’s Cuckoo deployments (including `cert-ee/cuckoo3` commit `0.10.0-51-gad46ffe` and Cuckoo Sandbox 2.0.7 used by `sandbox.pikker.ee` and `cuckoo.cert.ee`) allows attackers who can submit samples for analysis to cause incomplete or missing behavioral reports by triggering unbounded recursive process creation that generates deeply nested or oversized behavior data. The excessive log volume can exceed the JSON or database serialization limits, leading to silent analysis failure or a frozen web interface.

We reproduced the issue on both public instances and locally cloned deployments. In the affected analyses, we regularly saw `process.json` and `registry.json` grow to 45-50 MB, so the underlying collectors were clearly logging a lot of activity; however, the final behavioral report was empty. Attempting to open those reports often caused the web UI to stall or the browser to freeze, and in offline clones, the large report files and UI instability prevented access to otherwise completed analyses. Representative runs and measured artifacts are summarized in Table 5.

As shown in Table 6 and respective Fig. 5, `registry.json` dominates total size across all runs, making it the primary lever for UI overload; `file.json` and `report.json`

Timeout	Total	registry.json	file.json	report.json
30s	2.9 MB	2.6 MB	183 KB	120 KB
60s	7.7 MB	7.1 MB	471 KB	304 KB
100s	20 MB	18 MB	1.3 MB	823 KB
120s	22 MB	20 MB	1.4 MB	846 KB
180s	34 MB	31 MB	2.3 MB	1.5 MB
240s	35 MB	32 MB	2.3 MB	1.5 MB
300s	46 MB	40 MB	3.0 MB	1.8 MB
400s	51 MB	46 MB	3.4 MB	2.2 MB

Table 6: Artifact composition driving the failure.

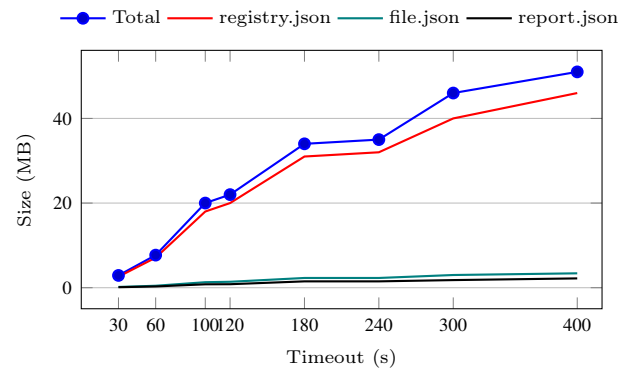


Figure 5: Growth of artifacts as timeout increases.

grow steadily and contribute to the tipping points.

5 Discussion

5.1 Scalability and Generalization

To assess whether TCAs generalize beyond our primary proof-of-concept, we conducted additional experiments varying sample characteristics, recursion parameters, and execution environments.

Cross-architecture consistency. We compiled variants of our proof-of-concept for both x86 and x64 architectures and tested them on Windows 10, Windows 11, and Windows Server 2019. The recursion depth required to trigger failures did not vary across architectures and OS versions for a given platform, indicating that the vulnerabilities are not artifacts of specific binary formats or OS configurations.

Reproducibility across runs. Each configuration was executed a minimum of three times. For the local tests, since we had access to the backend, error codes were identical across runs at the same depth. Hence, we conclude that the observed failures are deterministic consequences of pipeline limits rather than transient race conditions.

Generalization to other recursive behaviors While our primary proof-of-concept uses process spawning, the underlying vulnerability (unbounded collection meeting

bounded serialization) should manifest with any behavior that generates deeply nested or high-volume telemetry. We identify three additional vectors that require further investigation.

- **Registry Activity.** Sandboxes commonly serialize registry operations into hierarchical structures mirroring the registry's tree organization. So registry-focused TCAs may achieve comparable or greater impact with lower depth thresholds.
- **File System Operations.** Deep directory traversal and creation generate telemetry that many platforms serialize with path nesting. Windows extended-length paths support up to 32,767 characters (\\?\ prefix), but serializers may not handle such depths gracefully.
- **Network Connections.** High-frequency connection telemetry (e.g., thousands of short-lived sockets) stresses event queuing rather than nesting depth, potentially triggering the burst/queue failures modeled in Theorem 3.

We leave empirical validation of these vectors to future work, noting that the architectural patterns we identified are shared across telemetry types within each platform.

5.2 Root cause analysis and mitigation measures

Based on our findings, several practical steps can reduce the risk of DoA across monitoring platforms and the impact of TCAs.

Nesting every element in a single JSON object is suboptimal, as most languages and serializers impose depth and size constraints. Mapping processes to tree structures or graph databases can alleviate these issues; otherwise, large or deeply nested data should be split into chunks so that partial results are saved even when limits are exceeded, preventing full report failure. Telemetry should also be validated early, at the agent or collector boundary, against a strict schema and sanitized before ingestion. When information is truncated, a standardized marker should indicate alteration so analysts can identify and recover affected data. To preserve auditability, platforms should maintain an isolated, access-controlled forensic store that accepts raw artifacts when the main pipeline rejects them.

In the same vein, we highlight the crucial role of proper queue management and event collection. Sudden spikes in the monitored events and the corresponding load of recording the associated information can cause many problems, e.g., dropped events or collector crashes. Of special interest are short and temporary or recursive processes that poor queue management may miss, leading to, e.g., losing the relationship between child and parent

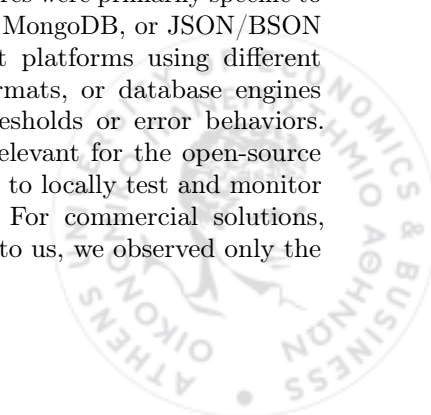
process, making them appear as orphans. We believe that delaying the processing of new processes, especially in malware sandboxes, could throttle such spikes and allow the collector to properly manage the collected information and the high number of events generated by spawned processes. Similarly, they may aggregate repetitive events (e.g., compress process-spawn sequences into summarized records) and canonicalize equivalent objects to cut down volume before transmission.

The reporting mechanisms should also become more robust. For example, incremental rendering of dashboards and reports can prevent interface issues with oversized datasets and selectively provide truncated previews when needed. In most cases, not all information has to be displayed to the user at once; therefore, some information can be prioritized, hinting that further information exists, improving usability and stability. An assessment of how much information exists and how this can be properly loaded, given the memory limits, internal size, and recursion thresholds, can also facilitate this process. Thus, dashboards must render lazily, using, e.g., virtualized trees and pagination, and avoid attempts to materialize large hierarchies in the browser or server render threads. This can be facilitated by using streaming JSON/BSON parsers to avoid loading entire documents into memory, enabling partial consumption and checkpointing of long streams.

5.3 Limitations

While TCAs introduce a new set of denial-of-analysis conditions for anti-malware solutions, we acknowledge several limitations of our work. First, although we tested multiple anti-malware solutions, the results may not be generalizable to all commercial or proprietary solutions, as differences in architecture, telemetry pipelines, and database backends could yield varying outcomes. Evidently, serialization libraries, the underlying data structures, as well as the goals that each solution has, affect the applicability of our attacks. For instance, an EDR is expected to be less prone to such an attack, as a fast forking process follows an aggressive pattern that the host does not need to be exposed to, so killing the process after a specific and small threshold is more optimal.

Second, the observed failures were primarily specific to systems relying on Python, MongoDB, or JSON/BSON serialization, meaning that platforms using different languages, serialization formats, or database engines might display distinct thresholds or error behaviors. Of course, this is mostly relevant for the open-source solutions that we were able to locally test and monitor each step of the process. For commercial solutions, since they are a black box to us, we observed only the



external behavior; internal implementation is unknown. Nevertheless, this may also mean that some relevant vulnerabilities exist, and we missed them either because we did not use the correct threshold or because the output or missed detection was a result of our attack.

Finally, while we propose some high-level mitigation strategies such as back-pressure mechanisms, incremental reporting, and robust serialization, a complete implementation and empirical evaluation of these defenses across diverse products, as well as the impact on other factors, e.g., efficiency and latency, are beyond the scope of this work.

6 Related work

Dynamic analysis of binaries with and without the use of automated sandboxing is central to malware analysis. Prior work has shown various ways in which adversaries try to defeat these systems. Evasion of malware sandboxes usually aims at the identification of the sandbox environment through various means, such as timing attacks against virtualization, registry, network, or CPU artifact attacks, and human interaction detection [13]. For example, SandPrint showed that sandbox implementations may leak stable fingerprints that malware can detect and react to [55]. Timing and resource channels remain practical. Embedding proof-of-work into malware exposes timing asymmetries between bare-metal and scaled cloud sandboxes, reducing the effectiveness of online analysis services [36].

6.1 Attacks on EDRs

While EDR systems offer far more comprehensive visibility than traditional antivirus solutions, they are not immune to evasion or abuse. Prior research and practitioner analyses have demonstrated multiple ways to bypass [24, 26, 43] or even weaponize them [3, 17].

One of the main directions in this research line targets the hooking mechanisms used by EDRs to intercept user-mode or kernel-mode APIs. Several works exploit unhooking or direct system-call invocation to execute unmonitored code paths, typically by calling native Windows functions from `ntdll.dll` without passing through instrumented stubs [4, 24, 30]. Because many EDRs rely on user-land hooks, both attackers and defenders operate at the same privilege level, creating inherent race conditions that, in the end, favor attackers. Recent work, such as Cymulate's *Blindside* technique, employed hardware breakpoints to selectively allow the loading of `ntdll.dll` while blocking other libraries [25].

Another common evasion vector leverages legitimate binaries and scripts to execute malicious payloads, known as *living-off-the-land* (LotL) tactics [33]. Attackers may also abuse DLL sideloading, placing a malicious library next to a trusted executable so that it loads

attacker-controlled code [9, 26]. Control Panel (.cpl) files have likewise been repurposed to execute arbitrary payloads outside of standard inspection paths [11, 52].

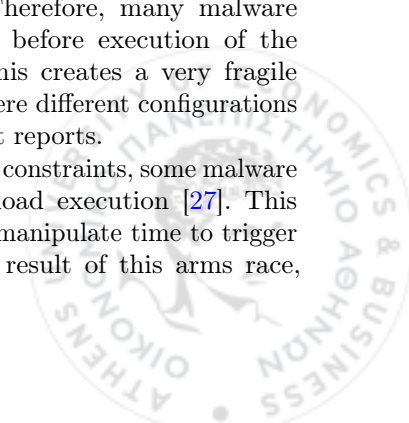
The aforementioned methods have been widely used in the real world, as well as methods to terminate the EDR processes. Indeed, some attackers may resort to the use of vulnerable drivers to terminate the corresponding processes, as in the case of GhostEngine [10]. Moreover, following the Malware-as-a-Service paradigm, cybercriminals sell tools like Baphomet EDR Killer [19], which automate this process and disable EDRs. Likewise, more advanced attackers have been reported manipulating Windows Defender Application Control (WDAC) policies to disable EDRs or whitelist malicious binaries [7].

More offensive strategies attempt to blind EDRs by blocking or corrupting their telemetry. For instance, tools such as EDRSilencer use Windows Filtering Platform (WFP) rules to cut off network traffic to EDRs [5], while EDR-Freeze [47] exploits Windows Error Reporting to suspend the monitoring processes.

6.2 Attacks on malware sandboxes

Contrary to EDRs, malware sandboxes allow malware to execute, as their goal is to actually observe what the malware does when it is executed. Therefore, malware tries to hide as much information as possible. Therefore, malware needs to identify the environment in which it operates and determine whether it will continue its operation. For instance, since the malware sandbox is a virtualized environment, malware tries to determine whether they are being executed in a virtual machine. Side information, e.g., names of devices or running processes, hardware identifiers, screen resolution, available memory, or even available threads and processors can easily give away the execution in a virtualized environment [13, 28, 48]. Similarly, a malware may try to find drivers, registry keys, or even folder names associated with virtual machines. Going a step further, malware may try to find elements of human interaction to determine whether it is executed in a malware sandbox. For instance, the uptime, the lack of keyboard and mouse interaction, the absence of browser history, or recent file activity can be used to infer the environment. Therefore, many malware would perform these checks before execution of the actual payload. Moreover, this creates a very fragile and sensitive environment where different configurations can provide radically different reports.

Given the resource and time constraints, some malware may opt to delay their payload execution [27]. This forced malware sandboxes to manipulate time to trigger their execution faster. As a result of this arms race,



malware tries to measure the timing of specific functions to determine whether the elapsed time is aligned with the expected processing time. Evidently, time plays an important role in the dynamic analysis of malware [29].

For more on evasive malware, the interested reader may refer to [22].

6.3 Denial-of-Service attacks on security solutions

Petsios et al. presented SlowFuzz, a method to automatically synthesize inputs that trigger worst-case for several well-known algorithms, causing severe behavioral issues in widely deployed libraries [42]. For what is relevant in this work, many various zip parsers used in antivirus software whose decompression was significantly delayed. Other related work detected and repaired DoS-prone regex patterns, often used by cybersecurity solutions to detect strings or bytes associated with malicious activity (e.g., ReDoSHunter, RegexScalpel) and exposed vulnerabilities in non-backtracking matchers [31, 32, 51]. In content-inspection pipelines, adversarial inputs that induce super-linear or exponential matching can exhaust CPU budgets, delay processing, and cause packet or alert drops.

6.4 Positioning

Our attacks are conceptually related in spirit to classic structured-data and resource-exhaustion techniques such as the ‘billion laughs’ XML entity expansion and fork bombs. Although the ‘billion laughs’ attack was initially targeted at the XML parsers [50], similar attacks can be launched against other parsers, e.g., YAML [44] and SAML [49]. Likewise, fork bomb attacks target resource limits by rapidly spawning processes [8]. Thus, we sit between these two ideas. Like billion-laughs, we exploit pathological data structures, and like fork bombs, we exploit process creation to exhaust downstream resources. However, we try to achieve completely different goals on different systems and architectures. In this regard, we consider our work to be closely related to the recent work of Filic et al. [20], in which the authors exploit implementations of well-known probabilistic data structures in various solutions.

More precisely, our focus falls in the Denial-of-Service space against anti-malware and malware sandboxes, but differs from current approaches in that we target weaknesses in data structures and telemetry-handling paths of monitoring systems. Instead of hiding or escaping containment, we weaponize the components used in data handling to induce denial-of-service or analysis failures. Not only do we effectively demonstrate

that these types of structures are widely used in many industry anti-malware solutions, but, to the best of our knowledge, we also show that this attack surface is underexplored and follows a distinct line of research, including evasion, regex-based availability, and other attacks.

7 Conclusions

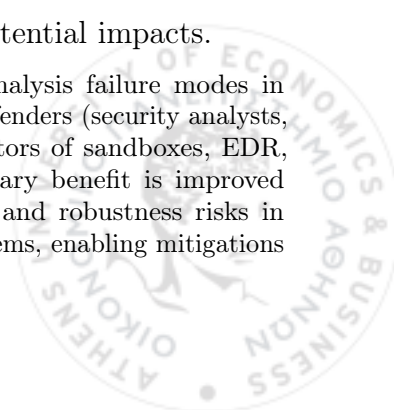
This work introduces TCAs, a novel method for inducing denial-of-analysis in modern anti-malware and telemetry monitoring systems by exploiting weaknesses in their data-handling and storage pipelines. We demonstrated that behavioral telemetry, not just sensors or hooks, can be an attack surface. Unlike traditional techniques that focus on hiding malicious behavior or evading detection, we directly target the internal telemetry infrastructure, demonstrating that unbound collection, serialization, and storage of monitoring data can itself be weaponized to disrupt analysis. Indeed, across 13 widely deployed products and cloud services, seven exhibited some form of denial-of-analysis or behavioral invisibility. Our findings led to the assignment of three CVE identifiers, with one additional CVE pending for Cuckoo. Other vendors patched or reconfigured their software following coordinated disclosure. Notably, the orjson vulnerability demonstrates that TCAs can expose weaknesses not only in security tools but also in widely used serialization libraries.

Recursive workloads in experiments pushed JSON and BSON serialization to their limits, suggesting a need for more thorough stress testing of deeply nested and complex data structures to identify thresholds. Moreover, our results highlight the importance of viewing telemetry pipelines as potential attack surfaces and hardening them accordingly. Thus, in future work, we plan to explore fuzzing of telemetry structures and graph-based storage for behavioral data. Moreover, since Python and MongoDB were at the core of most observed denial-of-analysis conditions, it is essential to test other programming languages, serialization libraries, and database engines to check whether similar weaknesses exist.

A Ethical Considerations

A.1 Stakeholders and potential impacts.

Our work studies denial-of-analysis failure modes in telemetry pipelines used by defenders (security analysts, SOCs) and by vendors/operators of sandboxes, EDR, and DFIR tooling. The primary benefit is improved understanding of availability and robustness risks in widely deployed defensive systems, enabling mitigations



that reduce blind spots for end users. A key risk is that the techniques could be misused by attackers to degrade defenders' visibility.

A.2 Risk mitigation in experimentation.

All local experiments were conducted in isolated, offline laboratory networks using virtual machines to prevent unintentional spread, external communication, or impact on third-party systems. When our test samples included network-capable behaviors (e.g., a reverse shell), they were configured to communicate only with a host under our control inside the isolated environment, and were not used against production systems. We did not target real user devices or collect personal data; logs, exported reports, and UI traces were retained only to support reproducibility and debugging of failure modes.

Listing 1: Nested JSON for spawned processes.

```

1  {"processes": {
2    "id": "id1",
3    "timestamp": 1761675797,
4    "children": [
5      {
6        "id": "id2",
7        "timestamp": 1761675798,
8        "children": [
9          {
10         "id": "id3",
11         "timestamp": 1761675797,
12         "children": [
13           {
14             "id": "id4",
15             "timestamp": 1761675799,
16             "children": [...]
17           ]
18         ... ]}

```

A.3 Use of public/third-party services.

For online sandbox validation, we submitted only sanitized samples intended to reproduce structural/telemetry stress behaviors without enabling deployment or abuse. Submissions were rate-limited and performed in a manner intended to minimize operational impact on service providers.

A.4 Responsible and coordinated disclosure.

All vulnerabilities discovered during this research were disclosed in accordance with coordinated and responsible disclosure practices. For each confirmed vulnerability, we followed a consistent process. (i) contact maintainers via their preferred security reporting channels within 48 hours of confirming exploitability; (ii) provide a 90-day disclosure window before public disclosure,

with extensions upon request for lower-severity issues or issues requiring significant prerequisites, consistent with common disclosure policies (e.g., Project Zero [23]); (iii) include technical details (affected versions, proof-of-concept, and suggested remediation when possible); and, where applicable, (iv) coordinate CVE assignment with maintainers who acknowledged the report and pursued remediation. When maintainers did not acknowledge the report, disputed that the issue constituted a vulnerability, or did not provide a remediation timeline, we proceeded with disclosure after the 90-day notification period, making reasonable efforts to minimize risk. In some cases, issues without a published CVE were still disclosed to maintainers and resulted in patches or configuration hardening; in one case, a vendor did not consider the issue a vulnerability and granted permission to proceed with publication.

Artifact release considerations. To reduce misuse risk prior to widespread deployment of defenses, we avoid releasing weaponized exploit chains or directly deployable binaries. Where artifacts are shared for review and reproducibility, we provide sanitized or structure-preserving variants sufficient to validate methodology and reproduce the reported resource/limit behaviors, and we document any omissions and their rationale.

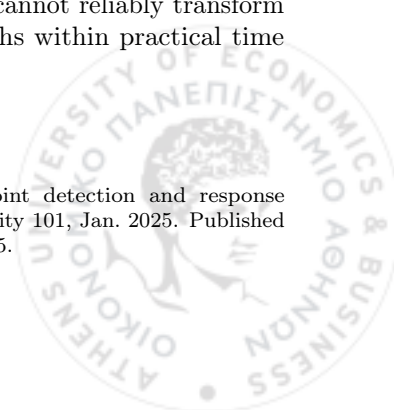
B UI/analysis degradation and process count stress test

The CAPE pipeline exhibits non-linear failure modes under high process counts: at modest loads (e.g., 1 process), behavior is recorded, but at thousands of processes, the system often produces no behavioral output, and at 10,000 processes the reporting stage can fail entirely. These outcomes were detected without tampering with sensors or instrumentation. Failure was induced by the volume of nested processes overwhelming downstream serialization and storage.

In contrast to hard reporting failures, Cuckoo shows progressive degradation: UI latency grows from seconds to tens of seconds (and up to ~80s around 9,000 processes), and the process tree is frequently truncated while the page becomes unresponsive or slow. This behavior is consistent with telemetry pipelines that continue collecting events but cannot reliably transform and render large process graphs within practical time and resource limits.

References

- [1] Aarness, A. What is endpoint detection and response (edr)? CrowdStrike Cybersecurity 101, Jan. 2025. Published 2025-01-07; accessed 2026-01-05.



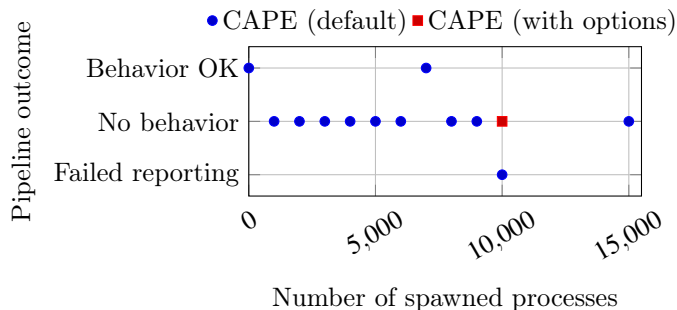


Figure 6: CAPE process-tree stress results. Increasing the number of spawned processes causes missing behavior reports and, at higher loads, reporting failures (e.g., serialization/DB-layer errors), despite execution continuing in user mode.

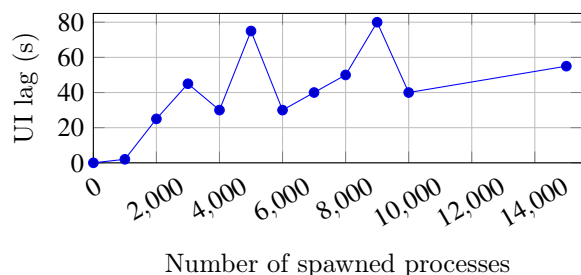


Figure 7: Cuckoo process-count stress results. As the number of processes increases, the analysis UI degrades sharply, with long load times and tree truncation becoming common, indicating a progressive loss of analyst visibility before outright failure.

[2] Afianian, A., Niksefat, S., Sadeghiyan, B., and Baptiste, D. Malware dynamic analysis evasion techniques: A survey. *ACM Trans. Web* 9, 4 (June 2018), Article 39 (28 pages).

[3] Alachkar, K., Gaastra, D., Barbaro, E., van Eeten, M., and Zhauniarovich, Y. EvilEDR: Repurposing EDR as an offensive tool. In *34th USENIX Security Symposium (USENIX Security 25)* (2025), pp. 587–605.

[4] Apostolopoulos, T., Katos, V., Choo, K.-K. R., and Patsakis, C. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Generation Computer Systems* 116 (2021), 393–405.

[5] Au, C. EDRSilencer. <https://github.com/netero1010/EDRSilencer>, 2023.

[6] AV-TEST Institute. Malware Statistics & Trends Report. <https://www.av-test.org/en/statistics/malware/>. Accessed Jan. 2026.

[7] Beierle, J. A Nightmare on EDR Street: WDAC's Revenge. <https://beierle.win/2025-08-28-A-Nightmare-on-EDR-Street-WDACs-Revenge/>, 2025.

[8] Berlot, M., and Sang, J. Dealing with process overload attacks in unix. *Information Security Journal: A Global Perspective* 17, 1 (2008), 33–44.

[9] Biswas, S. AQUARMOURY. <https://github.com/reveng007/AQUARMOURY>, 2021.

[10] Bitam, S., Bousseaden, S., DeJesus, T., and Pease, A. Invisible miners: unveiling GHOSTENGINE's crypto mining operations. <https://www.elastic.co/security-labs/invisible-miners-unveiling-ghostengine>, 2024.

[11] Borosh, S. CPLResourceRunner. <https://github.com/rvrsh3ll/CPLResourceRunner>, 2018.

[12] Bosworth, R. Decrypting sentinelone cloud detection | the behavioral ai engine in real-time cwpp. *SentinelOne Blog*, July 2025. Updated 2025-07-19; accessed 2026-01-05.

[13] Bulazel, A., and Yener, B. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium (New York, NY, USA, 2017)*, ROOTS, Association for Computing Machinery.

[14] Cable, J. Ransomwhere: A crowdsourced ransomware payment dataset, Oct. 2024.

[15] Cisco. Secure malware analytics (threat grid) integration. *Cisco Meraki Documentation*. Accessed 2026-01-05.

[16] Cisco. Cisco threat grid cloud: Data sheet. Tech. rep., Cisco, July 2021. Accessed 2026-01-05.

[17] Cohen, S. The dark side of EDR: Repurpose EDR as an offensive tool. <https://www.safebreach.com/blog/dark-side-of-edr-offensive-tool/>, 2024.

[18] CrowdStrike. CrowdStrike 2025 Global Threat Report. <https://www.crowdstrike.com/en-us/global-threat-report/>, 2025. Accessed: 2025-08-20.

[19] Cuprik, R. EDR killers get popular. Here is how to stop them. <https://www.eset.com/blog/en/business-topics/threat-landscape/stop-edr-killers/>, 2025.

[20] Filic, M., Hofmann, J., Markelon, S. A., Paterson, K. G., and Unnikrishnan, A. Probabilistic data structures in the wild: A security analysis of redis. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy, CODASPY 2025, Pittsburgh, PA, USA, June 4-6, 2025* (2025), J. Joshi, J. Vaidya, and H. Schulmann, Eds., ACM, pp. 167–178.

[21] Fortinet. Unleashing the power of sandboxing: A crucial need for organizations. Tech. rep., Fortinet, Nov. 2024. Published 2024-11-08; accessed 2026-01-05.

[22] Galloro, N., Polino, M., Carminati, M., Continella, A., and Zanero, S. A systematical and longitudinal study of evasive behaviors in windows malware. *Comput. Secur.* 113 (2022), 102550.

[23] Google Project Zero. Vulnerability disclosure policy, 2025. Accessed: 2025-01-21.

[24] Junior, H. C. Hookchain: A new perspective for bypassing edr solutions. *arXiv preprint arXiv:2404.16856* (2024).

[25] Kalendarov, I. EDR Evasion: A New Technique Using Hardware Breakpoints – Blindside. <https://cymulate.com/blog/blindside-a-new-technique-for-edr-evasion-with-hardware-breakpoints/>, 2025.

[26] Karantzas, G., and Patsakis, C. An empirical assessment of endpoint detection and response systems against advanced persistent threats attack vectors. *Journal of Cybersecurity and Privacy* 1, 3 (2021), 387–421.

- [27] Kolbitsch, C., Kirida, E., and Kruegel, C. The power of procrastination: detection and mitigation of execution-stalling malicious code. In Proceedings of the 18th ACM Conference on Computer and Communications Security (New York, NY, USA, 2011), CCS '11, Association for Computing Machinery, p. 285–296.
- [28] Koutsokostas, V., and Patsakis, C. Python and malware: Developing stealth and evasive malware without obfuscation. arXiv preprint arXiv:2105.00565 (2021).
- [29] Küchler, A., Mantovani, A., Han, Y., Bilge, L., and Balzarotti, D. Does every second count? time-based evolution of malware behavior in sandboxes. In 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021 (2021), The Internet Society.
- [30] Lewis, T. M., and Rimal, B. P. Effects of removing user-land hooks in endpoint protection during attack experiments. IEEE Access 12 (2024), 15820–15844.
- [31] Li, Y., Chen, Q., Wang, Y., and Chen, S. ReDoSHunter: A combined static and dynamic approach for eliminating regular expression DoS vulnerabilities. In USENIX Security Symposium (2021).
- [32] Li, Y., Zhou, J., Wang, Y., Xu, W., and Chen, S. RegexScalpel: Regular expression denial of service vulnerability repair. In USENIX Security Symposium (2022).
- [33] LOLBAS-Project. Living Off The Land Binaries, Scripts and Libraries. <https://lolbas-project.github.io/>, 2026.
- [34] Microsoft. Overview of endpoint detection and response capabilities - microsoft defender for endpoint. Microsoft Learn, Sept. 2025. Last updated 2025-09-29; accessed 2026-01-05.
- [35] MongoDB. MongoDB limits and thresholds. <https://www.mongodb.com/docs/manual/reference/limits/?atlas-provider=aws&atlas-class=general>, 2024.
- [36] Nappa, A., Papadopoulos, P., Varvello, M., Gomez, D. A., Tapiador, J., and Lanzi, A. Pow-How: An enduring timing side-channel to evade online malware sandboxes. In European Symposium on Research in Computer Security (ESORICS) (2021).
- [37] National Institute of Standards and Technology. CVE-2025-61301 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2025-61301>, 2025.
- [38] National Institute of Standards and Technology. CVE-2025-61303 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2025-61303>, 2025.
- [39] National Institute of Standards and Technology. CVE-2025-67221 Detail. <https://nvd.nist.gov/vuln/detail/CVE-2025-67221>, 2025.
- [40] Palo Alto Networks. Analysis environment (advanced wildfire). Palo Alto Networks Documentation. Accessed 2026-01-05.
- [41] Palo Alto Networks. Review wildfire analysis details (cortex xdr administrator guide). Cortex XDR Documentation. Accessed 2026-01-05.
- [42] Petsios, T., Polakis, I., Keromytis, A. D., and Jana, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In ACM Conference on Computer and Communications Security (CCS) (2017).
- [43] Pogonin, D., and Korokin, I. Microsoft defender will be defended: Memoryranger prevents blinding windows av. arXiv preprint arXiv:2210.02821 (2022).
- [44] Rasheed, S., Dietrich, J., and Tahir, A. Laughter in the wild: A study into dos vulnerabilities in yaml libraries. In 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE) (2019), pp. 342–349.
- [45] Recorded Future. Triage. <https://tria.ge/>, 2025.
- [46] Recorded Future. Triage sample. <https://tria.ge/250822-qfystshk51>, 2025.
- [47] Salarium, Z. EDR-Freeze. <https://github.com/TwoSevenOneT/EDR-Freeze>, 2025.
- [48] Shi, H., Alwabel, A., and Mirkovic, J. Cardinal pill testing of system virtual machines. In 23rd USENIX Security Symposium (USENIX Security 14) (San Diego, CA, Aug. 2014), USENIX Association, pp. 271–285.
- [49] Somorovsky, J., Mayer, A., Schwenk, J., Kampmann, M., and Jensen, M. On breaking SAML: Be whoever you want to be. In 21st USENIX Security Symposium (USENIX Security 12) (Bellevue, WA, Aug. 2012), USENIX Association, pp. 397–412.
- [50] Späth, C., Mainka, C., Mladenov, V., and Schwenk, J. SoK: XML parser vulnerabilities. In 10th USENIX Workshop on Offensive Technologies (WOOT 16) (Austin, TX, Aug. 2016), USENIX Association.
- [51] Turonov'a, L., Jeuring, J., and Middelkoop, A. Exposing ReDoS vulnerability of non-backtracking matchers. In USENIX Security Symposium (2022).
- [52] Tylous. ScareCrow. <https://github.com/Tylous/ScareCrow>, 2023.
- [53] Velocidex. Bugfix: Limit number of entries emitted into process_tracker_tree() #4454. <https://github.com/Velocidex/velociraptor/pull/4454>, 2025.
- [54] Wazuh, Inc. Wazuh Documentation — Log Data Indexing and Storage. <https://documentation.wazuh.com/current/getting-started/use-cases/log-analysis.html>. Accessed Jan. 2026.
- [55] Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., and Rossow, C. Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In Research in Attacks, Intrusions and Defenses (RAID) (2016).

