POLITECNICO DI TORINO

Master degree course in Cybersecurity

Master Degree Thesis

Forensic Analysis of Malware: Identification of Indicators of Compromise and Automation of the Investigative Process in Windows and Linux Environments

Supervisor
prof. Atzeni Andrea

Candidate
Cosimo Vergari

Academic Year 2024-2025

Summary

Digital forensic analysis is fundamental for comprehending malware attacks and rebuilding the actions taken by attackers to compromise information systems. Modern malware frequently employs sophisticated persistence and evasion techniques that leave behind evidence, known as Indicators of Compromise (IoCs), on various artefacts, including disk, memory, and network environment. Identifying such IoCs is of prime importance for post-mortem analysis, enabling analysts to infer attacker actions and impact on the system. Nonetheless, the amount and intricacy of forensic data present significant problems, therefore rendering the process laborious and prone to oversights.

This thesis examines the identification of IoCs in Windows and Linux environments and explores the combination of automation and AI to assist forensic workflows. The study focuses on three main areas: file system and disk artifacts, memory dumps, and network traffic. A proof-of-concept framework was developed for automating the extraction, analysis, and reporting of forensic evidence. Disk and file system artifacts are examined with Plaso and the Digital Forensics Virtual File System (DFVFS) python library, Memory dumps are analyzed through Volatility3 plugins, and network captures are analyzed via automatic extraction of contacted ips and domains and checking them against the VirusTotal Threat Intelligence APIs. The framework can generate a compiled, systematic report that combines all the IoCs identified and highlights unusual behavior for further analysis via forensic investigators. The generated report also suggests next steps to the Forensic Team and tries to provide a verdict about the state of the machine (compromised or not).

The framework was evaluated with actual malware samples within managed virtual machine environments. Findings show significant time savings via automation, as a number of forensic processes, done manually for centuries, become a single workflow. In addition, the thesis compares the performance and practicality of local AI models against world-leading cloud AI solutions (e.g., Gemini 2.5 Pro) via the application of some prompt engineering and optimisation techniques like chunking. It also evaluates the compromise between efficiency and privacy. Large language models are employed for the purpose of understanding extracted evidence, determining potential attack patterns, and generating report narratives, providing analysts with intelligible information along with ongoing expert direction.

The innovations of methodology presented here are threefold: (i) systematic evaluation and the related IoCs within the disk, memory, and network spaces for Windows and Linux; (ii) an experimental proof-of-concept framework showing the automated acquisition, examination, and reporting of forensic evidence; and (iii) an insightful analysis of AI-assisted forensic workflows, highlighting advantages, limitations, and privacy concerns. This work showcases that combining IoC-focused analysis with AI-supported automation can enhance the efficiency, consistency, and usability of digital forensic investigations, setting the stage for more scalable post-mortem analysis of malware.

Contents

Chapter 1

Introduction

1.1 Background and motivation

Cyberattacks and advanced malware pose a major threat to information systems nowadays. Digital forensics sits at the heart of investigating these incidents, helping analysts and researchers understand techniques, locate infected assets, and reconstruct timelines. By examining disk, memory, and network artifacts, investigators identify malicious activity through Indicators of Compromise (IoCs), which clarify attacker's actions and aid inference of likely objectives.

The emerging challenge in this field is scale and complexity. Modern malware often uses stealthy, persistent methods and leaves behind fragmented, confusing traces. Manually triaging terabytes of data scattered across diverse sources is laborious, slow, and prone to error. Traditional workflows struggle to match the volume and velocity of today's threats, increasing the risk of potential oversights and delayed incident response.

Manual effort has long been the bottleneck. Effective forensic work demands deep knowledge across file systems, memory internals, and network protocols. With data volumes expanding, completing sufficient analysis within operational timelines is increasingly difficult–yet essential for mitigating damage and preventing follow-on attacks. These pressures underscore the need for creative approaches that improve the efficiency, reliability, and reproducibility of malware forensics.

1.2 Problem statement

Digital forensic analysis of malware is currently hindered by the numerous and sophisticated nature of digital evidence. The manual extraction of Indicators of Compromise (IoCs) from different artifacts, such as file system entries, memory dumps, and connections in the network, is a resource-consumption operation which is not scalable. This manual workflow increases the potential for human error and significantly increases the time required for post-mortem analysis, thus pushing delay in response actions and critical insights. The lack of an integrated, automated process for collecting, analyzing, and reporting forensic evidence from multiple sources represents a key limitation in current practice.

1.3 Research objectives

This thesis aims to help addressing the above limitations by analyzing the application of automation and Large language Models (LLMs) to enhance the digital forensic analysis of malware. The primary objectives are the following:

- **Systematic IoC Identification:** To systematically identify and analyze the most important Indicators of Compromise out of three main forensic artifacts: disk image file system, memory dumps, and network traffic. The analysis will be conducted under both Windows and Linux operating systems.
- **Development of an Automated Framework:** To design and build an automated proof-of-concept framework for forensic evidence collection, analysis, and reporting. The framework will integrate established open-source tools with custom logic for optimizing the investigative process.
- **Analysis of AI-Assisted Workflows:** To examine the effectiveness and viability of leveraging large language models (LLMs) to support forensic analysis. This includes quantifying their capacity to parse evidence, identify attack patterns, and generate comprehensive report narratives, while balancing the efficiency, privacy, and performance trade-offs between on-premise and cloud-based AI deployments.

1.4 Scope of the Thesis

This research focuses on post-mortem analysis of malware in Windows and Linux operating systems that run on managed virtual machines. The study limits itself to three broad categories of forensic evidence: disk and file system artifacts (using Plaso and DFVFS), memory dumps (using Volatility3), and network captures (using automated domain/IP extraction and API-based threat intelligence). The proof-of-concept framework developed is built to demonstrate a systematic, automated workflow, and its testing is conducted on actual malware samples. The AI component of the research specifically utilizes and compares the performance of local and cloud-based LLMs, like Gemini 2.5 Pro, for interpretation of evidence and report generation. This thesis does not concern handling real-time incidents or coming up with new malware detection schemes.

1.5 Contributions of the Thesis

The significant contributions of the present thesis are threefold:

- A structured methodology for correlating and analyzing IoCs in the disk, memory, and network areas of Windows and Linux systems and providing a general overview of a breach.
- A proof-of-concept experimental setup that demonstrates an efficient, automated pipeline for the acquisition, analysis, and systematic reporting of forensic evidence with significant time savings compared to traditional manual methods.
- A reflective consideration of the integration of AI models within forensic workflows, presenting the advantages, constraints, and key privacy concerns associated with the use of large language models to support digital investigations.

This research collectively demonstrates how an IoC-focused analysis, powered by AI-augmented automation, can significantly enhance the efficacy, reliability, and usability of digital forensic analyses to pave the way for more efficient and scalable post-mortem malware analysis.

1.6 Thesis structure

This thesis consists of six chapters:

- **Chapter 1: Introduction** provides an overview of the research topic, states the problem, formulates the objectives of the study, and elucidates the scope and the contribution of the research.

- **Chapter 2: Theorical Background and Literature Review** discusses existing research and methodologies used in digital forensic analysis, with a focus on IoCs identification.
- **Chapter 3: Methodology** describes the tools and techniques used for data collection. It explains the setup of the virtual environments, the process for executing malware samples and for the collection of artefacts from the infected machines.
- **Chapter 4: Framework Implementation** describes the architecture and implementation of the automated forensic framework, giving a detailed analysis on how the framework works.
- **Chapter 5: Results Discussion and Evaluation** evaluates the performance of the built framework, analyzes the findings with regard to the performance of AI assistance, and compares the efficiency and privacy impacts of different AI models.
- **Chapter 6: Conclusion and Future Work** summarizes the key findings of the thesis, reformulates the main contributions, and describes potential future research avenues.

Chapter 2

Theoretical Background and Literature Review

2.1 Introduction to Digital Forensics

In its strictest connotation, Digital Forensics is the application of computer science and investigative procedures involving the examination of digital evidence - following proper search authority, chain of custody, validation with mathematics, use of validated tools, repeatability, reporting, and possibly expert testimony[1]. To maintain its reliability in legal and investigative contexts, digital forensics is grounded in a set of principles: evidence integrity, strict chain of custody, repeatability of methods, and admissibility in court. These ensure that findings are not only technically sound but also defensible in judicial proceedings. Digital Forensics spans multiple domains, including file system analysis (e.g., recovering deleted files or timestamps), memory forensics (e.g., uncovering running processes or injected code) and network forensics (e.g., tracing malicious traffic or intrusions). Each domain requires specialized methods and tools, but all share the same forensic rigor. Over time, digital forensics has had to adapt to numerous challenges: stronger encryption, anti-forensic techniques designed to obscure traces, increasing sophistication of malware, and the rapid evolution of computing environments. These factors complicate the extraction and interpretation of evidence, pushing the field toward more automation and intelligence-driven approaches. Within this context, digital forensics plays a crucial role in malware investigations, where identifying and correlating Indicators of Compromise (IoCs) can provide actionable insights into attacker behavior. Automating the detection, structuring, and reporting of IoCs not only accelerates investigations but also strengthens the reliability of forensic findings—making it a central theme of this research.

2.2 Indicators of Compromise (IoCs)

2.2.1 Definition and classification

An *Indicator of Compromise* (IoC) is a malicious indicator whose presence on a device or network indicates the device might have been compromised [2]. Also, IoCs "signal that a data breach, intrusion, or cyberattack has occurred" by revealing traces of malicious activity [3]. In Advanced Persistent Threats (APT) and postmortem workflows, IoCs are the traces that SIEMs (Security Information and Event Management) correlate to surface attacker presence and actions across the kill chain (step-by-step model describing the phases of a cyberattack from initial reconnaissance through the attacker's end goal), supporting incident reconstruction and threat hunting at host and network levels. Their practical value is tactical: rapid detection, containment, and enrichment of cases with actionable context tied to known malware, infrastructure, and TTPs (Tactics, Techniques, and Procedures) via structured Cyber Threat Intelligence (CTI) feeds. [4]

IoCs can be categorized in three main types:

- **Atomic IoCs:** Atomic indicators are those which cannot be broken down into smaller parts and retain their meanings in the context of an intrusion. Examples of atomic indicators include IP addresses and domain names. [4]
- **Computed IoCs:** Computed indicators are those which are derived from data involved in an incident. Examples of computed indicators include hash values and regular expressions. [4]
- **Behavioral IoCs:** Behavioral indicators are collections of computed and atomic indicators, often subject to qualification by quantity and possibly combinatorial logic. An example of a complex behavioral indicator could be repeated social engineering attempts of a specific style via email against low-level employees to gain a foothold in the network, followed by unauthorized remote desktop connections to other computers on the network delivering specific malware; a simpler example could be a document file creating an executable object. Such indicators are captured as tactics, techniques and procedures (TTP), representing the modus operandi of the attacker. [4]

Another classification can be based on the source of the IoC, which can be:

- **Network-based IoCs:** Artifacts derived from network traffic or configurations. Typical examples include malicious IP addresses or domains, unusual DNS queries, or anomalous packet flows. These indicators are detected by analyzing network logs and flow data.[3]
- **Host/file-based IoCs:** Artifacts found on endpoints. Examples include suspicious file hashes, file paths/names, malicious Windows Registry keys, or specific email attachment signatures. Forensic tools can scan disks for known malware hashes or YARA signatures in files.[3]

Some IoCs are more valuable than others. Atomic markers such as hashes and IPs are easy for attackers to change ("low on the pyramid of pain"), whereas higher-level indicators (e.g., behavioral patterns or TTPs) are harder to evade [5]. In practice, analysts use a hierarchy, such as David Bianco's *Pyramid of Pain* 2.1, to prioritize indicators from low-level (file hashes, IPs) up to high-level tactics and procedures.

2.2.2 Standards and representation formats

To share and automate IoC information, the community has adopted standardized formats. The *Structured Threat Information Expression* (STIX) is a widely used language and JSON-based schema for encoding threat intelligence, including indicators [7] [8]. A STIX `Indicator` object contains a pattern (using STIX Cyber Observable expressions) that describes malicious activity (e.g., a URL or file pattern) which can be matched against observed data [7] [8]. Similarly, the *Incident Object Description Exchange Format* (IODEF) is an IETF standard for exchanging incident information in a machine-readable way [9]. These formats allow IoCs (IP addresses, hashes, domain names, etc.) to be packaged with metadata (timestamps, confidence levels, descriptions) for sharing across organizations.

Specialized signature formats are also used. For example, *YARA* is a rule-based language for defining malware signatures: a YARA rule contains text or binary patterns that identify a malware family in files [8]. IoC feeds often include YARA rules to detect file-based threats. Other legacy formats include *OpenIOC* and sharing protocols such as *TAXII* (for transporting STIX data) [8]. In short, STIX/TAXII form a modern framework for sharing IoCs and threat intelligence across systems.

For broader context, IoCs are often mapped to frameworks such as *MITRE ATT&CK*, which defines adversary tactics and techniques. ATT&CK is "a globally-accessible knowledge base of adversary tactics and techniques based on real-world observations" [10]. Linking IoCs to ATT&CK helps analysts understand how an indicator (e.g., a registry change or a file hash) fits into an attacker's overall behavior.

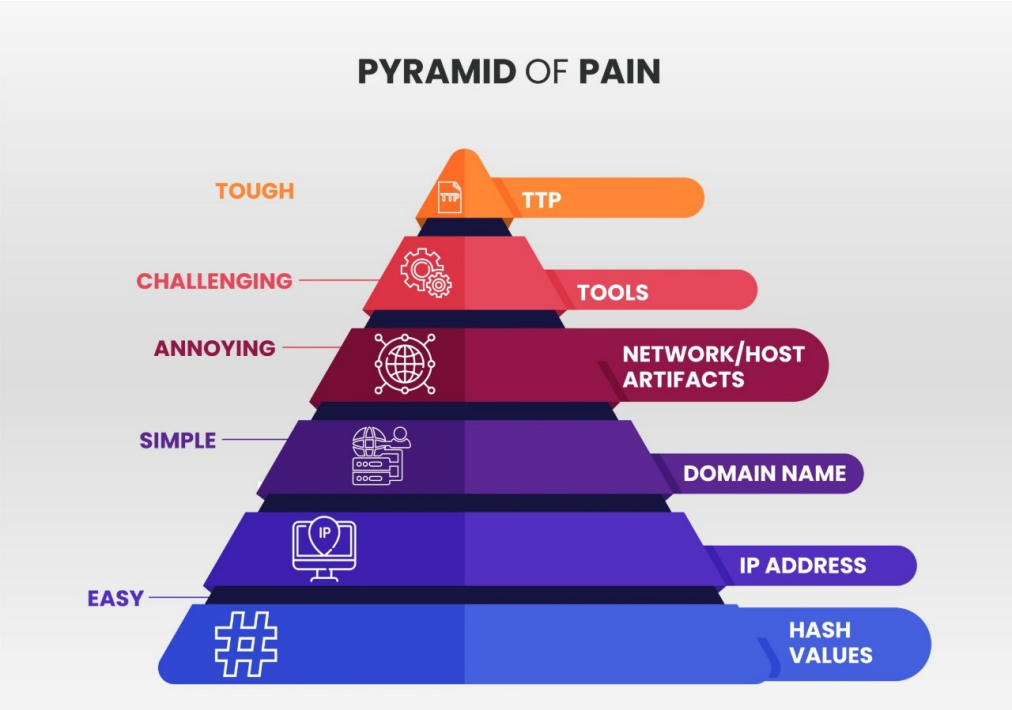

Figure 2.1. Pyramid of Pain, Source: [6]

2.3 Digital forensic investigation lifecycle

Digital forensic investigations generally follow a multi-phase lifecycle [11]. A common model, described in NIST SP 800-86, includes *Collection (Acquisition)*, *Examination/Analysis*, and *Reporting* [11]. Each phase has specific goals and requirements:

- **Acquisition:** This initial phase involves identifying, seizing, and imaging digital media while preserving its integrity [11]. Investigators create bit-for-bit forensic copies of storage devices (hard drives, SSDs, mobile storage) and may also capture volatile memory (RAM) and network logs. Tools such as write-blockers and cryptographic hashing are used to ensure the original data is not altered and its integrity can be verified. Chain-of-custody documentation is established at this stage. Because some data (e.g., RAM contents, active network connections) is transient, collection must be timely and methodical [11].
- **Analysis:** In this phase, examiners process and interpret the acquired data [11]. They use automated tools and manual techniques to extract relevant artifacts (files, emails, logs, registry entries, etc.), recover deleted content, and correlate events. For instance, timeline analysis may reconstruct a sequence of file accesses; keyword searches might locate incriminating documents; and log analysis can trace attacker activity. The analysis is guided by investigative questions and hypotheses. Sophisticated methods (e.g., carving unallocated space, parsing file systems, or using Volatility plugins) help uncover hidden or complex evidence. NIST notes that analysis must use "legally justifiable methods" to derive useful information from the data [11].
- **Reporting:** The final phase documents findings in a clear, structured report [11]. The report details the procedures used, the evidence discovered, and how conclusions were reached [11]. It may include summaries of tool output, diagrams (e.g., timelines, graphs of network flows), and expert interpretation. Effective reporting translates technical results into understandable language for non-technical stakeholders (e.g., judges, attorneys) and highlights the evidentiary value. Recommendations for security improvements or further investigation can also be included [11]. NIST emphasizes that the report should describe the methods and tools used, and provide a rationale for each conclusion [11].

Each phase is iterative: findings in analysis may prompt the collection of additional evidence, and reporting may reveal gaps requiring further examination. However, these core stages — acquisition, analysis, and reporting — form the backbone of any digital forensic process [11].

2.4 Analysis techniques in different domains

2.4.1 File system forensics

File system forensics focuses on on-disk data structures [12]. Investigators analyze the file system architecture (e.g., NTFS, FAT32, ext4) to reconstruct directories, files, and metadata [12]. This enables the recovery of deleted or hidden files and timestamps. For example, examining the NTFS Master File Table (MFT) and `$LogFile` can reveal when files were created, modified, or accessed [12]. Analysts also examine unallocated space (free areas) for file remnants, a process known as *file carving*, and review artifacts such as system logs, shortcut (LNK) files, and registry hives for user activity.

For NTFS, practitioners manually interpret Master File Table (MFT) records and attributes, including `$Bitmap`, `$AttrDef`, and `$UpCase`, while correlating redo/undo operations of `$LogFile` with MFT sequence numbers and directory relationships. Additional examination of unallocated clusters, slack space, and shadow copy metadata enables the recovery of orphaned records. User-centric traces, such as ShellBags, shortcut (LNK) files, and Most Recently Used (MRU) entries, are correlated only after establishing metadata coherence [13, 14].

In the ext4 file system, investigators parse superblocks, group descriptors, inodes, and extent-based structures. Journal transactions and bitmaps are then validated to test allocation integrity, while updates to directory entries are ordered with respect to journal commits to infer causal sequencing [15, 16]. File carving is employed cautiously as a hypothesis-driven approach, with recovered fragments treated as low-confidence until reconciled with allocation and journaling evidence [13, 14].

Commercial and open-source tools (e.g., EnCase, FTK, Autopsy/The SleuthKit) automate much of this work [12], but manual inspection of file system metadata is often required to interpret complex cases. Through file system forensics, examiners can detect malware placed on disk, identify exfiltrated data, and correlate file events with user or attacker actions [12].

2.4.2 Memory forensics

Memory (RAM) forensics analyzes the contents of a system's volatile memory image [17]. Since RAM holds running processes, open network connections, decrypted data, and other transient artifacts, memory analysis can reveal evidence not present on disk [17].

A principled approach involves validating structural invariants across multiple object graphs—such as process lists, Virtual Address Descriptors (VADs), thread tables, and handle references—to identify inconsistencies suggestive of code injection or process hollowing [18, 19]. Expected operating system semantics (e.g., coherence between Process Environment Block (PEB) and Thread Environment Block (TEB) structures, correct loader-linked module lists, and valid thread start addresses mapped to legitimate binaries) provide baselines for anomaly detection. Deviations prompt deeper validation, such as entropy assessments or Portable Executable (PE) header checks [18, 19]. Networking artifacts (e.g., sockets, TLS structures) are further correlated to their owning processes, providing attribution for communications without conflating discovery with interpretation [20].

Investigators capture a memory dump from a live system and then use frameworks such as Volatility to parse it. Volatility, one of the most widely used memory forensics platforms [21], provides plugins to extract process lists, DLLs, network sockets, registry hives, plaintext credentials, and more. For instance, malware that runs only in RAM (*fileless malware*) can be detected by scanning memory for known signatures or unusual code injections. By examining memory

artifacts, analysts can uncover the state of the system at the time of compromise, including processes spawned by attackers or remnants of encrypted communication. Overall, memory forensics is essential for capturing the "live" view of the system during an incident [17] [21].

2.4.3 Network forensics

Network forensics involves capturing and analyzing network traffic and logs to trace an attack's path [22]. It focuses on data in transit, such as packet captures (PCAP) from network taps, NetFlow records, router/firewall logs, and IDS/IPS alerts. By reconstructing TCP/UDP sessions, analysts can follow communications between infected hosts and external servers, revealing command-and-control channels, data exfiltration or lateral movements.

Network forensic analysis reconstructs communication flows and validates their semantics against expected service and protocol models. Initial triage often occurs at the flow level, where anomalous features such as beaconing patterns, asymmetric byte transfer, or baseline deviations identify sessions of interest [13, 23]. Session reassembly and protocol parsing refine this process, validating header–payload alignment, TLS handshake consistency, Server Name Indication (SNI) fields, Application-Layer Protocol Negotiation (ALPN), and timing coherence. DNS traffic is often analyzed as a control plane, where characteristics such as unusual top-level domains (TLDs), elevated `NXDOMAIN` response rates, or domain generation algorithm (DGA)-like queries guide the prioritization of inspection before conventional indicator-of-compromise correlation [24]. For exfiltration detection, analysts compare byte directionality and burst structures against the expected behavior of host roles, validate compression or chunking signatures, and investigate covert channels through entropy measurements and framing irregularities.

It enables responders to identify compromised devices on the network, understand the scope of a breach, and implement containment measures. In summary, network forensic techniques focus on traffic capture and analysis to uncover evidence of intrusion across the network layer [22].

In practice, a streamlined and reproducible toolchain is recommended. Evidence may be acquired via packet capture (PCAP) or structured flow logs and analyzed with a protocol parser such as Zeek, which generates structured summaries for connections, DNS, TLS, and HTTP traffic [25]. Wireshark is typically reserved for deep inspection of anomalous flows, while external threat intelligence services are queried to validate destination reputations. To ensure rigor, queries must be timestamped, and provenance preserved for later cross-correlation across forensic domains [26, 27].

2.5 Automation and AI in Digital Forensics

2.5.1 Existing tools

A number of forensic tools incorporate automation to streamline analysis. Suites such as *EnCase*, *FTK*, *Autopsy/SleuthKit*, and various commercial alternatives can automatically parse file systems, carve known file types, index keywords, and generate timelines [12]. Specialized tools also exist for memory analysis (e.g., Volatility) and network traffic examination (e.g., network flow analyzers, deep packet inspection tools) that automate specific tasks.

Despite these capabilities, human expertise remains essential. As one expert observes, "existing commercial digital forensics tools support automation to some extent, but there is still a need for a human expert" [28]. In practice, many time-consuming subtasks — such as validating matches, tuning extraction rules, or interpreting ambiguous artifacts — still require manual review.

Nonetheless, current tools greatly accelerate forensic workflows by handling routine tasks, allowing analysts to focus on deeper investigative analysis.

2.5.2 Potential and limitations of AI

Artificial Intelligence (AI) and Machine Learning (ML) promise to further augment forensic analysis. Because digital investigations often involve massive data volumes, AI can assist by automatically sifting through logs, disk images, and memory dumps to flag anomalies. For instance, ML classifiers can be trained to recognize malware patterns or suspicious user behavior; anomaly detection algorithms can highlight unusual network traffic profiles; and deep learning models may assist in triaging images or audio for relevant content [28]. In essence, AI excels at repetitive, high-volume tasks: it can rapidly "filter the haystack" and surface potentially relevant evidence for human review [28].

However, the integration of AI into forensic workflows faces significant challenges. Machine-learned models can inherit biases from their training data, leading to false positives or false negatives if the data is not representative [28]. Many AI algorithms, particularly deep learning, operate as opaque "black boxes," making it difficult to explain why a particular result was flagged. In forensic contexts, this is problematic because findings must be reproducible and defensible in court. As the literature notes, evidence derived from AI must be transparent: "in forensics, where findings may be used in court... having transparent and auditable decision-making... is a must" [28]. Accordingly, research emphasizes explainable AI (XAI) and human-in-the-loop approaches, where analysts guide or validate automated suggestions [28].

In summary, current AI tools can enhance efficiency (e.g., automatic triage of images, clustering of related artifacts, pattern recognition) [28]. However, they are not a panacea: experts must still interpret results and make final judgments. The state of the art is therefore a hybrid model, leveraging AI to handle scale and speed while relying on human analysts for insight and verification [28]. Future progress in AI forensics will focus on building robust, explainable models and curated training datasets, but the need for expert oversight will remain critical.

2.6 State of the Art in Malware Forensics and IoC Identification

This section reviews the state of the art in digital forensic examination of malware with a focus on the identification of Indicators of Compromise (IoCs) across Windows and Linux environments, emphasizing persistence mechanisms, their observable artifacts, and methodological approaches that enable reliable detection and repeatable investigations. The discussion synthesizes persistence taxonomies, detection and mitigation workflows, and operating-system–specific auto-start extensibility points, connecting technique-to-artifact mappings that inform automation and AI-driven correlation in subsequent chapters.

2.6.1 Foundations of IoC-Centric Forensics

A forensically robust IoC is a stable, attributable artifact that reliably signals malicious state transitions, including modified keys, created services, scheduled jobs, or hijacked components that persist across reboots and user sessions in Windows and Linux systems. Literature on persistence catalogs repeatedly shows that durable malware presence depends on auto-start extensibility points and privileged configuration surfaces whose changes can be enumerated and baselined to identify drift and malicious anchoring of code execution paths.

Methodologically, effective IoC discovery blends static topology of likely persistence locations with dynamic observation of installation-time mutations in registries, service control managers, scheduler stores, and user profile scripts to link cause–effect chains that survive system lifecycle events. Tool-supported workflows leverage well-known enumerators and monitors to detect anomalous entries, locations, and parent–child execution relationships that betray covert loading of malware at boot or logon boundaries.

2.6.2 Windows: Persistence Techniques and IoCs

Windows malware persistence literature converges on a core set of techniques that dominate field observations, providing a structured map from adversary methods to forensic artifacts and operational detectors. A comprehensive review[29] enumerates ten high-prevalence techniques: Run/RunOnce keys, Startup folders, Scheduled Tasks, Winlogon helper abuse, IFEO hijacking, Accessibility program swaps, AppInit DLLs, DLL Search Order Hijacking, COM hijacking, and Windows Services—each with characteristic registry paths, files, and event channels that yield actionable IoCs.

Boot/Logon Auto-Start

"**Registry Run/RunOnce Key (RRK)** is a typical sub-technique in the Boot or Logon Autostart Execution group. According to our assessment, this is malware's most used persistence technique. It is easy to implement, and malware exploits the run registry key for privilege elevation."[29] Run and RunOnce keys are frequently abused to invoke payloads per-user or machine-wide, rendering registry value creation in HKCU/HKLM (Hive Keys Local Machine/Current User) hive paths a primary IoC that can be baselined and continuously audited with autorun enumerators and change monitors.

The following run keys are created by default in Windows 32-bit version:[29]

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce
```

If the application is 32-bit but Windows is 64-bit, the run keys are created by default in:[29]

```
HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Run
HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\RunOnce
HKEY_CURRENT_USER\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\Run
HKEY_CURRENT_USER\Software\Wow6432Node\Microsoft\Windows\CurrentVersion\RunOnce
```

The difference between the Run and RunOnce is that "the RunOnce keys will run the programs when Windows boots the next time only, and then the entries will be deleted and not executed again."[29] The malware uses this for persistance by creating a subkey with a name of the attacker choice and then a value that represents the full path of the malicious payload to execute on startup. This is an example of a registry hive of a machine infected with the LokiBot malware as can be seen in figure 2.2:

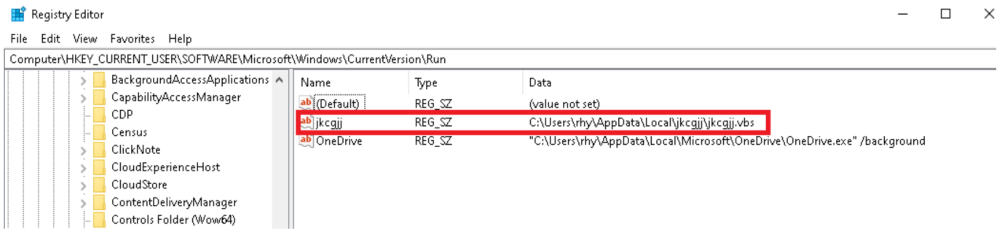

Figure 2.2. LokiBot infected hive example [29]

For a forensic analyzer point of view, this hives are present in the following files:

- HKCU entries live in the user's NTUSER.DAT hive.
  (e.g '\Users\username\NTUSER.DAT')

- HKLM entries live in the SYSTEM/ SOFTWARE machine hives. (e.g '\Windows\System32\SOFTWARE or \SYSTEM')

"**The Startup folder** is a feature available in Windows OS that enables a user to run a specified set of programs automatically when Windows starts. This folder aims to provide convenience for the users to gain instant access to the programs used frequently."[29] It provides an equivalent logon persistence as the hive keys, with filesystem shortcuts and executables whose placement and naming provide file path IoCs and timeline anchors correlated.

"Typically, malware downloads payloads to a specific hidden location (e.g., AppData), then builds a shortcut to those files and places them in the Startup Folder. This way, when Windows checks the Startup Folder, it will launch the malware's payload. Similar results can be achieved if malware copies itself to the Startup Folder."[29]

The cited folders are the following:

```
User-specific:
C:\Users\[Username]\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup

System-wide:
C:\ProgramData\Microsoft\Windows\Start Menu\Programs\StartUp
```

An example of infected StartUp folder is showed in figure 2.3

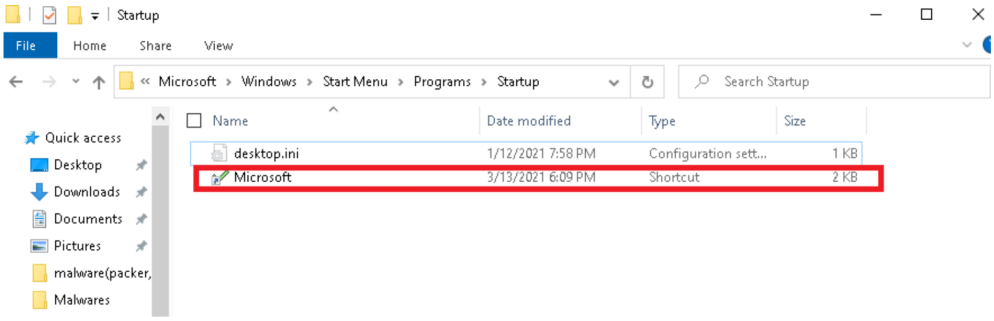

Figure 2.3. Infected Startup folder example [29]

"**Winlogon** or Winlogon.exe is a legitimate process that takes care of user logons, assigns security to the user shell, and loads user profiles into the registry. It can also be used to run the DLLs or executable files when a user logs in."[29] Winlogon helper abuse targets Shell, Userinit, and Notify parameters under Winlogon keys, introducing DLL or EXE paths that start at logon; modifications to these values and unexpected binaries in system directories constitute high-fidelity IoCs.

The programs that Winlogon launches are placed under the following registry keys:

```
32-bit:
HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Winlogon

64-bit:
HKLM\Software[\Wow6432Node\]\Microsoft\Windows NT\CurrentVersion\Winlogon
```

Attackers can modify these registry keys and add the path to the malware, which the Winlogon process will initiate. The subkeys which are commonly utilized by malware in this persistence technique are:[29]

- `Winlogon\Notify` refers to notification package DLLs responsible for handling Winlogon events. The Dipsind malware family and Bazar malware have registered as a Winlogon Event Notify DLL to achieve persistence.[29]

- `Winlogon\Userinit` refers to userinit.exe – when a user signs in, the user initialization software is performed. Some malware using this type of subkey are Wizard Siper and Remexi.[29]

- `Winlogon\Shell` refers to explorer.exe. The system shell is executed when a user logs on. By setting the value "Shell" with "explorer.exe, `%malware_pathfile%`" under the registry key, the Gazer backdoor can maintain its existence and repeat malicious activities on the victim system. Tropic Trooper and Turla malware also implement a similar technique.[29]

An example of infected hive registry can be seen in figure 2.4

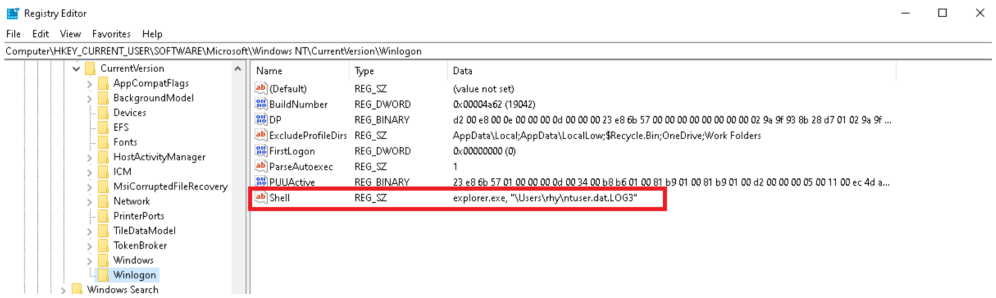

Figure 2.4. Infected Winlogon registry hive example [29]

Scheduled Execution

"Another way to perform persistence is to take advantage of the scheduling functionality when executing malware or for repeated malware execution. This technique is also called **Scheduled Task (or Job)**. Attackers can configure to execute the malware at a specified time or during system startup via Windows utilities such as schtasks or at."[29] Scheduled Task/Job persistence uses schtasks.exe or at.exe to trigger repeated execution on boot, logon, or intervals, producing IoCs in Task Scheduler definitions, registry-backed stores for legacy tooling, and operational logs that record task creation, modification, and invocation. Forensics emphasize reviewing task names, principals, actions/commands, and parent process ancestry to detect anomalous entries and lateralized remote scheduling, yielding both configuration and process lineage IoCs.

In a forensic analysis point of view, the folders and hives to look for this kind of IoCs are the following:

- **SOFTWARE hive file**
  Path on disk: `C:\Windows\System32\Config\SOFTWARE`
  It Contains:
  - `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tasks`
  - `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tree`

  which are the authoritative stores where Windows loads tasks from (triggers, actions, principals, GUID mapping).

- **Tasks directory (XML task files)**
  Path on disk: `C:\Windows\System32\Tasks\` (and subfolders)
  It Holds the Tasks XML definitions on disk.

An example infected folder and task would look like the ones in figure 2.5 and 2.6

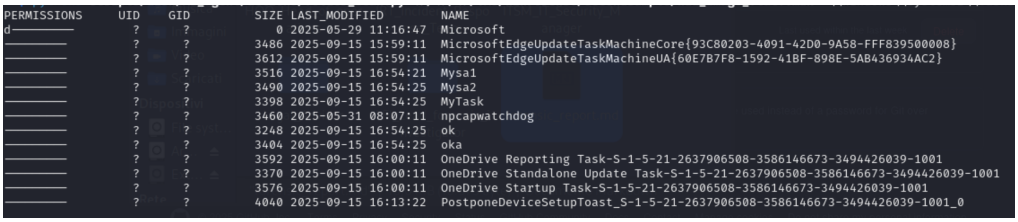

Figure 2.5. cotzhost infected task folder example

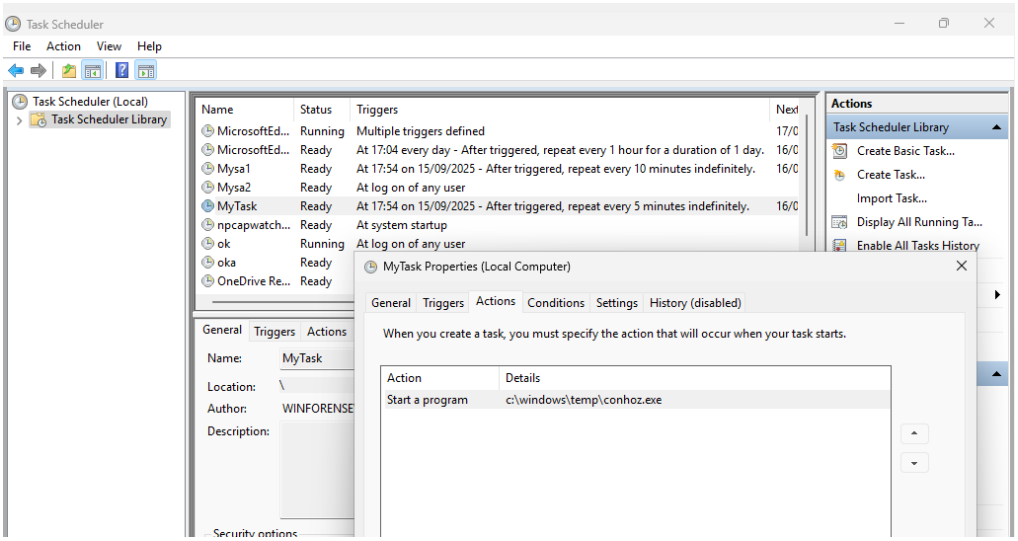

Figure 2.6. cotzhost infected task example

Execution Flow Hijacking

"**Image File Execution Options** (IFEO) is a Windows registry key, a sub-technique of the Event- Triggered Execution group. It means that the attackers may establish persistence and/or elevate privileges using system techniques that trigger execution based on specific events."[29] Image File Execution Options establish debuggers for target executables, enabling privilege escalation and persistence by redirecting application launches; registry Debugger values under IFEO keys form precise IoCs that are readily audited and rarely benign outside developer contexts. IFEO is placed at:

```
32-bit:
HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Image File Execution
Options

64-bit:
HKLM\SOFTWARE[\Wow6432Node]\Microsoft\Windows NT\CurrentVersion\Image File
Execution Options
```

The malware only needs to create a subkey as follows to make this configuration:

```
Key: HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Image File Execution
Options\notepad.exe
Value for debugger: REG_SZ: <malware_full_path>
```

an example of this is provided in figure 2.7 where a dummy payload is used (calc.exe).

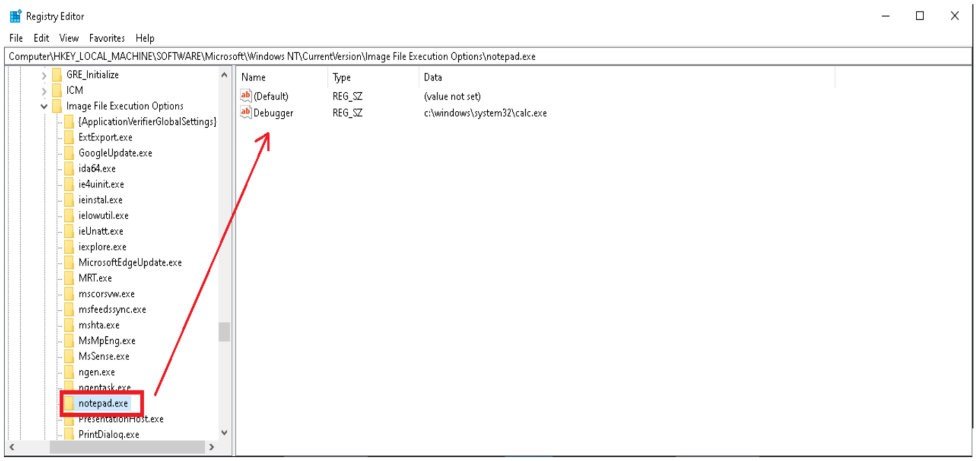

Figure 2.7. IFEO example [29]

If we start the notepad.exe process, we accidentally enable the malware payload even though it is not a debugger. That is the way malware establishes persistence by IFEO.[29]

"**DLL Search Order Hijacking** is a persistence technique in the Hijack Execution Flow group. The attackers execute malware or harmful payload by manipulating the way operating systems run applications. When a program is launched on the system, it loads several DLLs libraries into the memory space of its process. Windows searches these files in the predefined locations in specific search sequences. By taking advantage of this principle, malware will hijack the search order to maintain its activities on the system. If an application does not specify where to load a DLL from, Windows will load a specific DLL in the following order:"[29]

- Windows verifies if the DLL has previously been loaded into memory. If yes, Windows uses that DLL; if not, Windows will check if the DLL is listed in the KnownDLLs registry key (Figure 19), which speeds up system DLLs loading. If the DLL is loaded in the list of KnownDLLs, the system uses its copy of the KnownDLLs (and the KnownDLLs' dependent DLLs, if any) instead of searching for the DLL.
- `HKEY_LOCAL_MACHINE\SYSTEM\ CurrentControlSet\Control\Session Manager\KnownDLLs`
- The directory the calling process was loaded from.
- The system directory: `C:\Windows\System32`.
- The 16-bit system directory: `C:\Windows\System`.
- The Windows directory: `C:\Windows`.
- The current directory.
- The directories that are listed in the `PATH` environment variable.[29]

So DLL Search Order Hijacking places malicious DLLs earlier in lookup sequences, creating IoCs as unexpected DLL loads from user-writable locations, current working directories, or non-standard paths, detectable via module inventories and SafeDllSearchMode enforcement checks.

Attackers may put a malicious DLL in a directory, which will be searched before the location of a legitimate library is utilized. It causes Windows to load the attacker's malicious library whenever the victim process requests it.[29] To trace this persistence technique, we need to carry out process analysis and file system analysis to monitor the creation, replacing, renaming, or deletion of DLLs on the system. Besides, it is needed to monitor the DLLs loaded from abnormal directories.[29]

To trace this persistence technique, we need to carry out process analysis and file system analysis to monitor the creation, replacing, renaming, or deletion of DLLs on the system. Besides, it is needed to monitor the DLLs loaded from abnormal directories.[29]

In a forensic analyst poit of view, we need to check the registry hive to search for suspicious dlls.

Example of a clean KnownDLLs hive registry can be found on figure 2.8

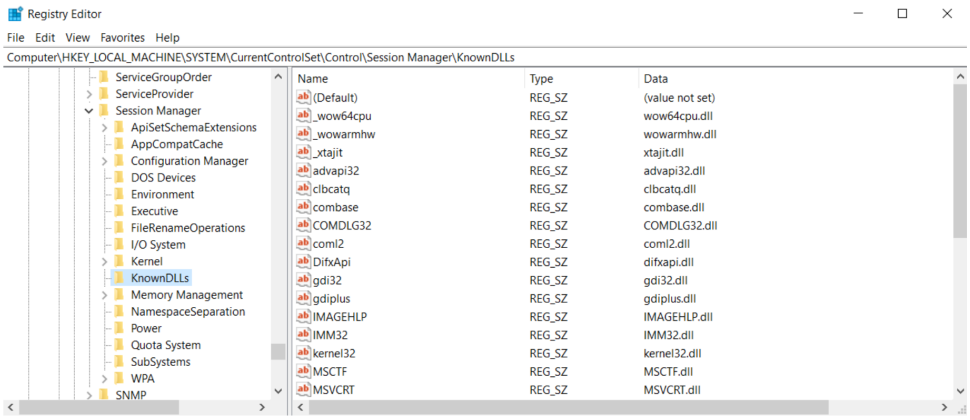

Figure 2.8. Clean KnownDLLs hive registry example [29]

"**AppInit DLLs** is a mechanism that allows custom DLLs to be loaded into the address space of every interactive application. Both legitimate softwares and malware use AppInit DLLs with the same purpose: to hook system APIs and implement alternate functionality."[29] AppInit DLLs abuse forces loading of attacker DLLs via user32.dll across interactive processes, making the AppInit_DLLs and LoadAppInitDLLs registry values, along with the presence of unknown DLLs in system directories, strong IoCs particularly on legacy or insecure-boot configurations.

The configuration values specify AppInit DLLs' operation in the registry key below. The malware only needs to use a batch script or through calls APIs like RegSetValueEx to set "LoadAppInit_DLLs" to value "1" to enable this technique.[29]

An example can be seen in figure 2.9.

```
32-bit:
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows

64-bit:
HKEY_LOCAL_MACHINE\Software[\Wow6432Node]\Microsoft\Windows
NT\CurrentVersion\Windows
```

"**Component Object Model (COM)** is a binary-interface standard working as a client-server model to implement objects used by different technologies and frameworks such as DCOM, OLE, OLE Automation, ActiveX, Browser Helper Object, COM+, Windows Shell, and Windows Runtime."[29] Component Object Model hijacking replaces InProcServer32 paths for CLSIDs, yielding registry redirections to untrusted locations and stealthy load-on-use persistence; per-user HKCU COM entries superseding HKLM defaults are especially rich IoCs. COM works by defining objects through interfaces that are independent of the language or framework used, ensuring interoperability across applications. When a COM client requests a service, the COM infrastructure locates the corresponding server and loads its implementation (often a DLL) into memory, enabling seamless communication. This mechanism, while powerful, also provides attackers with a pathway to abuse COM objects for persistence or malicious code execution. "The principle of COM is straightforward. Every time a program (COM client) runs on the system and uses the service provided by COM Server (or COM Object), it will download the associated DLL into the

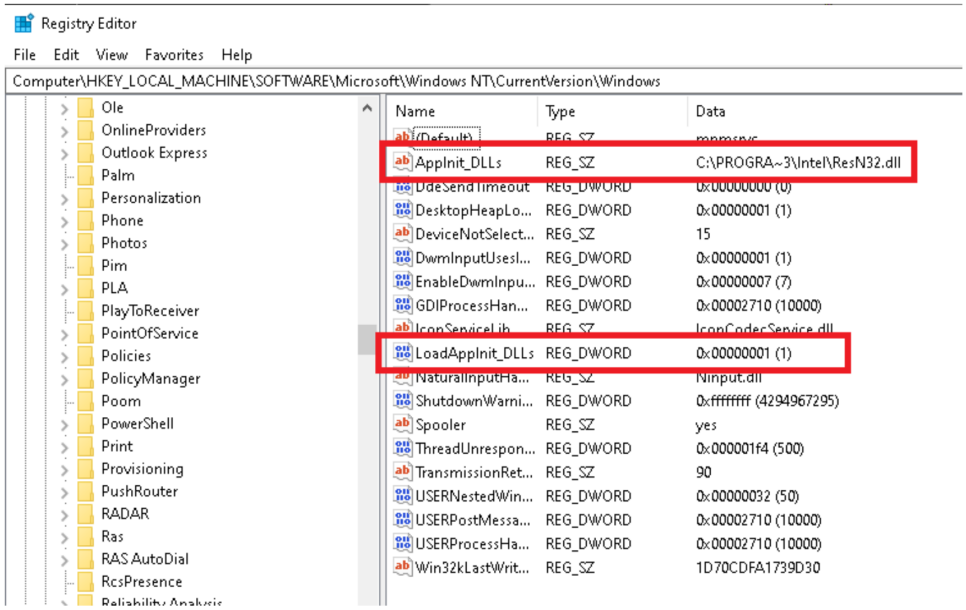

Figure 2.9. initDLL infected hive registry example [29]

process. It creates the opportunity for the attackers to take advantage of COM Object to perform persistence."[29] It should be noted that the COM server can be in the form of the in-process server (DLL file) or out-of-process server (EXE file).[29] A COM object is determined by a unique number named Class Identifier (CLSID) in:

```
HKEY_CURRENT_USER\Software\Classes\CLSID
HKEY_LOCAL_MACHINE\Software\Classes\CLSID
```

An example of infencted COM hive registry can be seen in figure 2.10.

Figure 2.10. infected COM hive registry example [29]

System Process Abuse

"**Windows Service** is a sub-technique in Create or Modify System Process group. The malware will be executed repeatedly by creating or modifying system-level processes. These processes are referred to as services and run in the background. It is the reason why this persistence technique is called Windows Service."[29] Windows Service persistence creates or modifies services to auto-start malware under SYSTEM, with IoCs including new service entries, anomalous binPath values, service types mapping to EXE/DLL/driver, and suspicious invocations of sc.exe, regsvr32.exe, PowerShell, or WMI during service creation. Literature underscores cross-view consistency checks between registry service configuration, service control

manager snapshots, and process trees to catch masqueraded or hijacked services. On Windows, the installed services and their configuration are stored in the following registry key: `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services`[29]

The attackers have many methods to create services on Windows:

- **Using `sc` utility:** malware uses `cmd.exe` to execute commands such as `sc create` (creating a service) and `sc start` (starting a service). For example, to create a service with the name `malwar3` for the file `malware.exe` located in `C:\Temp\` and launch it automatically:

  ```
  sc create malwar3 binpath=C:\Temp\malware.exe start=auto && sc start malwar3
  ```

  The parameter `binpath` specifies the payload path, and `auto` ensures the service starts automatically.

- **Using `regsvr32.exe`:** to register services with the following command:

  ```
  regsvr32.exe <path_to_service_dll>
  ```

- **Using batch script:** malware executes a specific `.bat` file downloaded from a Command & Control server, which invokes `sc` to create and start the malicious service.

- **Using Windows API functions:** malware may call functions such as `CreateService()` to create a service and `StartService()` to start it. Modern malware often includes these functions in its Import DLL table.

- **Using PowerShell:** malware can create a service with:

  ```
  New-Service -Name "malware" -BinaryPathName "C:\Temp\malware.exe" -Description
  "DemoWinService" -StartupType Automatic
  ```

  and then start it with the command `sc start malware`.

- **Using Windows Management Instrumentation (WMI):** WMI scripts can automate administrative tasks. Attackers may use `wmic` to create services or establish persistence.

Attackers can also modify or hijack existing, unused, or disabled services to reduce detection. Malware may embed its payload or load another malicious file from a remote server and register it as a secondary service. Some modern malware combines this technique with the Masquerading technique to make payloads appear legitimate.[29]

After registering as a service, the malware runs under the `services.exe` process, either directly if executable, or via `svchost.exe`. A listing of services launched under `svchost.exe` can be found here: `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Svchost`[29]

An example of services hives are in figures 2.11 and 2.12.

Representative Detection Tools and Workflows

Canonical detection tooling includes **Sysinternals Autoruns** for auto-start extensibility enumeration, **Process Monitor** for registry and filesystem change auditing, **Regshot** for differential hive snapshots, and timeline/process graphers that reconstruct installation-to-persistence sequences as evidentiary narratives. Methodologically, analysts correlate modified keys, file drops in hidden or profile paths, and scheduler/service entries with execution artifacts to build IoC sets that drive triage, hunt queries, and automated reporting.[29]

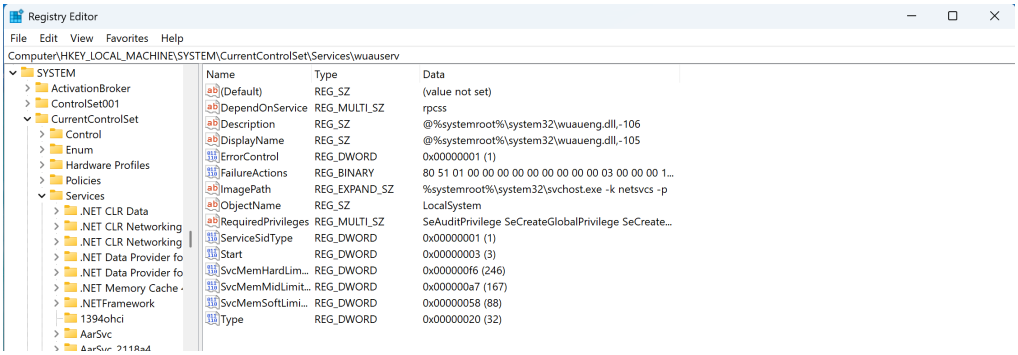

Figure 2.11. windows services hive registry

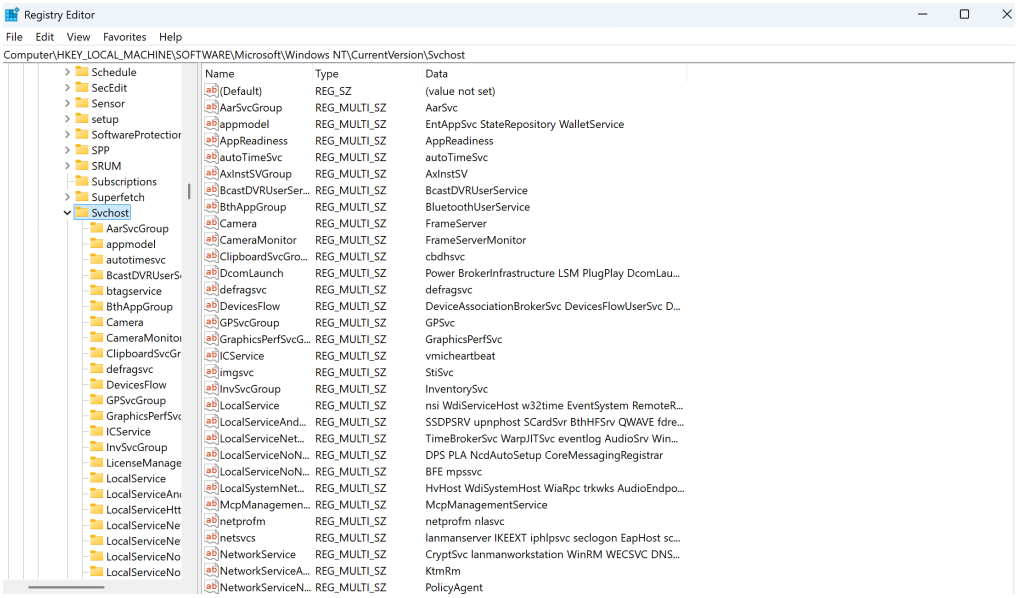

Figure 2.12. services that run under svchost

2.6.3 Linux: Persistence Techniques and IoCs

Linux persistence literature emphasizes account manipulation, SSH key-based access, scheduled execution via cron and systemd timers, shell initialization backdoors, dynamic linker abuse, and lower-level boot and module vectors, each mapping to distinct files, directories, and configuration surfaces that serve as IoCs. Contemporary surveys broaden beyond cron to systemd unit orchestration, XDG autostart in desktop contexts, and package and development tool hooks that convert routine administrative actions into covert execution.

Identity and Access Vectors

If an attacker successfully compromises a host system, one common technique to maintain persistence is through the **creation of a new user account**[30]. This approach leverages legitimate administrative utilities to minimize detection. For example, a new user with a corresponding home directory in `/home/username` can be created using the following command:

```
# Create a user account with a home directory in /home/username
sudo useradd -m <username>
```

```
# Set a password for the created account to enable login
sudo passwd <username>
```

On most Linux distributions, the root account exists by default but may be disabled or lack an assigned password. An attacker can activate the root account by assigning it a password, achieved via the following command:

```
sudo passwd
```

While enabling the root account grants direct administrative access, another persistence mechanism involves creating a new user and assigning them administrative privileges by adding the account to the `sudoers` group. This can be done as follows:

```
sudo usermod -aG sudo <username>
```

Through these methods, adversaries can establish persistent administrative access on a compromised Linux host, thereby facilitating continued control and privilege escalation.

User and root account creation or privilege assignment through sudoers modifications produce IoCs in passwd, shadow, group, and sudoers files, as well as in audit trails of useradd, passwd, and usermod invocations that indicate persistence provisioning.

From a forensic analyst point of view, the main location to look for this kind of IoCs in disk are the following:

- **/etc/passwd, /etc/shadow, /etc/group, /etc/gshadow** — these files record local accounts, hashed credentials references and group membership. Compare current copies with backups (`/etc/passwd-`, `/etc/shadow-`) to spot recent additions or modifications; check file timestamps (mtime, ctime) and entry UIDs/UID allocation gaps that often reveal ad-hoc accounts.
- **/etc/sudoers** and **/etc/sudoers.d/** — explicit sudo privileges and drop-in sudoers files. Malicious actors sometimes add a file to `/etc/sudoers.d/`; inspect for unexpected entries and incorrect file mode (should be 0440).
- **/home/[username]/** and **/root/** — newly-created home directories Check ownership, timestamps and shell history files (e.g. `.bash_history`).
- **/var/log/auth.log** (Debian/Ubuntu), **/var/log/secure** (RHEL/CentOS) and systemd journal — authentication events, sudo invocations, passwd changes, and useradd/usermod activity. Use `ausearch`/`aureport` against `/var/log/audit/audit.log` (if auditd enabled) to find syscall-level evidence of `useradd`, `passwd`, `usermod` and `chpasswd` calls.
- **/var/log/audit/audit.log** — if the Linux Audit framework is enabled this contains execve, setxattr, and other syscall records that can link processes to account creation or file modifications.
- **/var/log/faillog**, **/var/log/wtmp**, **/var/log/btmp**, **/var/log/lastlog** — interactive login records and failed login attempts; can show usage of newly-created accounts.

For maintaining persistence access to the machine, an adversary could also modify the related file `authorized_keys` file to maintain persistence on the victim host.[30] **SSH authorized_keys tampering** is a prevalent persistence method; unexpected keys, unusual comments, and anomalous file ownership or timestamps in user and root .ssh directories are crisp IoCs.

To verify whether unauthorized access keys have been introduced, the forensic analyst should inspect the contents of the `/home/[username]/.ssh/authorized_keys` file for newly appended or suspicious SSH keys.

Scheduled Execution

"Persistence access to the machine can be done by creating several specific scheduled tasks, there are usually two ways of creating the scheduled tasks: 'cronjobs modification/creaton' or 'malicious timer modification/creation'"[30] Cron-based persistence can be established in system-wide crontabs, cron.d drop-ins, and individual user crontabs. Indicators include newly added entries, jobs running as root that execute untrusted scripts, or unusual schedules used as opportunistic beacons. These modifications typically appear in specific cron configuration and spool locations such as:

- **/etc/crontab** — system-wide crontab file; inspect for new or suspicious entries.
- **/etc/cron.d/*** — drop-in cron configuration files; check for unexpected scripts or jobs.
- **/etc/cron.hourly/**, **/etc/cron.daily/**, **/etc/cron.weekly/**, **/etc/cron.monthly/** — scheduled task directories; review for unauthorized scripts.
- **/var/spool/cron/crontabs/*** — per-user crontab files; look for jobs running as root or unusual users.

Inspect these files for unexpected entries, suspicious command lines, or recent timestamps that align with other signs of compromise.

An example of cron file can be seen in figure 2.13. An attacker could just add to that file the line:

```
*/5 * * * * /opt/backdoor.sh
```

to run the backdoor script every 5 minutes.[30]

```
┌──(kali㉿kaliCTF)-[~]
└─$ cat /etc/cron.d/sysstat
# The first element of the path is a directory where the debian-sa1
# script is located
PATH=/usr/libexec/sysstat:/usr/sbin:/usr/sbin:/usr/bin:/sbin:/bin

# Activity reports every 10 minutes everyday
5-55/10 * * * * root command -v debian-sa1 > /dev/null && debian-sa1 1 1

# Additional run at 23:59 to rotate the statistics file
59 23 * * * root command -v debian-sa1 > /dev/null && debian-sa1 60 2
```

Figure 2.13. example of cron file

"all the services within the linux machine are triggered on boot time , they have specific init entry in boot process. but these services can be triggered at specific times also using 'timers'." [30] **Systemd timers** and **services** provide modular, stealthier persistence; new units in /etc/systemd/system with ExecStart pointing to non-standard scripts and periodic OnUnitActiveSec triggers are prime IoCs supplemented by journald records. [30]

In modern Linux distributions, **systemd** serves as the primary init system and service manager, responsible for initializing the user space and managing system services [31] [32]. While systemd provides a robust framework for legitimate service management, its flexibility can be exploited by adversaries to establish persistent footholds on compromised systems [33]. Forensic analysts must therefore be aware of all possible locations and mechanisms through which services may persist.

Systemd service definitions are primarily stored as unit files with a `.service` extension. These files define the execution parameters of a service, including its start commands, dependencies, and restart policies. From a forensic standpoint, the relevant locations include [31, 34]:

- **System-wide service directories:** `/etc/systemd/system/` for custom or administrator-installed services, and `/lib/systemd/system/` or `/usr/lib/systemd/system/` for default OS services. Analysts should compare these files against known package lists to identify anomalies.
- **User-level service directories:** `/home/[username]/.config/systemd/user/` contains services that start upon user login, which can be abused to bypass system-wide detection mechanisms.
- **Override directories:** `/etc/systemd/system/<servicename>.service.d/` and the directory `/run/systemd/system/<servicename>.service.d/` allow modification of legitimate services through drop-in files.
- **Target-linked symlinks:** `/etc/systemd/system/multi-user.target.wants/` contains the symlinks indicating which services are configured to start automatically at boot.
- **Timers:** Systemd timers, located at `/etc/systemd/system/*.timer` and the directory `/home/[username]/.config/systemd/user/*.timer`, can schedule the execution of services similarly to cron jobs.
- **Runtime and volatile directories:** `/run/systemd/system/` and `/var/lib/systemd/` store temporary and runtime unit files, which may reveal transient activity even if the system is rebooted.

Forensic analysis should also include a careful review of all `[Service]` directives in unit files. Fields such as `ExecStart`, `ExecStartPre`, `ExecStartPost`, and `ExecReload` may point to scripts or binaries in non-standard locations (e.g., `/tmp`, `/var/tmp`, or hidden user directories). Any execution command not associated with known packages or administrative scripts warrants deeper investigation [33].

Systemd introduces a multi-layered persistence landscape. Analysts must consider system-wide and user-level service definitions, drop-in overrides, target-linked symlinks controlling automatic startup, timer units as alternative persistence mechanisms, runtime artifacts revealing transient activity, and the contents of `[Service]` directives for anomalous execution paths. A systematic approach covering these elements increases the likelihood of detecting both obvious and stealthy persistence mechanisms in Linux environments [31, 33].

The following table summarizes the most relevant systemd persistence locations and mechanisms that a forensic analyst should investigate:

Persistence Point	**Location**
System-wide services	/etc/systemd/system/
Default OS services	/lib/systemd/system/, /usr/lib/systemd/system/
User services	/.config/systemd/user/
Drop-in overrides	*/*.service.d/ e.g. /etc/systemd/system/sshd.service.d/
Target-linked symlinks	*/*.wants/ e.g. /usr/lib/systemd/system/graphical.target.wants/
Timers	*.timer e.g. /etc/systemd/system/backup.timer
Runtime/volatile	/run/systemd/system/, /var/lib/systemd/

Table 2.1. Summary of Systemd Persistence Mechanisms

An example of malicious .service could look like this:

```
[Unit]
Description=Bad service
[Service]
ExecStart=/opt/backdoor.sh
```

with the associated .timer :

```
[Unit]
Description=malicious timer
[Timer]
OnBootSec=5
OnUnitActiveSec=5m
[Install]
WantedBy=timers.target
```

Here ‘OnUnitActiveSec=5m‘: is how long to wait before triggering the service again , after every 5 minute the malicious service is triggered , and potentially give persistence access to the machine.[30]

Shell and Session Hooks

”The shell in linux is the most crucial part of its enviroment , but it does load with lots of other configuration files , that are executed whenever the shell start or ends”.[30] User shell startup files (.bashrc, .bash_profile, .profile) and system-wide profiles (/etc/profile, /etc/profile.d, /etc/bash.bashrc) can be backdoored to execute payloads at interactive session start, creating IoCs as appended commands, encoded one-liners, or suspicious sourcing of remote paths. Table 2.2 summarizes the most relevant Bash initialization and termination files that may be leveraged for persistence and thus should be carefully examined during forensic analysis.

File(s)	Description
`/etc/bash.bashrc`	System-wide file executed at the start of an interactive shell (e.g., tmux).
`/etc/bash_logout`	System-wide file executed when the shell terminates.
`~/.bashrc`	Widely exploited user-specific startup script executed at the start of a shell.
`~/.bash_profile`, `~/.bash_login`, `~/.profile`	User-specific login files; whichever is found first is executed.
`~/.bash_logout`	User-specific cleanup script executed when a shell session closes.
`/etc/profile`	System-wide file executed at the start of login shells.
`/etc/profile.d/*.sh`	All `.sh` files in this directory are executed at the start of login shells.

Table 2.2. Common Bash initialization and termination files relevant to persistence and forensic analysis.

An example of an infected .profile would look like this:

```
# if running bash
if [ -n "$BASH_VERSION" ]; then
# include .bashrc if it exists
if [ -f "$HOME/.bashrc" ]; then
. "$HOME/.bashrc"
fi
fi
chmod +x /opt/backdoor.sh
/opt/backdoor.sh
```

Message of the Day (MOTD) update scripts under /etc/update-motd.d can be weaponized to trigger code upon SSH logins, leaving IoCs in modified templates and reverse-shell constructs embedded in header scripts. ”MOTD stands for Message of The Day which is a message that

gets displayed to users when they SSH into the system. It's configured in the /etc/update-motd.d/ directory and threat actors can place arbitrary commands into any of the files listed there. Therefore for this article it can be used as a method of persistence and we get a reverse shell back whenever a user SSH to the system."[30] For example, adding a one liner:

```
bash -c 'bash -i >& /dev/tcp/192.168.1.132/1234 0>&1'
```

To the file '/etc/update-motd.d/00-header' will create backdoor for persistance on the system each time the MOTD is visualized.[30]

Loader and Linker Abuse

Dynamic linker hijacking via LD_PRELOAD or /etc/ld.so.preload injects attacker .so libraries into processes, with existence or modification of preload files and non-standard shared objects in world-readable locations forming high-value IoCs, especially when system-wide. "The LD_PRELOAD allow attackers to load shared object files into to process's address space prior to the program itself, thus potentially allowing the control over the execution flow. The LD_PRELOAD can be set by writing the '/etc/ld.so.preload' file or utilizing the 'LD_PRELOAD' environment variable."[30] By default both LD_PRELOAD variable and file /etc/ld.so.preload are not set, so the sole exixstance of those files is extremly suspicious and should be analyzed by a forensic analyst.

SUID misuse and binary wrapping introduce persistence via privileged helpers; IoCs include unexpected SUID bits on scripts or custom binaries, divergence from distribution baselines, and wrapped utilities that proxy to originals after malicious action.[30] The SUID bit set alone does not consist in a critical or definitive IoC, but shold be investigated.

Boot, Service, and Autostart Surfaces

Legacy rc.local execution allows boot-time persistence when present and executable, generating IoCs as appended commands near exit lines and mismatched permissions and owners. The file is located in /etc/rc.local, and an exaple of an infected one is the following:

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each
multiuser runlevel.
# Make sure that the script will "exit 0" on
success or any other
# value on error.
#
# In order to enable or disable this script just
change the execution
# bits.
#
# By default this script does nothing.
/path/to/your/malicious/script.sh
exit 0
```

System-level services created via systemd units with benign names but untrusted ExecStart targets, coupled with enablement flags and timer pairings, provide durable, low-noise persistence with configuration file and journal IoCs. "Using systemd services is a modern and efficient way to achieve persistence in Linux. systemd is the init system and service manager in most Linux distributions, responsible for bootstrapping the user space and managing system processes after booting. By creating a custom systemd service, you can ensure that specific applications

or scripts run automatically at system startup." [30] Malicious services are created in the file /etc/systemd/system/. An example is the following:

```
[Unit]
Description=My custom service
After=network.target
[Service]
Type=simple
ExecStart=/path/to/my_script.sh
Restart=on-abort
[Install]
WantedBy=multi-user.target
```

Developer and Package Tooling Hooks

APT hooks under /etc/apt/apt.conf.d enable before/after command execution during package operations, rendering arbitrary reverse-shell one-liners or script invocations in hook files potent IoCs during routine maintenance windows.

Git hooks (.git/hooks/*) and pager configuration in .gitconfig can embed on-commit or on-log execution paths, leaving IoCs as executable hook scripts and environment-driven pager commands that chain to shells.[30]

Desktop and Device Event Triggers

XDG autostart persists in GUI sessions via /.config/autostart/*.desktop entries pointing to scripts or interpreters, with unexpected desktop entries and hidden script targets serving as IoCs.

Udev rules in /etc/udev/rules.d that trigger on device attach can run attacker code, yielding IoCs as newly introduced rules with RUN directives tied to network or shell actions.[30]

Kernel and Low-Level Extensions

Loadable kernel modules present stealthy persistence; IoCs include unsigned or non-vendor modules, anomalous module paths, and insmod/modprobe histories that deviate from platform baselines.

Database triggers and environment manipulation represent higher-level persistence in application stacks, with IoCs discovered via schema inspections and environment variable overrides that redirect execution or load alternate libraries.[30]

2.6.4 Comparative Themes and Detection Methodologies

Cross-platform patterns reveal that high-utility persistence exploits configuration substrates that are both durable and routinely read by trusted components, producing IoCs that concentrate in registries and service catalogs on Windows and in init/service managers and profile scripts on Linux. Effective methodologies therefore prioritize continuous enumeration of auto-start points, integrity checks of configuration stores, and correlation of file drops with scheduled or service-based invocations to attribute code-loading pathways.[29][30]

Windows-centric workflows emphasize registry diffs, autorun inspections, task and service audits, and DLL/module inventories, while Linux-centric workflows emphasize cron and systemd audits, SSH key reviews, and shell/profile integrity checks, each forming the backbone of automated IoC harvesting pipelines. Across both systems, process lineage analysis linking configuration changes to parent processes like scripting engines, service managers, and schedulers strengthens attribution and reduces false positives in IoC-driven triage.[29][30]

Below there is a summary table of all the IoCs files location for a quick look, For windows refer to 2.3 For Linux refer to 2.4

Persistence technique	**Files / Registry hives / Locations to analyze**

Registry Run / RunOnce keys (RRK)	`HKCU:\Software\Microsoft\Windows\CurrentVersion\Run`
	
	`HKCU:\...\RunOnce`
	`HKLM:\Software\Microsoft\Windows\CurrentVersion\Run`
	`HKLM:\...\RunOnce`
	On-disk: `C:\Users\<user>\NTUSER.DAT` (HKCU) and `C:\Windows\System32\Config\SOFTWARE` (HKLM).

Startup Folder	User startup: `C:\Users\<User>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup` System startup: `C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup` Look for `.lnk` shortcuts, payloads in `AppData`, timestamps and alternate data streams.
Winlogon helper abuse (Shell / Userinit / Notify)	Registry keys: `HKLM:\Software\Microsoft\Windows NT\CurrentVersion\Winlogon` (also `...\Wow6432Node ...` on x64 for 32-bit entries). On-disk: `C:\Windows\System32`, `NTUSER.DAT` for per-user changes if applicable.
Scheduled Tasks (Task Scheduler)	On-disk task XMLs: `C:\Windows\System32\Tasks (and subfolders)` Registry/task cache: `C:\Windows\System32\Config\SOFTWARE` Keys: `HKLM:\Software\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache\Tasks` `...\TaskCache\Tree` (triggers, actions, GUIDs).
Image File Execution Options (IFEO)	Registry: `HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<exe>` On-disk: indicated debugger path binary locations; check `C:\Windows\System32` and payload path.
DLL Search Order Hijacking	Registry KnownDLLs: `HKLM:\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs` Inspect process module lists, unexpected DLLs loaded from non-system or user-writable directories; on-disk filesystem locations where the malicious DLL resides.
AppInit DLLs	Registry: `HKLM:\Software\Microsoft\Windows NT\CurrentVersion\Windows` Values: `AppInit_DLLs`, `LoadAppInit_DLLs`; examine system DLL directories and any unknown DLLs referenced.
COM / InProcServer32 hijacking	Registry CLSID / ProgID paths under `HKCR` and per-user equivalents (`HKCU\Software\Classes`); explicitly check `HKEY_CURRENT_USER\Software\Classes\CLSID` and `HKEY_LOCAL_MACHINE\Software\Classes\CLSID`. Check `InProcServer32` paths for unexpected locations and the corresponding on-disk DLLs.
Windows Services	Registry: `HKLM:\SYSTEM\CurrentControlSet\Services` Check `binPath` values, service type, Start value, and `Svchost` groupings: `HKLM:\Software\Microsoft\Windows NT\CurrentVersion\Svchost` On-disk: service executable paths (e.g., `C:\Temp\<payload>.exe`), driver files, and Service Control Manager snapshots.
Accessibility / Binary Replacement (e.g., utilman.exe, sethc.exe)	On-disk system binaries in `C:\Windows\System32` (compare file hashes, sizes, timestamps); check IFEO debugger overrides and shadow copies; examine `SYSTEM` / `SOFTWARE` hives for related modifications.

Table 2.3: Summary — Windows persistence techniques and primary files/locations to analyze (IoC sources).

Persistence Technique	Files / Locations to Analyze
Account Manipulation / User Creation	• /etc/passwd, /etc/shadow, /etc/group, /etc/gshadow • /etc/sudoers, /etc/sudoers.d/ • /home/[username]/, /root/ • /var/log/auth.log (Debian/Ubuntu), /var/log/secure (RHEL/CentOS) • /var/log/wtmp, /var/log/btmp, /var/log/lastlog, /var/log/faillog
SSH Key Persistence	• /home/[username]/.ssh/authorized_keys • /root/.ssh/authorized_keys
Scheduled Execution: Cron Jobs	• /etc/crontab • /etc/cron.d/* • /etc/cron.hourly/, /etc/cron.daily/, /etc/cron.weekly/, /etc/cron.monthly/ • /var/spool/cron/crontabs/*
Scheduled Execution: Systemd Timers / Services	• /etc/systemd/system/*.service, *.timer • /lib/systemd/system/, /usr/lib/systemd/system/ • /.config/systemd/user/ • /etc/systemd/system/¡service¿.service.d/ • /etc/systemd/system/*.wants/ • /run/systemd/system/, /var/lib/systemd/
Shell and Session Hooks	• ˜/.bashrc, ˜/.bash_profile, ˜/.profile, ˜/.bash_logout • /etc/bash.bashrc, /etc/profile, /etc/bash_logout • /etc/profile.d/*.sh • /etc/update-motd.d/*
Dynamic Linker / Loader Abuse	• /etc/ld.so.preload • Non-standard shared libraries (.so) in writable directories • SUID binaries: unexpected / anomalous permissions
Legacy Boot Persistence (rc.local)	• /etc/rc.local
Developer and Package Tooling Hooks	• /etc/apt/apt.conf.d/* • .git/hooks/* • ˜/.gitconfig
Desktop Autostart / Device Event Triggers	• ˜/.config/autostart/*.desktop • /etc/udev/rules.d/*.rules
Kernel and Low-Level Extensions	• /lib/modules/ (malicious kernel modules) • lsmod / /proc/modules (loaded module inspection) • Application-level DB triggers / env variables

Table 2.4: Linux Persistence Techniques and Associated Files/Locations for Forensic Analysis

2.6.5 Mitigation, Evasion, and Research Gaps

Mitigations in Windows include baselining auto-start extensibility points, enforcing SafeDllSearchMode, restricting IFEO usage, hardening accessibility binaries, and monitoring for anomalous service and scheduled task creation, each reflecting technique-specific control points surfaced in the literature. Linux mitigations prioritize key hygiene, principle of least privilege for sudoers, hardening of systemd units and timers, linker preload controls, and integrity monitoring of shell profiles and automation hooks to reduce stealth execution paths, with routine audits providing rapid IoC discovery.[30][29]

Adversary evasion leverages masquerading, per-user configuration precedence (e.g., HKCU over HKLM COM), living-off-the-land binaries, and legitimate scheduler/service interfaces to blend in, pushing research toward deeper baseline modeling and anomaly detection on configuration semantics rather than mere presence. Open gaps remain in scalable, cross-artifact correlation and in automated narrative generation that connects IoC sets to attacker TTPs, motivating the integration of automation and AI explored in later chapters of this thesis.[29][30]

Chapter 3

Methodology

3.1 Experimental laboratory setup

To test the automated framework against machines that were infected by real malware samples, I firstly needed a controlled environment to acquire infected artefact samples safely. To achieve this, I set up a virtual laboratory consisting of both Windows and Linux machines. By infecting these machines myself, I maintained full control over the chosen samples and the persistence mechanisms or artefacts they left behind.

3.1.1 Virtual Machine Configurations

The first step in my methodology was the selection of the operating systems. I chose Windows 11 and Ubuntu 22.04, as they are among the most widely adopted platforms in real-world environments. Another reason for this was that both are supported by the ANY.RUN sandbox, which I later employed to validate the results produced by the automated framework. In particular, this allowed me to cross-check the framework's findings against Indicators of Compromise (IoCs) derived from the same real malware samples executed within the same sandbox environment. To generate disk artefacts suitable for later analysis, I adopted the flat VMDK storage method in VirtualBox, which produces fixed-size disk images ready for forensic examination. This method creates fixed-size disk images, which are particularly well-suited for forensic examination because they preserve the disk structure in a consistent and uncompressed form. In contrast, dynamically allocated disks, while more space-efficient, can introduce fragmentation and unnecessarily make more complex the task of acquiring the disk artefact in my case.

Windows VM

For the Windows virtual machine, I used Oracle VirtualBox and installed an official Windows 11 image. The VM was configured in the following way:

- **NAT network mode:** Provides internet access to the virtual machine while isolating it from the host network, reducing the risk of accidental malware propagation.
- **BitLocker disabled:** Ensures full access to the disk for forensic analysis and imaging, preventing encryption from interfering with artifact collection.
- **Guest Additions enabled:** Improves VM usability and integration, allowing for easier file transfer and enhanced performance without affecting malware behavior.
- **Additional tools installed:**
  - **WinRAR:** Facilitates the extraction and handling of compressed files or malware samples delivered as archives.
  - **Wireshark:** Enables network traffic capture for analyzing malware communication and identifying IoCs.
  - **DumpIt:** Allows for memory acquisition to analyze malware behavior in volatile memory. The tool is available on GitHub: DumpIt GitHub link.

Also, to avoid interference during malware execution in the virtual machine box, i had to permanently disable Windows Defender. The following steps ensured Defender was disabled before capturing an initial snapshot of the VM:

1. Disable Tamper Protection.
2. From an elevated Command Prompt, run:

```
reg add "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender\Real-Time Protection" /v
    DisableRealtimeMonitoring /t REG_DWORD /d 1 /f
reg add "HKLM\SOFTWARE\Policies\Microsoft\Windows Defender" /v DisableAntiSpyware /t
    REG_DWORD /d 1 /f
```

This two commands set the correspondant hive registry keys to disable real-time monitoring and the main anti-spyware component of Windows Defender, effectively preventing the antivirus from interfering with subsequent malware execution or the generation of disk artefacts. By modifying these keys prior to capturing the initial VM snapshot, I ensured that the system state reflected a clean baseline without Defender actively monitoring or altering file system activity. similar to a zero-day attack scenario or an environment with outdated antivirus definitions. Without such a configuration, the malware samples I use, being previously discovered and analyzed, would be immediately detected as threats by the antivirus and removed, preventing meaningful analysis. Once configured, the machine was ready for controlled infection and artefact acquisition.

Linux VM

For the Linux virtual machine, I installed Ubuntu 22.04.5 (desktop, amd64) on VirtualBox. A really important challenge arose when using Volatility 3 for memory analysis, as it requires the exact kernel symbol table corresponding to the installed kernel version. In some cases, retrieving the correct symbols proved complex. Further details are provided in the next chapter in the memory analysis section.

This VM is configured with **NAT network mode** and has **Guest Additions enabled**, just like the Windows VM, for the same reasons: to provide internet access while isolating the VM from the host and to improve usability without affecting malware behavior. There is no need to disable any antivirus, as Ubuntu does not include one by default. On this VM, I installed the following tools:

- **LiME:** Linux Memory Extractor, widely used tool for acquiring volatile memory in Linux OS. Used to acquire the memory dump from the VM to analyze malware behavior in RAM. The tool is available on GitHub: LiME GitHub link.
- **7-Zip:** Utility for handling compressed files, enabling extraction of malware samples delivered as archives.
- **TShark:** Command-line version of Wireshark, used for capturing and analyzing network traffic generated by malware.

Once configured, the VM was ready for malware infection and analysis.

3.1.2 Infection Procedures with Samples from Any.Run

To infect the virtual machines, I obtained malware samples from the platform `app.any.run`. I specifically searched for reports containing persistence mechanisms by filtering for "malicious" samples with ATT&CK Matrix persistence techniques (e.g., T1060: Registry Run Keys / Startup Folder).

This choice reflects the focus of the study: scenarios where the victim has already been infected and rebooted several times, leaving traces in memory, disk, or network artefacts.

I tried to select types of malwares with different kind of persistance technique where possible. However, the selection was constrained by the availability of samples in ANY.RUN. The platform does not provide access to all historical malware, but only to those publicly uploaded within the last six months (as of March 2025, the time of writing this thesis).

All samples were downloaded in compressed archives (password: `infected`) without file extensions.

In the following section lies the list of the sample collected from anyrun.

Windows: Worm:Win32/Ludbaruma

Worm:Win32/Ludbaruma is an MPress-packed worm that prioritizes persistence and stealth over immediate payload execution. It hijacks the screensaver mechanism, creates autorun registry entries under both HKCU and HKLM (including Wow6432Node), and drops executables with system-like names in various directories. Persistence is reinforced through registry tampering that disables user options and registry tools, hindering manual cleanup. The worm also shows SMB activity indicative of lateral movement but no C2 traffic during short execution.

From a forensic perspective, the most relevant Indicators of Compromise (IoCs) are anomalous entries in the Run and RunOnce registry hives. These entries use benign-looking names (e.g., "MSMSGS," "Serviceadmin") that map to suspicious binaries in non-standard paths (e.g., C.exe, Data.EXE). Since such registry artifacts persist across reboots and are recoverable post-mortem, they provide strong and reliable evidence of infection.[35]

Windows: Conhoz Loader

The Conhoz loader is a UPX-packed 32-bit PE malware that uses a living-off-the-land (LOtL) approach to establish persistence, manipulate system defenses, and stage additional payloads via scripted tasks and HTTP beacons. It launches cmd.exe and u.exe and leverages Windows utilities—schtasks, sc, reg, wmic, netsh, attrib, and PowerShell—to persist, tamper with policies, and execute scripts.

Main Forensically relevant IoCs:

- **Scheduled tasks:** The malware creates tasks such as MyTask, oka, Mysa1, and Mysa2, running under the SYSTEM account on logon or at frequent intervals. These tasks provide reliable automatic execution across reboots and can be repeatedly recreated if deleted, making them resilient against cleanup attempts.
- **WMI and IFEO modifications:** Conhoz manipulates WMI consumers and filters, and configures IFEO "Debugger" redirections targeting binaries like 001.exe and 32/64.exe. This allows the malware to hijack legitimate process launches, further strengthening persistence and stealth.

Additional IoCs include runtime artifacts (C.exe, C.dat, C.bat) and transient IE cache files. Among these, scheduled task names and WMI/IFEO modifications are the most forensically significant, providing clear evidence of automated execution and loader activity. These IoCs allow investigators to identify Conhoz infections, track persistence mechanisms, and reconstruct its loader and staging behavior.[36]

Linux: Prometei

The Prometei-associated Linux ELF sample is a 64-bit, statically linked malware that targets x86-64 systems, performing cryptomining preparation and lateral movement. It runs with elevated privileges, modifies its executable's ownership and permissions, and ensures persistence via a systemd service (uplugplay.service). During execution, it collects system information, inspects running processes, and modifies system files to establish a durable foothold.

Forensically relevant IoCs include:

- **Systemd persistence:** Presence and active/enabled state of the system service located in the path `/usr/lib/systemd/system/uplugplay.service`.
- **Executable placement and permissions:** `/usr/sbin/uplugplay` with root ownership or broad execution rights.
- **Host profiling artifacts:** Temporary changes to `/etc/hosts` and outputs from system interrogation commands (`os-release`, `hostnamectl`, `uname`, `uptime`).
- **Network indicators:** Outbound HTTP requests to Prometei-linked domains/IPs, following the "xinchao" C2/DGA pattern.

These artifacts collectively indicate stealthy persistence, reconnaissance, and modular design for propagation, allowing investigators to trace Prometei infections before cryptomining activity occurs.[37]

Linux: Gh0st RAT

This 64-bit ELF sample achieves persistence on Linux by creating and enabling a systemd service named remote-client.service. It runs with elevated privileges via sudo, adjusts ownership and permissions, and uses systemctl daemon-reload, enable, and start to ensure the service executes automatically across reboots. Supporting artifacts include launchtime.txt, ~/logs/remote-client.log, and the service unit file /etc/systemd/system/remote-client.service, providing clear forensic evidence of malware presence.

Process execution follows a chain of dash, bash, chown, chmod, sudo, and systemctl, culminating in the RAT component running under systemd supervision. The malware exhibits Gh0st RAT behavior, performing environment reconnaissance (locale, timezone, network configuration) and connecting to C2 servers at a87.mee333.com:443 and lc.liuddiase1li.com:5650. Network activity is marked by repeated 200 responses and Go-http-client user agents, with sandbox whitelisting isolating malicious flows to Gh0st-linked hosts.

Forensically relevant IoCs include:

- **Service unit:** `/etc/systemd/system/remote-client.service`
- **Log file:** `~/logs/remote-client.log`
- **C2 domains/IPs:** `a87.mee333.com` / `103.215.78.27`, `lc.liuddiase1li.com` / `118.107.46.162`

These artifacts make systemd-based persistence the primary forensic indicator, allowing investigators to identify Gh0st RAT infections, trace initialization procedures, and reconstruct the malware's remote-control activity.[38]

3.1.3 Acquisition of Artefacts (HDI, RAM, PCAP)

After infecting the VMs, I manually acquired the artefacts generated by the infected machine. This included disk image, memory dumps (RAM), and packet captures (PCAP).

- **Disk Image:** Used the generated `vmdk-flat` file from VirtualBox VM directory. **Note:** If you want to convert a normal vmdk to a flat one, you can use the `VBoxManage` command-line tool provided with VirtualBox:

  ```
  VBoxManage.exe clonemedium disk "source.vmdk" "destination.vmdk" \
  --format VMDK --variant Fixed
  ```

- **RAM:**
  - **Linux:** Created a memory dump using LiME:

    ```
    sudo insmod lime-$(uname -r).ko "path=memdump.lime format=lime"
    ```

  - **Windows:** Used DumpIt by executing the provided binary.
- **PCAP:** Captured network traffic for 60 seconds with Wireshark or TShark. Done with the GUI with Wireshark or with the command line with TShark:

  ```
  sudo tshark -i any -a duration:60 -w net_traffic.pcap
  ```

All artifacts were collected after a full shutdown and reboot of the machine, simulating a real-world scenario in which the malware was allowed to persist in the system.

Chapter 4

Framework Implementation

4.1 Automated Framework and Processing Pipeline

After manually acquiring these artefacts, I developed a Python-based framework to process them automatically, integrating suitable libraries for parsing and analysis.

This chapter focuses on the practical implementation and internal functioning of the automated forensic framework. Unlike the methodology chapter, which discussed the rationale behind the chosen approaches, here we concentrate on how the framework operates in practice.

The following sections provide a detailed description of the framework's architecture, the individual modules that process and analyze forensic evidence, the workflows that guide its operation, and the outputs it produces. Through this chapter, the reader will gain a clear understanding of the framework's structure, capabilities, and behavior during forensic analysis.

The PoC framework, can be found in the following github repository: `https://github.com/fuju/ForensicAutonomousFramework`.

In figure 4.1 there is a call-graph describing the workflow and provides an high-view description of each script role in the framework.

A full overview on the framework project folder structure, can be seen in the appendix A.3.

The User Manual and the Programmer's Guide can be found in the appendix A.1 and A.2 respectively.

4.1.1 Framework Setup

To prepare the environment for the analysis framework, the following steps should be performed:

- **Install Python.** It is recommended to use Python 3.9 to ensure compatibility with all dependencies and scripts, as some components may not fully support more recent Python versions.
- **Create a virtual environment.** Using a virtual environment isolates the project dependencies from the system Python installation, avoiding potential version conflicts and ensuring reproducibility.
- **Install dependencies.** All required Python packages are listed in `requirements.txt`. Installing them ensures that the framework can execute all scripts and modules correctly.
- **Modify virtual environment paths if necessary.** By default, the virtual environment is located at `<framework_home>/venv`. If a different path is used, it must be updated accordingly in the main cofiguration file `config.py`.
- **Install Volatility3 symbol tables.** These tables are required for memory analysis. They can be installed either directly from the repository or by manually extracting them and converting via `dwarf2json`. More on this in section Section 4.2.2.
- **Optional fix for older python versions:** Run `patch_tshark.py` only if you chose to use a python version older that 3.7. This script fixes the errors related to the deprecated library "distutils".

After completing these steps, the environment is fully prepared and ready for analysis tasks.

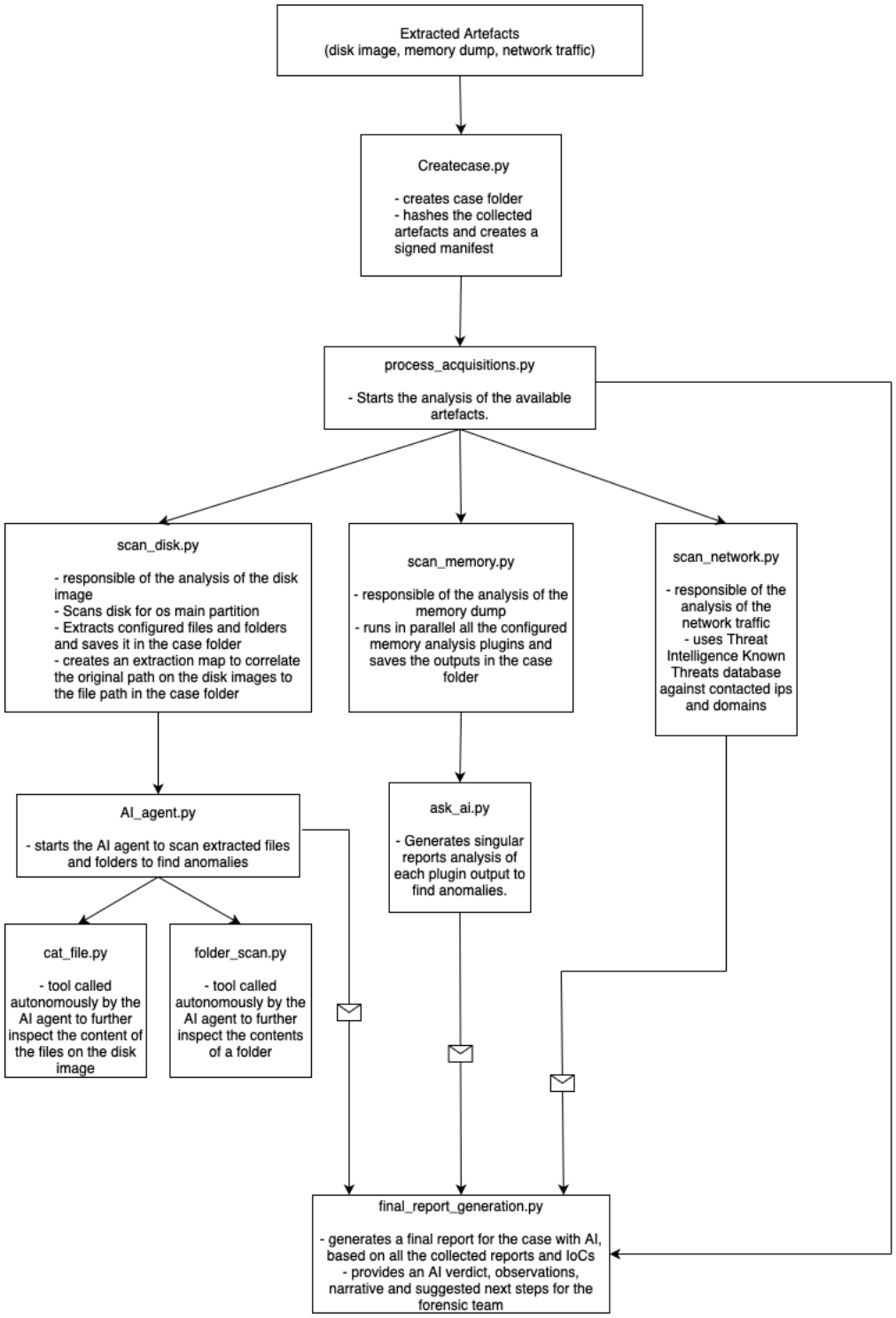

Figure 4.1. Framework call graph and overview

4.1.2 Initialization of a new forensic case: createcase.py

The initialization phase constitutes a critical step in preparing digital forensic investigations. To address this requirement, a dedicated Python utility was developed to automate the creation of a new forensic case, promoting consistency, reproducibility, and evidentiary reliability.

The `create_case.py` script ingests digital artifacts—such as hard disk images, memory dumps, and network traffic captures— and establishes a standardized working environment for subsequent forensic

analysis. It ensures that the case directory structure, metadata generation, and integrity verification are performed in a uniform and auditable manner.

Functional Capabilities. The tool is organized into modular components, each handling a specific aspect of case initialization:

- **Logging configuration:** dual logging records events in real-time (console) and persistently (log file), ensuring full traceability.
- **Artifact acquisition:** forensic artifacts are collected, categorized (disk, memory, or network data), and hashed using SHA-256 to verify integrity.
- **Directory and template creation:** a standardized case structure is generated with subdirectories for acquisition, processing, and reporting. Template files and example modules are instantiated to streamline future workflow.
- **Metadata generation:** core case information (case identifier, OS type, acquisition date) is documented. A manifest of computed file hashes is produced and optionally digitally signed.
- **Signature verification:** public keys can be used to authenticate manifests, ensuring authenticity and evidentiary integrity.
- **Finalization:** artifacts are moved to their permanent acquisition directory, hashes are recomputed and verified, and automated downstream processing can be triggered.

Workflow. The operational sequence proceeds as follows:

1. The investigator provides the case identifier, hash identifier, OS type, and artifact source directory.
2. Artifacts are temporarily copied, hashed, and validated.
3. A permanent case directory is created with standardized subdirectories:
   - `00_meta`: metadata files and optional malware samples,
   - `01_acquisition`: original forensic artifacts,
   - `02_raw_checksums`: manifest of hashes (digitally signed if required),
   - `03_processing`: directories for subsequent analysis (disk, memory, network),
   - `04_reports`: contains the generated final report.
4. Integrity checks recompute file hashes at the destination.
5. Metadata and optional digital signatures are generated to maintain chain-of-custody.
6. Automated analysis pipelines may launch for the next investigative stage.

Through systematic SHA-256 hashing and optional digital signatures, the program safeguards forensic materials' authenticity and provides verifiable proof that no alterations occur during acquisition and preparation. This methodology aligns with chain-of-custody requirements, ensuring transparency and reproducibility. The initialization utility establishes a reproducible, standardized, and secure foundation for forensic investigations, supporting both manual and automated analysis.

An example command for executing the script is:

```
python3 createcase.py --signing-key private_key.pem --verify-key \
public_key.pem 2025 demo ../../../VMs/WinForense/artefacts/ \
windows -y
```

This command will initiate the framework by creating the case folder structure, copying and hashing the artefacts from the '../../../VMs/WinForense/artefacts/' folder, creates the manifest file, signing and verifying it with the given public private key pair, and runs automatically the processing phase without prompt asking for a windows operating system.

4.1.3 Integrity check of artefacts: verify_signatures.py

Ensuring the integrity of collected digital artefacts is essential for maintaining evidentiary reliability throughout the forensic process. To this end, a Python-based utility was implemented to verify the authenticity of forensic case files by validating their digital signatures against a provided public key. This verification step complements the case initialization procedure by confirming that integrity manifests and checksum files have not been altered since their creation.

Functional Capabilities. The script is structured into modular functions that collectively implement automated digital signature verification:

- **Signature verification:** the function `verify_signature()` leverages the `OpenSSL` command-line utility to confirm whether a hash or manifest file matches its corresponding digital signature using the specified public key. Verification outcomes are logged, indicating success or failure.
- **Signature discovery:** the function `get_signature_files()` retrieves all files with the configured signature extension (e.g., `.sig`) searching for the signed manifest file from the checksum subdirectory of the forensic case folder.
- **Batch processing:** the function `process_signatures()` orchestrates the validation process. It iterates over all discovered signature files, locates the corresponding checksum or manifest file (derived by stripping the signature extension), and performs verification. Missing or corrupted files are logged with warnings or errors.
- **Logging:** the program employs Python's logging module, configured via global parameters, to provide detailed and timestamped audit trails of verification activities.

Workflow. The operational workflow proceeds as follows:

1. The investigator specifies the path to the forensic case folder and the path to the public key (PEM format).
2. The script derives the checksum directory based on the case folder and the configuration constants.
3. All signature files in the directory are identified (e.g., `manifest.txt.sig`).
4. Each signature file is paired with its corresponding hash or manifest file and verified against the provided public key using OpenSSL.
5. Results are logged, indicating for each file whether verification succeeded, the corresponding hash file was missing, or the verification failed.

By automating the validation of SHA-256 digital signatures, the utility provides strong assurance that forensic case manifests and associated checksum files remain unaltered after signing. Successful verification reinforces the chain-of-custody by demonstrating integrity and authenticity of the evidence. **Warning:** Non-repudiation is possible if the public and signed keys are associated with a valid certificate, but certificate controls are not automatically made by the framework.
Conversely, verification failures signal either accidental corruption or deliberate manipulation, both of which are critical findings in a forensic investigation. In this way, the integrity verification utility functions as a safeguard, ensuring that all downstream analysis is performed on trustworthy and verifiable artefacts.

An example command for executing the script is:

```
python3 verify_signatures.py windows/2025_demo public_key.pem
```

4.1.4 General analysis workflow

The framework workflow is structured into the following main phases:

- **Disk Analysis**
  - Extraction of files and folders from the disk image using Plaso and Digital Forensics Virtual File System modules. If windows os, also extraction of registry hives and autorun keys using the python-registry library.
  - AI agent-based analysis of selected files and folders to detect anomalies or suspicious behavior.
- **Memory Analysis**
  - Use of Volatility 3 plugins for memory analysis.
  - AI analysis of plugin outputs to identify anomalies or indicators of compromise.
- **Network Analysis**
  - Scanning IP addresses and domains using the VirusTotal API.
- **Final Report**
  - Consolidation of all findings into a single final report using AI.
  - Issuing a final verdict based on the collected evidence and providing suggested next steps for the forensic team.

4.2 Processing orchestrator: process_acquisition.py

Whereas the case initialization and signature verification scripts focus on preparing and validating evidence, the acquisition processing utility (`process_acquisition.py`) acts as an orchestration layer for the automated execution of forensic analysis pipelines. Its primary role is to coordinate the processing of disk, memory, and network artefacts by invoking specialized external tools, managing their execution environment, and consolidating their outputs.

Functional Capabilities. The script is composed of modular functions that encapsulate distinct tasks in the orchestration workflow:

- **Disk scan:** the function `run_disk_scan()` launches a dedicated disk forensic script (`scan_disk.py`), passing the acquired disk image as input.
- **Memory scan:** the function `run_memory_scan()` executes the memory forensic script responsible of running the configured volatility plugins (`scan_memory.py`) and subsequently processes its plugin outputs using an AI-based analysis module (`ask_ai.py`).
- **Network scan:** the function `run_network_scan()` executes the network forensic script responsible of extraction known malicious ips and domains (`scan_network.py`) on packet capture (`.pcap`) files.
- **Virtual environment support:** the helper function `run_script()` enables each tool to be executed within a Python virtual environment, ensuring dependency isolation and reproducibility across heterogeneous toolchains.
- **Final reporting:** once all scans are completed, the script invokes the reporting utility python script (`final_report_generation.py`) to integrate results into a structured forensic report, using AI to generate narratives and suggested next steps for the forensic team.

Workflow. The operational sequence of the tool can be summarized as follows:

1. The investigator specifies the path to the forensic case folder.
2. The script executes the disk scan, memory scan, and network scan in sequence.
3. After each stage, if an error is encountered, the investigator is prompted to decide whether to continue or abort processing.
4. During memory analysis, plugin outputs are further processed through the AI module, and temporary lock files used by rate-limiting mechanisms are removed once processing completes.
5. After all modules have been executed, the final reporting script is launched to consolidate results into a single forensic report.

By automating the coordinated execution of multiple forensic modules, encapsulating their dependencies within isolated environments, and generating an integrated report, the acquisition processing utility provides an end-to-end orchestration layer that streamlines and standardizes the forensic investigation workflow.

4.2.1 Disk Image processing module

The Disk Image processing module is responsible for analyzing disk images acquired from the infected virtual machines. It leverages established forensic tools and integrates AI-driven analysis to identify persistence mechanisms and other indicators of compromise. It is divided in two steps:

1. **Extraction of persistence-related directories/files:** This step involves identifying and extracting directories, files and registry hives that are likely to contain persistence mechanisms, such as the ones discussed in chapter 2 of this thesis.
2. **Agentic AI analysis:** In this step, an AI-agent is employed to analyze the extracted files for suspicious content. This includes searching for unusual patterns, metadata inconsistency and other signs of malicious activity.

Extraction of persistence-related directories/files

The script that manages this is: `scan_disk.py`

This Python script automates the extraction of persistence-related artifacts from disk images and subsequently analyzes them using a an AI agent that can think using cloud based (*Gemini AI*) or any local LLM model. It integrates `dfVFS` library and `Plaso` to support forensic file system access across multiple disk image formats and operating systems.

Main functionalities. The script provides a range of functionalities that cover acquisition, metadata documentation, and AI-driven reasoning:

- **Volume scanning and partition selection:** a custom mediator guides `dfVFS` to select the most relevant partition (the partition containing the file system), prioritizing NTFS when available, or otherwise the largest partition. This ensures that key operating system volumes are consistently analyzed.
- **OS detection and artefact targeting:** the file system type is analyzed to classify the image as Linux or Windows. Corresponding predefined artefact paths are then loaded (e.g., autorun registry hives, startup folders, cron jobs).
- **File and directory extraction:** artefacts are recursively copied to a case-specific output folder. Path normalization and collision handling mechanisms prevent overwriting when duplicate names occur (eg. multiple NTUSER.DAT files assigned to different users).
- **Registry hive analysis:** if registry hives (`NTUSER.DAT`, `SOFTWARE`, `SYSTEM`) are encountered, autorun-related keys are automatically parsed using `python-registry`, and human-readable reports are generated.
- **Extraction map:** a CSV map is created to document the correspondence between original disk paths and their local copies, ensuring transparency and reproducibility.
- **AI-assisted reasoning:** AI analysis is launched for every extracted artefact or directory. Individual files are reported in JSON, directories in Markdown. Reports are consolidated under a dedicated "agent reports" folder for easy review.
- **Logging and error resilience:** a configurable logging system records debug, warning, and error messages, ensuring traceability of failures without halting the entire process.

Workflow. The automated procedure follows a structured pipeline:

1. The investigator specifies the input disk image and case folder.
2. Volumes are scanned; the partition containing the file system is selected.
3. The OS type is inferred and corresponding persistence artefact paths are loaded from the main configuration file.
4. files and folders are recursively extracted, while registry hives are parsed and results are exported in plain-text.
5. An extraction map in CSV format is saved, linking original and local artefact locations.
6. The AI agent is invoked on the extracted artefacts, producing JSON/Markdown reports that highlight potential persistence mechanisms.

Through its combination of file system scanning, registry hive extraction, and AI-assisted post-processing, the tool extends classical forensic methods with automated reasoning. The CSV extraction map guarantees reproducibility of the acquisition process, while the AI-generated reports provide actionable insights into subtle or concealed persistence artefacts. As such, the utility plays a central role in bridging evidence extraction with interpretative analysis in digital forensic workflows.

Agentic AI analysis

The `AI_agent.py` script implements a semi-automated forensic reasoning framework that integrates deterministic Python-based extraction scripts with probabilistic reasoning from the Gemini LLM or a local Ollama backend. It coordinates the iterative analysis of disk image artefacts by orchestrating a loop of LLM-based inference and external tool execution until a forensic conclusion is produced, using LangGraph.

Configuration and Logging. The utility imports all runtime parameters from a centralized configuration file, including Gemini API keys, model names, token limits, script paths, timeouts, and logging settings. Multiple API keys are maintained and rotated automatically in case of daily quota exhaustion. Logging is dual-channel: results are written both to the console and to a persistent Markdown log, with each run beginning from a clean execution log to preserve case isolation.

State Management. Analysis state is maintained in the `MyState` structure, a typed dictionary containing the history of LLM responses, the current step index, flags for finalization, and the most recent output. Long conversation histories are automatically summarized through Gemini or Ollama to reduce prompt size while retaining forensic context.

Reasoning Workflow. The agent is modelled as a LangGraph state machine with two primary stages:

1. **Think (LLM reasoning):** A prompt is built from the execution context and the current input (directory listing or file content). The LLM is instructed to behave as a forensic analyst and may issue either a *Final Answer* with IoCs and anomalies, or an *Action* request to run a Python tool for further inspection. Large inputs are segmented into row-preserving chunks and reintegrated after deduplication to keep the analysis tractable.
2. **Act (tool execution):** When an action is requested, the agent validates the script path, executes it under timeout control, captures both stdout and stderr, and appends the results to the conversation history. Outputs are truncated if overly long in local mode.

This think–act loop continues until the LLM outputs a conclusive forensic verdict.

Error Handling and Resilience. Robustness is provided through layered exception management:

- **API key rotation:** on daily quota exhaustion, the client switches to the next available Gemini key.
- **Transient failures:** errors such as rate limiting or server faults trigger exponential retry attempts up to a configurable maximum.
- **Tool errors:** failures in auxiliary scripts are logged but do not terminate the entire reasoning cycle.

File Examination. When invoked with the `--examine` option, the agent retrieves the target file content using an external extraction script and submits it to the LLM. The model must return a structured JSON response including the filename, a suspiciousness flag, explanatory reasoning, and a list of IoCs. Reports are saved to an optional output file for integration with downstream analysis pipelines.

Execution Flow. Two principal operational modes are supported:

1. **Folder analysis:** the selected starting folder for the analysis from a disk image is scanned via an auxiliary script (`folder_scan.py`), and the directory listing is iteratively explored through the think–act loop.
2. **File analysis:** a single artefact is inspected in depth, visualizing its contens via an auxiliary script (`cat_file.py`) and structured results are written into a JSON file.

Reporting. All interactions are preserved in Markdown logs, including LLM responses, observations, tool outputs, and errors. The final stage produces a consolidated response containing the most recent reasoning step, extracted forensic observations, and a verdict of suspicious findings. Optionally, a compact final report containing only the last LLM, observation, and answer blocks may be exported.

The design thus implements a hybrid forensic agent: deterministic utilities provide guaranteed extraction fidelity, while the LLM contributes adaptive reasoning and anomaly detection. Together, they enable reproducible, explainable, and resilient forensic workflows over disk images.

AI-agent tools

As said before, the AI agent can call these two auxiliary tools:

- **Cat Tool:** The `cat_file.py` tool script implements a minimal but effective forensic utility to locate the root operating system partition within a disk image and shows the content of a file relative to that root. It integrates `dfVFS`, Plaso's `StorageMediaToolVolumeScanner`, and `pytsk3` to achieve cross-platform support for Linux and Windows file systems with bounded, investigator-friendly output.

  Core functionalities.

  - **Automated partition scanning:** a custom mediator (`MyMediator`) ensures non-interactive enumeration of partitions in the provided disk image.
  - **OS root detection:** the script attempts Linux detection first, then falls back to Windows heuristics:
    1. Linux roots are validated by checking for canonical directories (`/usr`, `/var`, `/etc`, `/bin`) at the file system root.
    2. Windows roots are confirmed when top-level entries (`Windows`, `Program Files`, `Users`) are found on NTFS/FAT volumes.
  - **file content:** given a relative path, the tool retrieves and prints the file content:
    * NTFS files are accessed via `pytsk3`, using a custom `Img_Info` adapter to bridge `dfVFS` file objects with TSK's API.
    * EXT files are read using `dfVFS` path specifications, with symbolic links detected and reported instead of being followed.
  - **Safe output formatting:** content is first decoded as UTF-8; on failure, a byte-wise fallback is applied with simple run-length encoding (e.g., `X*12*`) to preserve readability under `MAX_OUTPUT` and `CHUNK_SIZE` constraints.
  - **Configurable logging and safety:** output limits are configurable, adapting the tool to quick triage or deeper inspection use cases.

  Workflow. The typical workflow is:

  1. The tool action is called by the AI agent, that specifies the disk image path and the file path relative to the OS root.
  2. The scanner enumerates available partitions and constructs a scan tree.
  3. Linux root detection is attempted; if unsuccessful, Windows heuristics are applied.
  4. If no root is found, the process terminates with an error. Otherwise, the chosen root is used to resolve the requested file.
  5. If a file is specified, its content is displayed with strict safety bounds and sanitized formatting.

  Forensic considerations. This utility provides the AI agent with a compact method for inspecting the content of suspicious files that needs to be further analyzed (e.g., configuration files, startup scripts, logs) without mounting the full file system. Its bounded output minimizes risks associated with large or binary data, while the conservative root detection heuristics ensure resilience across diverse disk images. Symbolic links are reported to prevent accidental traversal, and NTFS access via `pytsk3` offers fine-grained forensic control.

  By combining flexible scanning, OS-aware heuristics, and safe file content display, the script delivers an efficient forensic tool to permit the AI-agent to scan the content of files.

- **Folder ScanTool:** This tool focuses on analyzing the contents of a folder within the disk image. The script is a lightweight forensic utility for enumerating the contents of directories within disk images. It leverages `dfVFS` and Plaso's `StorageMediaToolVolumeScanner` to provide a cross-platform, read-only file system navigator with OS-aware heuristics for root detection.

Core functionalities.

- **Automated volume scanning:** has the same workflow as the above mentioned `cat tool`.
- **Directory listing:** prints contents with aligned columns, including:
  - Permissions (POSIX style for Linux, simplified indicators for Windows)
  - Owner and group identifiers
  - File size
  - Last modification timestamp
  - File or directory name
- **Cross-platform formatting:** supports Linux-style (`ls -l`) and Windows-style listings. Missing or incomplete metadata is replaced with placeholders to prevent errors.

Workflow.

1. The AI agent invokes the tool providing the disk image and a directory path to list.
2. The scanner enumerates volumes and constructs a hierarchical scan tree.
3. Linux root detection is attempted first; if unsuccessful, Windows heuristics are applied.
4. Upon identifying a root partition, the corresponding file system is opened using DFVFS.
5. The requested directory is listed with the appropriate formatting routine based on OS type.

Forensic relevance. The tool provides a rapid mechanism to provide a directory listing without mounting the image. Directory listings expose critical metadata useful for triage and preliminary analysis, such as identifying system directories, user files, or anomalous content.

Notes and safeguards.

- Root detection heuristics are conservative; unusual or highly customized systems may require manual verification.
- Missing or incomplete metadata is safely replaced with placeholders to ensure output consistency.
- Windows-style permissions are simplified and do not reflect full NTFS access controls.
- The script is read-only and avoids modifying the source media.

So `folder_scan.py` is a compact, cross-platform file system navigator for forensic triage. It combines DFVFS, Plaso volume scanning, and OS-specific heuristics to identify root partitions and produce clear, aligned directory listings for both Linux and Windows file systems to feed the AI-agent.

4.2.2 Memory processing module

The memory processing module focuses on analyzing RAM dumps acquired from the inspected machine. It utilizes Volatility 3 plugins for in-depth memory forensics and incorporates AI-driven analysis to identify anomalies and potential indicators of compromise.

Setup of the Volatility 3 environment

This step is crucial for memory processing and must be executed manually if inspection machines that have Linux os. The main difficulty lies in the fact that Volatility 3 requires specific symbol table images for analyzing memory, which are not always easy to obtain. While Windows 11 (or any other Windows distribution) symbols are automatically downloaded by Volatility, for Ubuntu symbols (or any other Linux distribution) is required for the user to manually feed them into the volatility framework. While there are some repositories online made by users that distribute ready-to-use symbol tables for various linux distribution, the exact symbol table for the kernel of the Linux distribution some may use, is not guaranteed to be in it. So i will give a guide in case the forensic investigator needs to extract it manually.

The procedure is the following:

Step 1: Check the Banner from the Memory Dump

```
python3 vol.py -f memdump.lime banners
```

this will give the user the **exact banner to match** in the symbol table. For example, in my case was:

```
Linux version 6.8.0-60-generic (buildd@lcy02-amd64-098) (x86_64-linux-gnu-gcc-12
(Ubuntu 12.3.0-1ubuntu1~22.04) 12.3.0, GNU ld (GNU Binutils for Ubuntu) 2.38)
#63~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Tue Apr 22 19:00:15 UTC 2
(Ubuntu 6.8.0-60.63~22.04.1-generic 6.8.12)
```

Step 2: Verify if the Community Repository already provides the Symbol Table In my case, the symbol table was already available in a community-maintained repository [39].

Step 3 (if not in the Repository): Build a Matching Symbol File A comprehensive guide on how to create Linux symbol tables for Volatility using a tool called `dwarf2json` is available in HackTheBox resources [40].

Step 4: Inspect the Banner in the generated Symbol File

```
python3 vol.py isfinfo -f volatility3/symbols/linux/linux-<version>.json.xz
```

At this point, it is **necessary to ensure that both banners match *byte-for-byte***. In my personal experience i got a lot of trouble before realizing this point: even minor changes in the banners (eg. the day or the hour in the date of release) will cause volatility3 to not work.

Step 5: Install the symbol table in Volatility3 framework Copy the symbol table (.json or .json.xz) to the correct folder for linux symbols, which is:

```
<volatility3_home_path>/symbols/linux/
```

With this setup Volatility3 is ready to scan both Windows and your specific Linux distribution memory dumps.

Parallel execution of Volatility plugins: scan_memory.py

The memory processing module leverages the parallel execution capabilities of Volatility 3 to enhance the efficiency of memory analysis. By distributing the workload across multiple CPU cores, the system can concurrently process different memory artifacts, significantly reducing the time required for in-depth analysis. This approach is particularly beneficial when dealing with large memory dumps or when multiple plugins need to be executed in tandem. The script that manages this is: `scan_memory.py`

Main functionalities.

- **Task orchestration for memory analysis:** Builds a list of Volatility plugin executions across multiple memory dumps organized by OS type (Linux/Windows) and case folders. Tasks include dynamically constructing shell commands and managing output paths.
- **Parallel plugin execution:** Uses `ThreadPoolExecutor` to run multiple Volatility plugins concurrently, optimizing CPU usage and reducing total runtime. Progress is displayed via `tqdm` progress bars.
- **Logging and monitoring:** All activity—including task start, success, failure, and cleanup—is logged to a configurable file. Logs are timestamped and structured for easy review.
- **Output validation and cleanup:** Some volatility plugins, may fail to produce outputs for various reasons (e.g. malformed memory data sector). So after plugin execution, output files with very short content (metadata with empty output), are automatically removed.
- **Configurable execution parameters:** Global configuration variables control maximum parallel workers, and logging settings.

Workflow.

1. The memory dump is parsed though the default case artefact directory (or optionally a custom external folder).
2. Volatility3 plugins tasks are executed in parallel using a thread pool; each task logs its start and reports success or failure.
3. After completion, output files with no content are removed to reduce clutter.

Notes and safeguards.

- Parallel execution is controlled by `MAX_WORKERS` to avoid overloading system resources.
- The script uses read-only operations on memory dumps, maintaining forensic soundness.

This design supports efficient and reproducible memory forensics at scale. Parallel task management accelerates investigations, logging ensures accountability, and conservative cleanup maintains clarity in results. By relying exclusively on read-only operations, the script safeguards forensic integrity, while adjustable global parameters—such as the maximum number of workers—allow investigators to balance performance against resource constraints in diverse environments.

AI-driven analysis of memory artifacts: ask_ai.py

After generating the Volatility plugin outputs, this script creates a report analyzing them using AI.

`ask_ai.py` is a dedicated Python utility script developed to automate the production of structured forensic reports using LLMs.

The script generates narrative Markdown reports that emphasize anomalies and interpret their forensic relevance. It integrates with large language models (LLMs)—either via the Gemini API or local LLMs (e.g., Ollama)—to augment traditional forensic interpretation, while enforcing strict controls on rate limiting, token management, and quota compliance to guarantee operational stability.

Functional Capabilities. The tool is organized into modular components, each addressing a distinct stage of automated report generation:

- **Forensic report generation:** produces Markdown-formatted sections directly from plugin outputs, highlighting deviations of evidentiary importance and providing contextual interpretation.
- **LLM integration:** communicates with the Gemini API or local Ollama models to generate structured analyses, supporting multi-key rotation for quota exhaustion and configurable retry strategies.
- **Rate limiting and token control:** when using cloud-based services, enforces maximum requests per minute (RPM) and tokens per minute (TPM), validates input size against service constraints, and automatically waits, retries, or rotates API keys to maintain reliability.
- **Input and output management:** ingests target output files, dynamically constructs prompts, and writes Markdown reports to structured case directories for reproducibility.
- **Chunking strategy for large inputs:** if the input exceeds model size limits, the script splits the data into smaller row-based chunks, processes each sequentially, and reassembles results into a unified forensic report. This part is extremly useful for local LLMs configurations, as larger input token sizes significantly increase the model's processing time.

Workflow. The operational sequence unfolds as follows:

1. The processing orchestrator calls this script sequentially with all the plugin outputs available in the case.
2. The target system output is parsed and a structured prompt is dynamically constructed, instructing the model to identify anomalies and generate an evidence-focused Markdown report.
3. For Gemini mode, the rate-limiting mechanism validates the prompt against quota thresholds (RPM/TPM), waiting when necessary and handling retries or API key rotation automatically.
4. For local mode, oversized inputs are chunked and processed iteratively, with intermediate results merged into a coherent final report.

5. The resulting Markdown report is written to the appropriate case folder within a standardized directory structure, maintaining traceability.

Through systematic analysis of memory outputs, the utility highlights suspicious activity and abnormal system behavior in a reproducible manner. The combination of automated anomaly detection, structured reporting, chunk-based processing for large datasets, and robust operational safeguards ensures that the generated outputs are both analytically valuable and evidentially sound. By integrating large language models under strict control mechanisms, the tool supports investigators in producing transparent, reliable, and actionable forensic analyses.

4.2.3 Network processing module

The network processing module automates the analysis of network traffic captures obtained from potentially compromised machines. It leverages Wireshark for packet inspection and integrates with threat intelligence services to identify suspicious IP addresses, domains, and other observable indicators.

The script `scan_network.py` implements this functionality, providing a reproducible and structured workflow for network forensic triage.

Functional Capabilities. The module is organized into distinct components, each addressing a key stage of network analysis:

- **IOC extraction from PCAPs:** Parses network capture (PCAP) files to extract potential Indicators of Compromise (IoCs), including IP addresses, domains, and file hashes, using the `Pcap` helper module.
- **Threat intelligence integration:** Queries the VirusTotal API for each extracted IOC, retrieving summary statistics that classify detections as malicious, suspicious, harmless, or undetected.
- **Rate limiting:** Enforces configurable delays between queries (`VT_RATE_LIMIT_DELAY`) to ensure compliance with API request constraints.
- **Result aggregation and management:** Consolidates query results into a CSV file while saving detailed JSON reports for IOCs flagged as malicious or suspicious.
- **Summary reporting:** Produces concise overviews of malicious or suspicious IOCs, including previews of relevant sections of the full JSON report for rapid assessment.

Workflow. The operational sequence proceeds as follows:

1. The processing orchestator sends command-line arguments, including the PCAP file to process, the source of IOCs, and the destination case folder for output storage.
2. The script extracts potential IoCs from the PCAP file using the `Pcap` module.
3. For each IOC, the VirusTotal API is queried, with summary statistics collected while respecting rate-limiting constraints.
4. Aggregated results are saved to a CSV file, and detailed JSON reports are retained for IOCs identified as malicious or suspicious.
5. A summary of all flagged IOCs is printed to the console, providing a quick reference for further analysis.

Forensic Relevance. This module provides a systematic and reproducible method for identifying and evaluating network-based Indicators of Compromise. By integrating threat intelligence services, it enables rapid triage of suspicious IP addresses, domains, and o ther network observables, while producing both high-level summaries and detailed reports suitable for forensic investigation.

Safeguards and Operational Notes.

- IOC types are normalized to comply with VirusTotal API endpoints.
- HTTP errors and missing data are handled gracefully, with statuses recorded in the results.
- Rate-limiting ensures compliance with API constraints, preventing service throttling or rejection.

- CSV outputs are sanitized to maintain readability and manageable file sizes, while detailed JSON files preserve comprehensive forensic information.

Through these measures, the network processing module ensures that extracted IoCs, analyses, and reports are accurate, reproducible, and evidentially reliable, supporting both automated triage and detailed forensic investigation.

4.3 Creating the final report based on acquired evidence: final_report_generation.py

The `final_report_generation.py` module automates the collection, formatting, and synthesis of artefacts into a unified case report. It provides a scalable pipeline for aggregating outputs from disk, memory, and network analyses, converting heterogeneous formats into structured Markdown, and enriching the results with an AI-generated forensic summary.

The system begins by traversing predefined subfolders within a case directory, scanning for files with supported extensions. Each artefact is loaded and normalized according to its type: plain text, JSON, and Markdown are incorporated directly, while CSV files are parsed into structured tables and re-encoded for readability. Unsupported or missing formats are logged with warnings, preserving transparency while avoiding workflow interruptions. File contents are consistently formatted and prefixed with block-quoted syntax, ensuring that original records remain visually distinct within the consolidated report.

Once artefacts are assembled, the module generates a hierarchical Table of Contents. This is derived automatically from Markdown headings, with anchors formatted in a manner compatible with version-controlled platforms such as GitHub. This guarantees navigability across potentially lengthy reports, while maintaining alignment with forensic documentation standards.

A critical feature of the tool is the integration of large language models for evidence-driven summarization. The system can operate either with the local Ollama API or remotely via Google Gemini, selected by the investigator through a simple command-line flag. To accommodate long or fragmented input, content can be segmented into word-based chunks, with each part summarized iteratively before being synthesized into a final verdict. This chunking strategy prevents token overflows while enabling contextual aggregation across intermediate summaries. When operating locally, retries and backoff mechanisms protect against transient failures during model inference. In cloud-based operation, the script automatically rotates across multiple Gemini API keys if quotas are exceeded, and introduces delays for rate-limited or server-side errors. These safeguards ensure continuity of operation even under constrained conditions.

The final report is composed of three integrated sections: the collected artefacts grouped by analysis domain, the AI-assisted synthesis presented as a forensic summary and verdict, and a navigable Table of Contents for structured access. The entire process is logged in detail, including discovered files, processing decisions, API interactions, and error handling routines. This combination of deterministic artefact collection and adaptive summarization provides both reproducibility and interpretative support, balancing the rigor of forensic documentation with the efficiency of automated analysis pipelines. A more detailed description of the contents of the final report is given in the next chapter.

Chapter 5

Results Discussion and Evaluation

5.1 Obtained reports and IoCs detected

The Framework generates a final report that includes all the results on the analysis taken by the automated tools and AI, giving a fast overview of the findings and the IoCs detected on the target system. The report is structured in different sections, each one dedicated to a specific aspect of the analysis. The sections are the following:

5.1.1 Disk Analysis

This section aggregates filesystem and registry hives artifacts.

- **MD reports:** Contains all the reports generated by the AI regarding the AI crawling of the file system configured (in config.py) folders. An example can be:

```
## LLM Response
I have reviewed the contents of the '/etc/cron.d/.placeholder' file. The file
contains comments explaining its purpose: it's a placeholder to prevent the
package manager ('dpkg') from removing the '/etc/cron.d' directory if it
becomes empty. This is a legitimate practice, and the file itself contains
no executable commands. Therefore, my initial suspicion was unfounded, and
this file is not malicious.

The other files in the directory, 'anacron' and 'e2scrub_all', are standard system
cron jobs with modification dates well before any potential incident, making them
unlikely to be involved.

Based on this, the '/etc/cron.d/' directory does not appear to contain any anomalous
files or persistence mechanisms.
Observation: The content of '/etc/cron.d/.placeholder' confirms it is a benign system
file. The other files in the directory also appear to be legitimate. No anomalies were
found in this directory.
Final Answer: {}
```

- **JSON reports:** Contains the reports of the analysis of a single file:

```
**etc_profile.d_debuginfod.csh.json**

# File Analysis Report

"""json
{
  "file_name": "/etc/profile.d/debuginfod.csh",
  "suspicious": true,
  "reasoning": "The script is intended to configure the DEBUGINFOD_URLS environment
```

```
  variable by reading from system configuration files. However, the very last line,
  'setenv DEBUGINFOD_URLS \"\"', unconditionally overwrites this variable with an
  empty string. This action nullifies the entire purpose of the script and effectively
  disables the debuginfod client functionality for any user logging in. This could
  be a deliberate act of sabotage to hinder debugging and analysis on the system,
  which is a suspicious modification.",
  "indicators": [
    "The script's final command 'setenv DEBUGINFOD_URLS \"\"' negates all preceding
    logic and disables the intended functionality."
  ]
}
"""
```

- **TXT reports:** Contains all the configurable registry key found in the extracted registry hives. for example:

```
Registry autorun keys extracted from NTUSER_WsiAccount.DAT
============================================================

[Software\Microsoft\Windows\CurrentVersion\Run]
  OneDriveSetup = 'C:\\Windows\\System32\\OneDriveSetup.exe /thfirstsetup'

[Software\Microsoft\Windows\CurrentVersion\Run]
  OneDriveSetup = 'C:\\Windows\\System32\\OneDriveSetup.exe /thfirstsetup'

[Software\Microsoft\Windows NT\CurrentVersion\Winlogon]
  ExcludeProfileDirs = 'AppData\\Local;AppData\\LocalLow;
  $Recycle.Bin;OneDrive;Work Folders'
  BuildNumber = 26100
  FirstLogon = 0
```

5.1.2 Memory Analysis

This section containts the AI generated reports of the configured volatility3 plugin raw outputs. It is structured as follows:

- **MD reports memory:** Contains all the AI generated reports regarding the output analysis of the volatility3 plugins. for example:

```
## Memory Analysis - PsList

### Plugin Overview

The `PsList` plugin enumerates the processes running on the system at the time of
memory acquisition.
This is a fundamental step in memory analysis, used to identify running
applications, system services,and potentially unauthorized or malicious processes.
The plugin provides key details for each process, including its Process ID (PID),
Parent Process ID (PPID), name, user context (UID/EUID), and creation time.

### Narrative Analysis

The process listing reflects a standard boot sequence and user session for a
Linux graphical desktop environment.
The initial processes are kernel threads (PPID 2) and the `systemd` init
process (PID 1).These are followed by the startup of various system-level
daemons and services, all children of `systemd`.

...
```

```
### Forensic Interpretation

The process ‘b560f76f7603e3e‘ (PID 763) is highly suspicious and warrants
immediate further investigation. Its cryptic name, execution as root, and
persistence as a service are strong indicators of potential malicious activity.
The process appears to be a backdoor or remote access trojan designed to run
automatically at system startup.

**Recommended Next Steps:**
1.  Dump the process memory for PID 763 to perform static and dynamic
analysis on the executable.
2.  Use the ‘netscan‘ plugin to determine if this process has established
any network connections.
3.  Examine systemd service unit files (e.g., in ‘/etc/systemd/system/‘,
 ‘/run/systemd/system/‘) for entries related to this executable.
4.  Correlate the process creation time with system logs (e.g., ‘journalctl‘,
 ‘/var/log/syslog‘) to find related events or error messages.
5.  Analyze the command line arguments and environment variables for this
process to gain further context.
```

5.1.3 Network Analysis

This section contains a csv report about all the contacted IPs and domains within a time span of 60 seconds analysis. The report checks all the ips and domains against the VirusTotal Known Threat intelligence API to discover which vendors assign that value as malicious, suspicious or harmless. This kind of analysis serves to an investigation to confirm a compromise with possible C2 communications. An example:

```
    **vt_ioc_results.csv**

"""csv
ioc,malicious,suspicious,harmless,undetected
"{’type’: ’ip’, ’value’: ’91.189.91.100’}",0,0,60,35
"{’type’: ’domain’, ’value’: ’lc.liuddiase1li.com’}",3,3,58,31
"{’type’: ’domain’, ’value’: ’canonical-bos01.cdn.snapcraftcontent.com’}",0,0,63,32
"{’type’: ’ip’, ’value’: ’185.125.188.61’}",0,0,60,35
"{’type’: ’ip’, ’value’: ’2001:67c:1562::1d’}",0,0,0,95
"{’type’: ’domain’, ’value’: ’dashboard.snapcraft.io’}",0,0,64,31
"{’type’: ’domain’, ’value’: ’api.snapcraft.io’}",0,0,63,32
"{’type’: ’ip’, ’value’: ’185.125.188.62’}",0,0,62,33
"{’type’: ’ip’, ’value’: ’162.159.140.220’}",0,0,62,33
"{’type’: ’domain’, ’value’: ’lc.yusfezu99.com’}",0,0,0,95
"{’type’: ’ip’, ’value’: ’2620:2d:4000:1012::fc’}",0,0,0,95
"{’type’: ’ip’, ’value’: ’202.58.105.227’}",0,0,0,95
"{’type’: ’ip’, ’value’: ’185.125.188.54’}",0,0,62,33
"{’type’: ’ip’, ’value’: ’91.189.91.101’}",0,0,60,35
"{’type’: ’ip’, ’value’: ’2620:2d:4000:1010::117’}",0,0,0,95
"{’type’: ’ip’, ’value’: ’2620:2d:4000:1010::42’}",0,0,0,95
"{’type’: ’ip’, ’value’: ’172.66.0.218’}",0,0,61,34
"{’type’: ’ip’, ’value’: ’2620:2d:4000:1010::2d6’}",0,0,0,95
"{’type’: ’ip’, ’value’: ’185.125.188.57’}",1,0,60,34
"{’type’: ’domain’, ’value’: ’lc.zhanas1fa32.com’}",3,1,60,31

"""
```

5.1.4 Forensic Summary & Verdict

This final section fuses disk, memory, and network findings into a concise narrative using AI, trying to give a final verdict about the state of the machine: Compromised (Suspicious) or Clean (Inconclusive / Not Compromised). The AI also suggests next steps for the forensic team to continue the investigation. For example:

```
## Forensic Summary & Verdict
### **Suspicious/malicious activity (report)**

A comprehensive analysis of the provided disk, memory, and network evidence reveals
multiple correlated indicators of compromise. While forensic analyst activity was
identified and excluded, several key findings strongly suggest the system has been
compromised.

1.  **Persistence Mechanism Established on Disk:**
    Disk analysis uncovered a suspicious systemd service file named
    `remote-client.service` created in `/etc/systemd/system/` on 2025-09-22.
    The `multi-user.target.wants` directory was modified minutes later, indicating
    this service was enabled to start automatically on boot. This is a common technique
    used by malware to establish persistence.

2.  **Suspicious Process Running in Memory:**
    Memory analysis corroborates the disk findings. A process named `b560f76f7603e3e`
    (PID 763) was found running with root privileges (UID 0). Its parent process is
    `systemd` (PID 1), and its start time of `2025-09-22 13:33:55` aligns closely with
     the creation of the `remote-client.service` file. The randomized, hexadecimal-like
      process name is a strong indicator of malware attempting to evade detection.

3.  **Network Communication with Malicious IOCs:**
    Network analysis identified communications with domains and an IP address flagged
    as malicious by VirusTotal. Specifically, the domains `lc.liuddiase1li.com` and
    `lc.zhanas1fa32.com`, and the IP address `185.125.188.57` have multiple malicious
    verdicts. The pattern of the domain names is consistent with Domain Generation
    Algorithms (DGAs) often used for Command and Control (C2) communication.

4.  **Anti-Forensic/Anti-Debugging Modifications:**
    Two system configuration scripts, `/etc/profile.d/debuginfod.sh` and
    `/etc/profile.d/debuginfod.csh`, were modified. In both cases, a line was added to
     unconditionally clear the `DEBUGINFOD_URLS` environment variable, effectively
     disabling the `debuginfod` client. This action serves no legitimate purpose and is
      likely an attempt to hinder debugging and analysis of processes on the system.

Forensic analyst activity, including the use of `su`, compiling and loading the LiME
kernel module with `insmod`, and handling a sample file named `virus.elf`, was observed
in the memory analysis but is considered benign and consistent with the evidence
collection process.

---
**Final Verdict: Suspicious**

**Reasoning:** The evidence is strongly correlated across disk, memory, and network
artifacts. A persistence mechanism (`remote-client.service`) on disk directly corresponds
to a suspicious, randomly named process (`b560f76f7603e3e`) running in memory.
This activity is further linked to network communications with known malicious domains
and an IP address. The presence of anti-forensic modifications to system scripts
strengthens the conclusion of malicious intent.

**Next Steps for Forensic Team:**
1.  **Analyze the Service File:** Examine the contents of
`/etc/systemd/system/remote-client.service` to determine the full path of the
executable it launches.
2.  **Recover the Malware:** Based on the service file, locate and recover the
binary associated with the `b560f76f7603e3e` process for static and dynamic analysis.
3.  **Full Network Log Review:** Correlate the malicious IOCs from the network
report with full packet captures or firewall logs to determine the timeline and
 extent of C2 communication.
```

5.2 Prompt engeneering

A focused prompt-engineering strategy was essential to make the forensic AI components accurate, explainable, and operationally reliable at scale across disk, memory, and network artefacts. The final prompt set, that can be consulted in A.4, enforces tight output schemas, embeds domain guardrails that suppress false positives from forensic tooling and workflows, and enables chunked, stateful reasoning that respects token and latency constraints for both local and cloud LLMs. In practice, these design choices serve four main goals:

- Standardize outputs for deterministic downstream parsing
- Reduce hallucinations by constraining style and scope
- Encode forensic domain caveats (e.g., LiME, DumpIt side-effects, volatility3 plugin malfunctions)
- Support long-context inputs via chunking and summarization for local models under strict token context lenght.

These goals reflect the framework's modular pipeline and the dual-mode operation with Gemini and local LLMs described in Chapter 4.

Output schemas and parsing Every operational prompt mandates exact formats (JSON keys or Markdown sections) so that agent outputs can be machine-parsed without brittle heuristics. Examples include the DISK file analyzer's strict JSON and the chunk summarizers' fixed-key JSON. Also, to reduce randomness in the responses for local models, the TEMPERATURE parameter of the LLM model was set to '0.1'. The temperature parameter is used to control the level of creativity and variability in the model's outputs. A lower value (closer to 0) makes the responses more deterministic and focused as we need in our case because an agentic behevior is needed for post processing, while higher values increase diversity and randomness, which can lead to more creative but potentially cause promblems in the parsing. This minimized post-processing errors and enabled reliable aggregation into the final report generator that composes multi-source narratives deterministically.

Domain guardrails to reduce false positives Prompts explicitly whitelist analyst actions (e.g., extracting password-protected samples "infected", LiME insmod, memory acquisition tools) and caution against reading plugin parsing artifacts as malicious (e.g., absurd UID/GID, corrupted names, impossible timestamps). This codifies recurring caveats from the methodology and implementation, materially lowering false positives in Volatility outputs and during disk scanning decisions. The final summarization prompt further enforces conservative verdicting, requiring corroboration across evidence types before "Suspicious," and allowing "Not Compromised" or "Inconclusive" otherwise.

Token management and chunking Long artefacts (directory listings, Volatility dumps, multi-plugin corpora) are processed via chunk prompts that:

- preserve row boundaries
- filter for anomalous/relevant lines
- carry compact context between chunks
- emit stable JSON for lossless merge

Agentic control and action formatting The disk-folder analysis prompt implements a constrained think–observe–act loop: the LLM must decide anomalies, propose a single highest-priority Action with exact CLI syntax, and keep all other leads in "Final Answer" for later follow-up. This reduced tool-chaining unuseful information and encouraged depth-first exploration with traceable state summaries via the DISK summarizer prompt, aligning with the LangGraph state machine and auxiliary scripts (catfile, folderscan).

Date- and path-aware heuristics Disk prompts embed precise rules for timestamp anomalies (future-only, strict greater-than comparisons) and platform-accurate path separators, plus treatment of symlinks and SUID/SGID anomalies. These micro-heuristics, enforced in-prompt, cut spurious flags from legitimate historical files and from virtualization artefacts intentionally excluded, improving precision without external code filters.

Conservatism and evidence correlation The final report prompt encodes conservative decision rules: do not over-weight single weak artefacts; require cross-domain corroboration (e.g., disk persistence plus matching network hits) for "Suspicious"; and emphasize follow-up steps over categorical claims under ambiguity. This directly improved verdict calibration across the integrated report generator described in the implementation

Cloud vs Local Because local LLMs have smaller contexts and slower long-input throughput, the prompts emphasize chunking, brevity, and deterministic schemas; for cloud models, rate limits and key rotation are complemented with prompts that avoid verbose reasoning and strictly control output length. In both modes, the same guardrails and schemas yielded comparable structure and reduced variance, facilitating fair efficiency and privacy comparisons in Chapter 5.

Representative prompt patterns

- Strict JSON for file triage with named keys and boolean "suspicious" to enable automated counting and dashboards. The prompt reminds models of current date handling to avoid time-based false positives.
- Folder analysis with "LLM / Observation / Final Answer / Action" blocks to segment reasoning, findings, and next-step tooling, capped at a single action to stabilize the loop and logs.
- Memory reporting with baseline-aware comparison, a compact summary table, narrative significance, and explicit caveats on LiME and plugin artifacts, producing courtroom-friendly sections with minimal editing.
- Chunk templates that force single-JSON-object outputs and forbid extra prose, preventing parser failures during large-batch runs across plugin outputs and directory trees.

5.2.1 Observed effects in practice

Suppressing known analyst- and tool-related artefacts, together with filtering plugin anomalies, reduced false positives across both memory and disk phases, thereby strengthening the reliability of automated alerts prior to human review. At the same time, the combination of chunking and summarization preserved throughput for local models, prevented quota overruns in cloud deployments, and, through the use of consistent schemas, simplified the merging of results in the reporting pipeline. Finally, the enforcement of a single-action constraint and the application of a future-date suppression rule minimized noisy escalations, ensuring that tool invocations were directed toward the highest-risk files and directories while maintaining a reproducible audit trail in the agent logs.

5.3 Why Local LLM: suspect privacy concerns in the forense investigation

Local LLM inference keeps sensitive disk, memory, and network artefacts on-premise, reducing exposure of PII, credentials, and proprietary code to third parties and aligning with evidentiary integrity requirements. It preserves chain of custody and data sovereignty by avoiding outbound transfers and cross-border processing, simplifying audits and legal defensibility. It eliminates cloud-side telemetry and retention risks, allowing strict egress controls and isolated execution within analysis VMs or enclaves with least-privilege access to artefact slices. It supports deterministic redaction and hashing workflows before inference, while maintaining the framework's conservative, corroboration-based verdicting without leaking intermediate findings to external providers.

5.4 Evaluation of automation effectiveness

The Automated framework was tested against artefacts collected from Virtual Machines infected with one of the samples described in Chapter 3 or not infected at all, to see the framework effectiveness of detecting post-mortem compromises in a machine.

All the referred reports in the next sections can be found in the github link alongside the Framework PoC at https://github.com/fuju/ForensicAutonomousFramework

5.4.1 Windows: Ludbaruma Infected Machine

Groud truth from AnyRun live sandbox reports

The Any.Run live sandbox analysis of the Ludbaruma sample provides a reliable ground truth for the Windows infection scenario with this malware, capturing both static identifiers and the dynamic behaviors that define its tradecraft on Windows. The submitted object is a PE32 GUI executable tagged as Ludbaruma and adjudicated "Malicious activity".

At runtime, the specimen dropped and executed multiple binaries named to mimic core Windows processes—WINLOGON.EXE, SERVICES.EXE, CSRSS.EXE, and LSASS.EXE in the user-scoped AppData Local WINDOWS directory, all bearing the same "Oncom" company string and 0.00.0020 version, signaling process masquerading as part of the execution chain annotated by the behavior graph with LUDBARUMA labels on the parent and the spawned WINLOGON.EXE. These faux system-named children inherited medium integrity under the "admin" account context and were associated with file system activity including executable drops, creation in user directories and temporary paths, and a failed attempt to create an executable under the Windows directory, with additional manipulation of desktop.ini that may serve to cloak folders.

Persistence is unambiguously established via registry-based mechanisms, prominently a screensaver hijack: the sample and its spawned WINLOGON.EXE wrote HKEY_CURRENT_USER/Control Panel/Desktop values SCRNSAVE.EXE to C:/WINDOWS/system32/Mig mig.SCR, ScreenSaverIsSecure to 0, and ScreenSaveTimeOut to 600, which Any.Run elevates as "The process uses screensaver hijack for persistence." Complementing this vector, the malware set multiple Run keys across HKCU and HKLM/-SOFTWARE/WOW6432Node to auto-start both a payload at C:/WINDOWS/xk.exe (value "xk") and the masquerading binaries via values MSMSGS, Serviceadmin, Logonadmin, and System Monitoring, pointing to the dropped WINLOGON.EXE, SERVICES.EXE, CSRSS.EXE, and LSASS.EXE under the user's AppData Local WINDOWS path, while policy hardening keys were abused to degrade visibility by writing NoFolderOptions=1 and DisableRegistryTools=1. Any.Run explicitly flags "Changes the autorun value in the registry," "Process was added to the startup," and "Executable file was dropped," aligning with a registry-centric autorun strategy that blends user and WOW6432Node locations to ensure reliable logon persistence and hinder remediation.

Network and interprocess communication observations show a local SMB named pipe interaction by WINLOGON.EXE via the srvsvc pipe with data transfer, a pattern often associated with lateral coordination or local service enumeration, though the standout behavior in this run remains the local persistence and masquerading rather than overt command-and-control; ancillary processes like slui.exe queried proxy configuration and software policy settings, and background activity from Outlook was captured but appears orthogonal to the core malicious chain. The behavior overview further lists "Known threat" and "Inspected object has suspicious PE structure," corroborating the family tag and the packed nature of the binary.

In sum, the sandbox-derived ground truth for the Ludbaruma infection on Windows demonstrates a compact, packed dropper that achieves persistence through an HKCU screensaver hijack and multiple autorun Run keys, deploys masquerading executables under user AppData with system-like names, performs desktop.ini manipulation suggestive of cloaking, and engages SMB srvsvc pipe communication, collectively producing a set of high-confidence Indicators of Compromise across disk, registry, and process artifacts that are consistent with dropper-led, registry-persistent malware on modern Windows hosts.

Autonomus Framework Final Report and Comparison

The autonomous framework reproduces core elements of the ground truth profile by extracting Run-key persistence for system-like payloads under user profile paths and identifying anomalous Startup artefacts indicative of auto-execution on logon. Specifically, the pipeline surfaces HKCU Run entries pointing to WINLOGON.EXE and SERVICES.EXE under Local Settings\Application Data\WINDOWS, mirroring the ground truth's masquerading and registry-based autorun strategy within the user context. In parallel, the Startup folder inspection flags an unexpected "Empty.pif" executable in the all-users Startup path as a strong persistence signal, expanding coverage beyond the registry and reinforcing multi-vector persistence detection in line with the dropper's objectives. The file was indeed a malicious payload as confirmed manually later after the review of the report with a virustotal sandbox scan, that can be consulted at www.virustotal.com/gui/file/5516e1764287a417ab2fb7710333d81fbcc4986c075583ccbecd0df8cd434786.

In volatile memory, the framework's Volatility3 workflow detects false positive caused by plugin malfunction: it wrongly address this as "process masquerading" consistent with the tradecraft evidenced

on disk, including truncated executables such as fontdrvhost.ex and VBoxService.ex, which emulate legitimate binaries while deviating in naming and location semantics. This happens because the 'psscan' plugin wrongly reconstructs the name of a process truncating it, in this case: "VBoxService.ex" or "fontdrvhost.ex".

Network telemetry enrichment via VirusTotal highlights at least one contacted IP adjudicated malicious by a vendor, providing a tentative external corroboration channel while acknowledging that Ludbaruma's standout behaviors in the ground truth are local persistence and masquerading rather than overt C2 in the observed window.

To highlight agreement and divergence, four comparisons are instructive.

- First, both the ground truth and the framework converge on user-scoped masquerading of system process names and Run-key autoruns, confirming the same persistence vector and naming scheme within the user profile hierarchy.
- Second, the framework surfaces an additional Startup artefact (Empty.pif) not enumerated in the ground truth narrative, plausibly reflecting environmental or temporal variation and illustrating the framework's breadth in persistence enumeration across filesystem loci beyond registry paths alone.
- Third, while Any.Run emphasizes screensaver hijack keys, these specific keys are not explicitly present in the provided framework outputs, delineating a coverage gap that motivates extending registry parsing to Control Panel\Desktop hives for high-recall persistence validation.
- Forth, the memory scan produced a false positive caused by a volatility plugin malfunction, misclassifying some legitimate processes as masquerading executables. This illustrates the need for cross-plugin corroboration and truncation-aware heuristics to avoid such artefacts being misreported as true malicious activity. This should be already addressed by the memory prompt part:

  ```
  Handle plugin malfunctions carefully: do not interpret them as malicious
  indicators.
  Examples:
  - Extremely large or non-standard values for PID, PPID, UID, GID, EUID, or EGID.
  - Corrupted or garbled process names.
  - Impossible timestamps (e.g., year 1930 or similar).
  ```

  as can be seen in A.4. These guidelines should help reduce false alarms, though in some cases the AI may still flag them incorrectly as can be seen here.

Effectiveness is anyway evidenced along three dimensions.

- **Accuracy** is demonstrated by reproducing core Ludbaruma IoCs across disk (user-path WINLOGON.EXE/SERVICES.EXE) and autorun mechanisms, yielding a final verdict of suspicious that coheres with the sandbox adjudication and behavioral expectations for a dropper maintaining logon persistence.
- **Coverage** iis achieved through the concurrent harvesting of Startup folders, Run keys, Winlogon context, memory process scans, and network IoC triage, enabling cross-artefact triangulation that mitigates false negatives common in single-source workflows.
- **Actionability** is reflected in concrete next steps emitted by the pipeline—process dumping, artefact isolation, and IoC-driven network scoping—which shorten the gap between detection and investigative escalation in post-mortem operations.

In summary, when evaluated against the Any.Run ground truth, the framework reliably detects Ludbaruma's hallmark persistence and masquerading behaviors and augments the picture with Startup-based persistence findings, thereby validating the automation approach while revealing precise opportunities to expand registry hive coverage for screensaver hijack and WOW6432Node keys. This outcome supports the thesis claim that IoC-focused, AI-assisted automation can improve efficiency and consistency in malware post-mortem analysis on modern Windows hosts.

5.4.2 Windows: Conhoz Loader Infected Machine

Groud truth from AnyRun live sandbox reports

The Any.Run live sandbox execution of conhoz.exe on Windows establishes a clear ground truth of a UPX-packed loader that performs system reconnaissance, aggressive persistence via Task Scheduler and

WMI, security evasion through Windows Defender exclusions, service and firewall manipulation with netsh/sc, and multi-endpoint HTTP communications indicative of payload staging and C2 checks. The sample is a PE32 console executable compressed with UPX, tagged as a loader with "evasion" traits.

The malware spawns a dense chain of living-off-the-land binaries to assert control, persist, and tamper with system configuration. It creates and removes numerous scheduled tasks, including the creation of "MyTask" to execute C:/Windows/Temp/onhoz.exe as SYSTEM every 5 minutes and "oka" to run C:/Windows/inf/aspnet/lsma22.exe as SYSTEM every 3 minutes, alongside multiple deletions of tasks named in Windows-like patterns (e.g., "WindowsUpdate1/3," "MicrosoftsWindows[u/z/a]," "Update2/3/4"), indicating both installation and cleanup of autorun mechanisms. It further issued schtasks /run on "oka," confirming activation of the scheduled payload. Complementing Task Scheduler, the specimen exercised WMI-based persistence and cleanup in the root/subscription namespace by adding and deleting __EventFilter, CommandLineEventConsumer, ActiveScriptEventConsumer, and FilterToConsumerBinding objects (e.g., deleting consumers "killmm3/killmm4"), a hallmark of stealthy autorun and tasking that survives restarts and user context changes.

Security and service posture were actively manipulated to reduce detection and enable payload execution. The process executed PowerShell to add broad Windows Defender exclusion paths that cover Temp, Debug, Public Pictures, various INF/aspnet and system directories, SysWOW64 caches, and service profile Temp directories, a move that materially degrades on-access scanning in the most likely drop zones. It used IFEO (Image File Execution Options) to set Debugger values for uihost32.exe, uihost64.exe, and vid001.exe, a technique discussed in chapter 2, and attempted ACL changes via cacls/icacls to protect scripts and block wscript/cscript execution, suggesting anti-remediation and script handling controls. Service management via sc.exe included configuring a service xWinWpdSrv and attempts to start it, consistent with service-mode persistence even if the referenced service was absent at the time (Exit code 1060).

Network stack controls were tuned through netsh and IPsec policy manipulation in ways that align with lateral movement gating and exfiltration facilitation. The sample created and assigned an IPsec "BlockPorts" policy with filter lists and rules affecting TCP ports 135 and 445 (add/delete mirrored filters, create/delete firewall rules "deny tcp 139/445," and broad "tcp all" allow rules), toggled firewall state across profiles, and added/deleted custom rules, indicating dynamic reconfiguration to both restrict detection vectors and maintain command channels. The command flow also included repeated PING delays to sequence operations and TASKKILL invocations targeting mssecsvc.exe and named payloads, implying either cleanup of competing malware or staged updates.

System reconnaissance and environment shaping steps were evident throughout execution. Conhoz.exe read machine GUID, computer name, product name, environment variables, and Internet Settings, repeatedly touching WPAD keys, zone mappings, proxy enablement, and legacy connection settings, including writing and deleting values under HKCU/Software/Microsoft/Windows/CurrentVersion/Internet Settings and associated WPAD subkeys, which influences outbound routing and detection evasion. The program created files and folders under user directories and Temp, executed batch files (wmia.bat), and hid command output, while issuing multiple WMIC queries that enumerate and prune processes (including a predicate to delete svchost.exe instances with unexpected paths), demonstrating both situational awareness and suppression of suspicious host artifacts.

Overall, the sandbox ground truth for conhoz.exe typifies a modern UPX-packed Windows loader: it seeds persistence through both scheduled tasks (SYSTEM context, frequent intervals) and WMI consumers/filters, disables or sidesteps protections via Defender exclusions and ACL edits, prepares service-based hooks, and actively reshapes the firewall/IPsec surface, while establishing multiple external HTTP touchpoints that trigger IDS signatures associated with executable delivery and scripted tasking. These observations yield high-confidence IoCs across binary hashes, scheduled task names and paths, WMI subscription artifacts, Defender exclusion registry usage via PowerShell, IFEO Debugger keys, and specific external IPs/domains contacted during the run.

Autonomus Framework Final Report and Comparison

The autonomous framework reproduces key elements by extracting a cluster of suspicious tasks (e.g., *Mysa1*, *Mysa2*, *MyTask*, *ok*, *oka*) and confirming an operational task that executes the malicious file `C:\windows\temp\conhoz.exe` as *SYSTEM* every 5 minutes, evidencing scheduled task–based autorun and payload execution. It further detects security control tampering by identifying removed triggers from the *Windows Defender Scheduled Scan* task on the incident date, consistent with defense evasion highlighted in the sandbox, while additional anomalies (e.g., *RegisterDeviceWnsFallback* timestamp alignment) support a coordinated configuration phase.

Volatility memory analysis augments attribution: an anomalous parent-child relationship where a process called `svchost.exe` spawns `ctfmon.exe` suggests process abuse in a service context, Moreover, svchost.exe instances holding handles to the IFEO registry hive suggest IFEO-based hijack preparation or persistence, reinforcing the sandbox's IFEO narrative through memory evidence. These findings tie autorun configuration to runtime manifestations, enabling cross-artefact reasoning about loader lifecycle and execution surfaces.

Network enrichment via VirusTotal shows at least one contacted IP with a malicious verdict by a vendor, partially aligning with the sandbox's multi-host HTTP telemetry and IDS detections despite dataset emphasis on benign Microsoft/CDN infrastructure, and reflecting the loader's tendency to pair local foothold with opportunistic HTTP staging.

In summary:

- Both analysis confirm Scheduled Task persistence with explicit execution paths and SYSTEM context, validating task parsing fidelity.
- Both attest to IFEO involvement: the sandbox via Debugger keys and the framework via IFEO handle activity in memory, indicating the same evasion/persistence vector from distinct evidence planes.
- The sandbox documents WMI subscriptions and Defender exclusion keys, which are not yet enumerated by the framework output, motivating module extensions for WMI repository parsing and Defender policy artefact extraction to improve recall on stealthy persistence and AV evasion.

Overall, the framework demonstrates accuracy by reproducing core loader IoCs, breadth by correlating disk, memory, and network artefacts, and actionability via concrete response steps (isolation, payload and memory dumps, IFEO audit, network correlation), thereby validating the automation approach against the Any.Run ground truth while revealing specific opportunities to expand WMI and Defender artefact coverage.

5.4.3 Windows: Not Infected Machine

In the report of a not infected windows machine analysis, the disk and task artefacts match a non-infected Windows baseline: user/system Startup folders contain only standard `desktop.ini`, registry `Run/RunOnce` keys list benign Microsoft OneDrive entries, and `\Windows\System32\Tasks` enumerations (e.g., `XblGameSaveTask`) are consistent with stock services without anomalous names or timestamps. This convergent absence of persistence IoCs across multiple loci corroborates the ground truth of a clean system.

Network reputation checks predominantly return harmless for Microsoft CDN and telemetry endpoints, with a single IP (`150.171.22.17`) flagged by one vendor, an expected low-confidence outlier for large CDNs absent any process or flow attribution. In a clean baseline, such solitary reputation noise should not influence the overall classification without corroborating evidence.

Memory analysis reported two anomalies for `svchost.exe` (PID 1648): duplicate `_EPROCESS` records via `psscan` and two handles with code-like names, which the report interprets as potential DKOM and injection. Yet, without confirmatory `pslist` concealment, `malfind/dlllist/vadinfo` indicators, or process-bound network correlations, these findings are low-specificity and may reflect stale structures or handle decoding artifacts common in live captures.

Three overstatements are evident.

- DKOM is inferred from duplicate PIDs without validating hidden status or memory region anomalies, exceeding evidential support in a clean context.
- The "code-like" handle names are elevated without verifying handle types, owning services, or alternative parsing explanations, risking misclassification of benign artifacts.
- The final verdict suggests linkage to a weak VirusTotal flag despite no process attribution, illustrating narrative drift beyond available data.

Taken together, the framework shows strong precision in disk artefacts and scheduled tasks, as well as appropriate network whitelisting. However, it disproportionately elevates ambiguous memory heuristics to a "Suspicious" verdict, even in a clean-host scenario. Introducing a correlation threshold—requiring consistent indicators across at least two artefact classes before escalating severity—would better align reporting with the clean ground truth and mitigate LLM-style hallucinations in baseline comparisons. Although this correlation threshold is already enforced at the prompting stage, the AI occasionally still overstates findings, leading to LLM-style hallucinations despite the safeguard.

5.4.4 Linux: Prometei Infected Machine

Groud truth from AnyRun live sandbox reports

The Any.Run live sandbox execution of the Linux ELF sample on Ubuntu 22.04.2 provides authoritative ground truth consistent with Prometei botnet behavior, capturing both static and dynamic identifiers. The binary is identified as an ELF 64-bit LSB executable for x86-64, statically linked and notably lacking a section header, a trait often observed in minimally structured droppers; with tags "prometei" and "botnet.

Dynamic behavior begins with privilege preparation and service-based persistence, aligning with Prometei's known emphasis on durable footholds for resource hijacking. The launcher chain employs sudo to change ownership and mark the ELF executable, then executes the payload under root, after which the specimen interacts intensively with systemd: daemon-reload, enable, and start operations target a unit named uplugplay.service, culminating in the creation of /usr/lib/systemd/system/uplugplay.service and the invocation of /usr/sbin/uplugplay as a persistent service; Any.Run explicitly flags "Writes to Systemd service files (likely for persistence achievement)" and records subsequent starts of uplugplay with the -Dcomsvc parameter, affirming service-managed persistence in the host OS. The behavior summary also notes "Process was added to the startup" and "Process starts the services," reinforcing the operational pivot from a transient dropper to a resident service executing under root.

Process and host-enumeration activity are pervasive and consistent with a botnet client staging for mining and command reception. The specimen and its service component issue repeated environment checks—hostnamectl, uptime, uname -a—and probe for the presence of process names such as uplugplay and upnpsetup via pgrep and pidof, suggestive of self-monitoring and conflict avoidance routines common to long-running Linux bot clients. It reads /etc/hosts, touches profile files, and records a small operational footprint in /etc/CommId, while Any.Run lists six malicious and two suspicious processes overall during this task, with uplugplay singled out by both heuristic and network-based detections as the primary malicious component. The service's execution context is root with unknown integrity level, amplifying the potential for system modification and network reconfiguration beyond user space.

Network telemetry strongly corroborates Prometei command-and-control patterns and infrastructure, including detections tied to the family's "xinchao" motif. The uplugplay service issues HTTP GET requests over cleartext to multiple external endpoints: 152.36.128.18 at /cgi-bin/p.cgi with parameters including base64-encoded host inventory and environment descriptors; 18.142.91.111 at xinchaocccfea.net /cgi-bin/p.cgi with auth and enckey parameters, marked malicious; and 85.214.228.140 at xincccfea.org /cgi-bin/p.cgi returning 404, all consistent with multi-endpoint C2 or fallback logic. The debug output logs repeated Suricata alerts "ET MALWARE Prometei Botnet CnC DGA - xinchao Pattern," as well as "Possible Compromised Host AnubisNetworks Sinkhole Cookie Value" indicators, strengthening attribution to Prometei and indicating potential interaction with sinkholed infrastructure. Baseline whitelisted traffic to connectivity-check.ubuntu.com and api.snapcraft.io is also recorded, providing environmental context and preventing false attribution of benign OS telemetry to the malware.

Taken together, the sandbox-derived ground truth shows a Prometei Linux implant that establishes persistence via a systemd unit (uplugplay.service) and runs a root-owned backdoor client (uplugplay) which conducts host reconnaissance and engages in HTTP communications with infrastructure patterned around "xinchao," triggering IDS signatures specific to the Prometei botnet. These observations yield high-confidence Indicators of Compromise spanning binary hashes, the systemd service path/name (/usr/lib/systemd/system/uplugplay.service; /usr/sbin/uplugplay), the on-host artifact /etc/CommId, and specific external IPs and domains (152.36.128.18, 18.142.91.111 at xinchaocccfea.net, 85.214.228.140 at xincccfea.org) with characteristic /cgi-bin/p.cgi URLs, forming a comprehensive basis for disk, memory, and network detection within Linux-focused forensic workflows.

Autonomus Framework Final Report and Comparison

The autonomous framework corroborates this profile by identifying a root-executed `uplugplay` process chain in memory (`uplugplay` into `sh` into `uplugplay`), with parentage from `systemd` (PPID 1), which is emblematic of service-managed daemonization and parent detachment, thereby confirming a resident implant rather than transient execution; process lineage and timestamps in `pslist` further align with the sandbox's persistence narrative, enhancing attribution fidelity for the observed runtime.

Disk artefacts explicitly surface the persistence mechanism: the unit file located in the following path `/usr/lib/systemd/system/uplugplay.service` configured to launch `/usr/sbin/uplugplay`, alongside a clean baseline across system cron and systemd directories otherwise showing only standard units and

VMware/Snap dependencies, which concentrates suspicion on the uplugplay pair as the single anomalous service footprint. In parallel, `/etc/profile.d/debuginfod.sh` and `/etc/profile.d/debuginfod.csh` contain terminal lines that clear `DEBUGINFOD_URLS`, which the analysis classifies as deliberate defense evasion rather than a benign variance, because the override unconditionally disables debuginfod and is inconsistent with the scripts' prior logic and typical packaging defaults.

Network enrichment flags multiple Canonical/Snapcraft endpoints as benign while isolating three IPs with vendor-malicious verdicts (185.125.190.18, 185.125.190.27, 185.125.188.57), furnishing an external corroboration channel that can be temporally correlated with the active `uplugplay` runtime to strengthen C2 attribution despite the absence of family-specific URL motifs in this dataset. Extending URL-path extraction, high-port correlation, and content-signature matching would further improve recognition of Prometei-style patterns in future runs without relying solely on reputation tallies.

Three comparisons emphasize convergence and residual gaps.

- Both sources confirm a root-owned `uplugplay` process tree anchored to `systemd`, validating volatile attribution against service-based persistence and establishing a coherent disk-to-memory linkage for the implant.
- The framework enumerates the exact unit and binary paths: (`/usr/lib/systemd/system/uplugplay.service`, `/usr/sbin/uplugplay`) and flags a false positve debuginfod script sabotage as defense evasion.
- Family-specific C2 markers (for example, "xinchao" and `/cgi-bin/p.cgi`) remain absent in these outputs, motivating module extensions for HTTP path extraction and signature matching to reach parity with sandbox detections when such telemetry is available.

Overall, the framework demonstrates accuracy by reproducing the implant's runtime identity and `systemd` lineage, breadth by correlating disk persistence, in-memory execution, and reputation-backed network endpoints, and actionability through concrete next steps (isolation, hashing and reversing of the malicious binary `/usr/sbin/uplugplay`, targeted memory dumps for PIDs 1597/1599, and time-aligned flow correlation), thereby validating the automation approach while identifying focused enhancements for Linux persistence enumeration and C2 pattern reconstruction in subsequent iterations.

5.4.5 Linux: Gh0st RAT Infected Machine

Groud truth from AnyRun live sandbox reports

The Any.Run live sandbox execution of the Linux ELF sample on Ubuntu 22.04.2 establishes a clear ground truth consistent with Gh0st-family Remote Access Trojan behavior, including explicit Suricata detections for GH0ST C2 traffic and outbound communications to infrastructure flagged as malicious. The sandbox tags the specimen as "remote," "rat," and "gh0st," and classifies the verdict as "Malicious activity," providing a robust static and behavioral fingerprint for downstream IoC correlation.

Dynamic execution shows privilege-oriented setup followed by service-based persistence consistent with a Linux RAT installing itself as a systemd-managed client. The launcher sequence uses sudo to change ownership and permissions of the ELF, then invokes the payload under root, after which the program interacts with systemd to daemon-reload, enable, and start a unit named remote-client.service, with corresponding file writes to /etc/systemd/system/remote-client.service and the creation of runtime generator artifacts characteristic of systemd activation workflows; Any.Run flags "Writes to Systemd service files (likely for persistence achievement)" with activity attributed to systemd and sudo during the run, indicating high-confidence service persistence on Ubuntu targets.

Process and file activity further corroborate RAT staging and operational logging. The behavior pages enumerate the creation of /home/user/Desktop/launch_time.txt and /home/user/Desktop/logs/remote-client.log by the ELF process, suggesting explicit time-stamping of execution and operational telemetry under a logs directory, while the behavior graph notes "Process was added to the startup," "Executable file was dropped," and "Task has injected processes," collectively aligning with a service-managed persistence and possible in-memory tasking patterns often observed in RAT families. The task recorded 290 total processes with 71 monitored and one marked malicious, reflecting the instrumented Ubuntu environment and the malware's service-related interactions, including repeated systemd generator invocations during enable/start cycles of the remote-client unit.

Network telemetry captures both benign baseline contacts and Gh0st-typical C2 indicators. Alongside whitelisted Canonical and Snapcraft endpoints used by the OS, the ELF issued a successful GET to https://a87.mee333.com/api/get_client (103.215.78.27) and established a TLS connection to the socket

118.107.46.162:5650 for lc.liuddiase1li.com, the latter marked "malicious" with a nonstandard high port fitting RAT command channels; the debug output includes multiple "ET MALWARE Backdoor family PCRat/Gh0st CnC" detections and "Go-http-client" user-agent observations, consistent with a Go-based networking stack leveraged by contemporary RAT reimplementations. Any.Run summarizes the sample as "Connects to the CnC server," "Connects to unusual port," and "Contacting a server suspected of hosting a CnC," providing both destination IoCs and protocol/port characteristics for network-based detection and blocking.

Environmental checks and system shaping behaviors support evasion and stability of the installed service. The malware process checks timezone, reads network configuration via sudo, and issues systemctl is-active and start commands for remote-client.service, indicating logic to verify and maintain service state post-install; additional shell utilities (df, cat, tr, mesg) appear in the chain as part of environment enumeration and scriptable control. While the sandbox reports "No Malware configuration" in the static pane, the runtime C2 beacons and service file creation strongly indicate a configuration retrieved or embedded at runtime via the /api/get_client request path, a pattern consistent with RATs that bootstrap tasking from remote profiles after initial execution.

Taken together, the sandbox ground truth for this ELF sample demonstrates a Gh0st-like Linux RAT that achieves persistence by installing a systemd service (remote-client.service), logs execution artifacts to user-accessible paths, and communicates with external infrastructure including a87.mee333.com and lc.liuddiase1li.com over HTTPS and an unusual TCP port 5650, respectively, generating Suricata signatures tied to PCRat/Gh0st families. These observations yield high-value IoCs across binary hashes, the systemd unit file path/name, runtime log and timestamp file paths, and specific domains, IPs, and ports contacted, forming a comprehensive basis for disk, memory, and network detection in Linux forensic workflows.

Autonomus Framework Final Report and Comparison

The framework corroborates service persistence by detecting a newly created service located in the path `/etc/systemd/system/remote-client.service` and near-immediate modification of the trigger file `multi-user.target.wants`, indicating enablement at boot in line with the sandbox's "Writes to Systemd service files" and startup behavior. Parallel inspection of `/usr/lib/systemd/system` and user-scope service directories distinguishes stock units from attacker-controlled persistence, localizing the implant's foothold to the system scope.

Memory analysis identifies a root-owned process `b560f76f7603e3e` (PID 763) with PPID 1 (`systemd`) and a start time shortly after boot, which is emblematic of unit-managed daemonization consistent with a RAT client. Environment and netfilter analyses contextualize the process within a standard Ubuntu GNOME/VirtualBox session and show no kernel-hook tampering, supporting a user-space implant model rather than a rootkit. Bash-history reconstruction separates benign analyst actions (LiME build and `insmod`) from the suspicious service-run process, improving timeline attribution.

Network enrichment reports mixed reputations, including malicious verdicts for `lc.liuddiase1li.com` and related domains amid benign Canonical/Snapcraft traffic, offering external corroboration while underscoring the need for UA/port-aware filtering to disambiguate RAT beacons from OS telemetry. The absence of explicit `/api/get_client` path and TCP 5650 correlation in current outputs motivates extending URL-path extraction, high-port correlation, and TLS-fingerprint capture to fully mirror Gh0st C2 signatures. So in comparison to the ground truth sandbox report:

- Both sources confirm systemd-based persistence via `remote-client.service`, validating unit detection and enablement timing.
- Memory forensics exposes a hex-named root process under `systemd`, strengthening runtime linkage between disk persistence and execution beyond static evidence.
- The sandbox documents exact C2 paths and ports, whereas the framework presently surfaces domain-level verdicts without path/port reconstruction, motivating module extensions for URL, port and TLS-aware telemetry.

Overall, the framework demonstrates accuracy by reproducing the RAT's persistence and runtime identity, breadth by correlating disk, memory, network, and anti-analysis artefacts, and actionability through concrete next steps (unit inspection, binary recovery, and C2 correlation), validating the automation approach against Gh0st ground truth while identifying precise enhancements for URL-path and port-aware C2 detection.

5.4.6 Linux: Not Infected Machine

In the report of a not infected linux machine analysis, the disk baseline is consistent with a non-infected Ubuntu system: `/etc/cron.d` contains only standard maintenance entries (e.g., `e2scrub_all`, `anacron`), `/etc/systemd/system` presents expected `.wants`/`.requires` symlinks and no anomalous units, and `/usr/lib/systemd/system` reflects stock multi-user services with coherent metadata; user-scope autoruns likewise show only benign Snap desktop integration. These convergent findings indicate absence of service-, timer-, or user-scope persistence and align with the clean ground truth.

Memory-resident network controls show integrity: Netfilter hooks resolve to the in-kernel symbol `apparmor_ip_postroute` with *Is Hooked* set to false across namespaces, providing no evidence of redirection or packet-hiding consistent with kernel-level malware. Environment-variable inspection characterizes a VirtualBox-hosted Ubuntu GNOME Wayland session for user `vboxuser` with standard `PATH` and locale configuration and without suspicious variables (e.g., `LD_PRELOAD`), supporting a benign runtime context.

Two anomalies merit cautious interpretation. First, the same false-positive as discussed before `/etc/profile.d/debuginfod.sh` and `debuginfod.csh` end with statements that clear `DEBUGINFOD_URLS`; while flagged as anti-analysis, this remains a configuration variance until confirmed against package baselines (`dpkg --verify`) or change history, and is insufficient alone to infer compromise. Second, `kworker/u18:4` appears in `pslist` with implausible credentials; in the absence of corroborating indicators (suspicious modules, malfind hits, hooked netfilter), plugin parsing artifacts are a more likely cause in a clean image.

Relative to the ground truth, the final narrative overweights isolated, low-specificity signals. Elevating the debuginfod override to defense evasion without provenance checks risks false positives in managed or lab systems. Similarly, presenting the kworker credentials as DKOM-adjacent while conceding probable parsing error contributes to an "Inconclusive" verdict despite broad evidence of cleanliness, indicating a need to may recalibrate severity thresholds other than the already present prompt safeguards against this.

5.5 Evaluation for Local Models

5.5.1 Setup of the test-bed via polito HPC

The successful testing of local models was made possible thanks to the PoliTo HPC infrastructure[41]. This setup enabled experiments with high-end models, such as llama3.1:70b on an A100 GPU, as well as fast inference with qwen3:30b and gemma3:27b on A40 GPUs.

The computing node that used for hosting the Ollama local model was not directly accessible from outside the cluster and could only be reached through the login nodes. As a result, I was unable to expose an API port from the computing node to my local machine—even attempts with SSH tunneling were blocked by the firewall. Consequently, the only viable solution was to install and run the framework directly on the computing node. Installing the framework on the computing node posed some challenges. I could not use apt or sudo, and the default Python version available was 2.7, whereas my framework required Python 3.11. Initially, I attempted to install all binaries locally, but encountered issues with certain C libraries—most notably with the pytsk3 dependency. To overcome this, I chose a hybrid solution: I installed only the Ollama binaries directly on the computing node, while running the rest of the framework inside a Docker image I had prepared on my laptop. This image, containing all precompiled dependencies, was executed on the computing node through Apptainer.

With this setup, I was able to successfully test the framework with local models. The DockerFile to build the image, can be found in the thesis github repository:
https://github.com/fuju/ForensicAutonomousFramework

5.5.2 Comparison with Gemini

The reports generated by the local model produced results that were largely comparable to those obtained with Gemini, though with some notable limitations. Due to its comparatively simpler architecture, the local model occasionally misclassified modification timestamps as occurring in the future. This issue persisted even when the model was explicitly prompted with contextual information about the current date and provided with examples of how to determine whether a date lies in the future. Additionally, the

local model introduced a small number of false positives; however, these did not significantly compromise the overall quality or reliability of the reports.

The complete set of reports generated with the local model is available in the associated GitHub repository https://github.com/fuju/ForensicAutonomousFramework.

For these experiments, the local model employed was "dengcao/Qwen3-30B-A3B-Instruct-2507", whose configuration is summarized below:

- Model architecture: qwen3moe
- Parameters: 30.5B
- Context length: 262,144
- Embedding length: 2,048
- Quantization: Q4_K_M

Capabilities
- Completion
- Tools
- Reasoning

Experimental parameters
- Context window: 31,072
- Repeat penalty: 1
- Stop sequences: "<|im_start|>", "<|im_end|>"
- Temperature: 0.1
- Top-k: 20
- Top-p: 0.95

5.6 Summary overview

The PoC framework effectively automates cross-artifact IoC discovery on Windows and Linux, reproducing sandbox-derived tradecraft while identifying targeted coverage gaps; its AI-assisted reporting accelerates analysis but requires stricter corroboration thresholds to avoid over-weighting isolated anomalies in clean baselines or false positives. Cross-platform strength is evident in registry and scheduled-task persistence detection on Windows and systemd-backed persistence on Linux, with memory–disk–network correlation improving attribution and actionability.

Chapter 6

Conclusions and Future Work

This thesis demonstrated that an IoC-centric, cross-artifact workflow—spanning disk, memory, and network—combined with automation and carefully constrained LLM assistance, can materially improve the efficiency, consistency, and actionability of malware post-mortem forensics across Windows and Linux systems. The proof-of-concept pipeline reduced manual toil, produced structured evidence syntheses, and enabled conservative, corroboration-based verdicts, while exposing concrete avenues to strengthen precision and coverage in operational settings.

6.1 Interpretation of Results

The results validate three core contributions:

- A systematic methodology for correlating Indicators of Compromise (IoCs) across disk images, memory dumps, and network captures for both Windows and Linux.
- An automated pipeline that acquires, processes, and consolidates artefacts into a unified report.
- An evaluation of AI-assisted reasoning that balances efficiency with privacy by comparing local and cloud models under prompt safeguards and chunking constraints.
  - In Windows scenarios, the pipeline consistently surfaced registry- and task-based persistence, IFEO-related signals, and startup artefacts.
  - In Linux scenarios, it detected systemd-backed persistence and aligned memory/runtime observations.
  - VirusTotal-enriched network observables provided external corroboration when available.

The integrated reporting favored cross-domain agreement—requiring persistence evidence to align with runtime and network traces for escalated verdicts—thereby improving calibration against both infected and clean baselines, though occasional overstatement in memory heuristics underscores the value of stricter correlation thresholds and plugin-aware caveats already introduced in the prompting layer.

Operationally, automation converted many "for centuries manual" steps into a reproducible workflow that preserved chain-of-custody principles, logged decisions, and produced machine-parsable reports for downstream review; this yielded meaningful time savings and reduced variance in triage quality across cases and artefact types. AI assistance, when constrained by deterministic schemas, rate-limit controls, and domain guardrails, enhanced narrative cohesion and anomaly prioritization across long, heterogeneous outputs; local models provided privacy benefits for sensitive artefacts, whereas cloud models offered robust long-context behavior and stability, highlighting an actionable trade-off envelope for forensic teams.

6.2 Limitations

Despite promising results, several limitations remain. First, the framework showed incomplete detection coverage in specialized Windows artefacts—such as WMI subscriptions, Defender policy artefacts, screensaver hijacks—as well as in Linux command-and-control (C2) telemetry (e.g. TLS fingerprints),

indicating the need for expanded parsers and richer network feature extraction beyond domain/IP reputation. Additionally, detection coverage remains incomplete for disk and memory-based malware persistence, suggesting the integration of techniques such as YARA rule-based scanning for the disk module and reference outputs from clean systems for the memory module. Network analysis could also benefit from Suricata/Snort-based traffic inspection to detect a broader range of malicious behaviors.

Second, memory analysis precision can be affected by malformed plugin artifacts (e.g., truncation or duplicate EPROCESS in psscan), which, if not counterchecked, may bias verdicts; while prompt guardrails mitigate this, code-level cross-plugin heuristics would further reduce false positives in clean baselines.

Third, the LLM layer—especially local models with smaller or slower contexts—occasionally misjudged false future date anomalies or elevated weak signals; although strict schemas, temperature control, and context chunking improved determinism, residual variability motivates additional post-inference validators, correlation thresholds, or specialized LLM tools before verdict escalation.

Privacy and deployment constraints also shape applicability. Cloud inference raises data sovereignty and evidentiary exposure risks that many forensic cases cannot accept, whereas local inference demands GPU capacity and model curation. Future deployments may instead rely on secure offline environments or automated data-sanitization processes that consistently remove sensitive information before analysis.

Finally, while Any.Run-sourced malware traces and clean VMs provide grounded and reproducible scenarios, broader validation across malware families, versions, and live enterprise telemetry would better quantify generalization and robustness at scale. Furthermore, the framework could benefit from multi-role report generation, dynamically adapting structure and technical detail according to the intended audience—technical analysts, investigators, or managerial stakeholders—to improve usability and decision-making within forensic teams.

6.3 Future Work

Several improvements are immediately actionable and define the pathway toward a more comprehensive, scalable, and explainable forensic automation framework.

On the disk side, as said before, extending Windows coverage to WMI repository and subscription parsing, Defender exclusion and policy extraction, and screensaver hijack detection will address current blind spots. Similarly, Linux telemetry coverage—particularly in command-and-control (C2) traces such as URL path, port profiling, and TLS/JA3 fingerprinting—requires further expansion. These additions would bridge the incomplete detection coverage observed in the experiments and support broader generalization across different persistence and evasion mechanisms.

On the network side, integrating richer context—such as URL path analysis, TLS fingerprinting, and lightweight flow features—alongside PCAP-to-process attribution, Suricata/Snort-based inspection, and expanded port profiling will raise the precision of network traffic scans beyond basic domain/IP reputation. This will enable the framework to detect more subtle and diverse malicious behaviors consistent with real-world RAT and C2 operations.

On memory analysis, constructing correlation heuristics that fuse plugins deltas with baseline outputs from non-infected systems and module signature validation can optimize the memory analysis process. This will reduce false positives and strengthen the AI model's ability to distinguish genuine anomalies from routine system activity.

On automation and reporting, the inclusion of YARA-based scanning for recovered binaries and STIX/TAXII export will further enhance CTI interoperability and downstream threat-hunting capabilities. Moreover, the introduction of multi-role report generation, where output complexity and terminology adapt dynamically to the audience (technical analysts, investigators, or managerial staff), can significantly improve communication, situational awareness, and decision-making within forensic teams.

For the AI layer, continued evaluation of local versus cloud LLMs—focusing on distilled, long-context local models and structured few-shot exemplars—can preserve privacy while enhancing stability. Nonetheless, the observed variability in inference quality, particularly under limited context windows or high load, suggests the need for additional post-inference validators, strict schema enforcement, and specialized LLM-based tools to ensure reliability. Incorporating clean-system reference data could also guide the LLM toward context-aware differentiation between benign and anomalous patterns.

In terms of validation and scalability, extending the experimental scope to encompass a wider spectrum of malware families, versions, and enterprise telemetry will be crucial to assess robustness and generalization at scale. Leveraging containerized and HPC-enabled workflows—as already explored for local LLMs

under Apptainer—can support large-batch investigations, reproducible experiments, and controlled data egress compliant with enterprise Digital Forensics Incident Response (DFIR) requirements.

Finally, beyond technical refinement, future work should include usability and transparency studies, examining how AI-assisted automation can be integrated into operational forensic pipelines while maintaining evidentiary rigor and auditability. Together, these directions form the foundation for a more interoperable, resilient, and human-aligned digital forensics ecosystem capable of meeting the evolving challenges of modern cyber investigations.

6.4 Closing Remarks

By centering the investigation on IoC correlation across disk, memory, and network, and by pairing automation with principled AI assistance, this work shows a practical path to faster, more reliable post-mortem malware forensics that respects evidentiary rigor while embracing scalable analysis patterns. The framework's strengths—reproducibility, cross-domain synthesis, and privacy-aware AI—provide a foundation for broader adoption; the identified gaps define a clear roadmap toward higher recall, stricter precision, and richer attribution in real-world incident response.

Appendix A

Appendices

A.1 User Manual

To use the framework, the easieast way is to use the DockerFile provided in the related thisis github: https://github.com/fuju/ForensicAutonomousFramework.

First, clone the github framework with the following command:

```
git clone https://github.com/fuju/ForensicAutonomousFramework
```

Then, edit the main framework configuration (Framework/config.py) file with your needs, in particular:

```
OLLAMA_MODEL_FAST : "name of the ollama model to use for simpler tasks like summarizing"
OLLAMA_MODEL : "name of the main ollama model used for the thinking agent and more complex tasks"
OLLAMA_BASE_URL = : "url for contacting the ollama API, default is http://0.0.0.0:11434"

GEMINI_MODEL_NAME = "name of the cloud model to use, default is gemini-2.5-pro"
```

Important: If you want to analyze Linux artefacts, you NEED to provide the correct symbol table. To do so, just put it in the DockerFile folder under the name of: "linux_symbol_table.json.xz" To find or generate the correct symbol table, refer to the section of the thesis 4.2.2.

To build the image you need to install docker. Open a terminal and navigate in the folder with the dockerFile, then use the command:

```
docker build -t autonomus_framework:latest .

If you are on Mac or any other arch:
docker buildx build --platform linux/amd64 -t autonomus_framework:latest .
```

This will generate the docker image ready to be used in a container.

Then to interact with it, use the command:

```
docker run -it autonomus_framework:latest bash
```

I reccommend to bind the artefacts folder into the docker image, to do so, use the command:

```
docker run -it -v /path/to/artefacts:/app/artefacts autonomus_framework:latest bash
```

this will bind your artefacts folder into the container. Note: use this exact file names for the different tipes of artefacts:

- Disk image: 'hdd_image_flat.vmdk'
- Memory dump: 'memdump.lime' or 'memdump.raw'
- Network scan: 'net_traffic.pcap'

If you want to do permanent changes to your config file or have problems with containers speed or space, bind also the FrameWork into the container:

```
docker run -it -v /full/path/to/artefacts:/app/artefacts \
-v FrameWork:/app/FrameWork \
autonomus_framework:latest bash
```

Note: The "-v FrameWork:/app/FrameWork" command **does not** bind a host folder — it creates a named Docker volume called FrameWork. That lives inside Docker's storage, not your filesystem. You can check the mountpoint of the volume with the command:

```
docker volume inspect FrameWork
```

Once inside the container, export your API keys into those ENV variables (e.g with the command "export GEMINI_API_KEY=¡your_api_key¿):

```
GEMINI_API_KEY : Gemini API keys, up to 4 different keys
                 (they will rotate if you arrive at you key quota).
GEMINI_API_KEY_2
GEMINI_API_KEY_3
GEMINI_API_KEY_4

VT_API_KEY : VirusTotal API key.
```

To create a Gemini API key, follow this guide: https://ai.google.dev/gemini-api/docs/api-key?hl=it
To create a VirusTotal API key, follow this guide: https://docs.virustotal.com/v2.0/reference/getting-started

if you want to use signing, put your private and public PEM keys in the folder:
`FrameWork/certificates/private_key.pem` and `FrameWork/certificates/public_key.pem`

To start the full analysis, go into the FrameWork folder and activate the virtual environment:

```
cd FrameWork/
source venv/bin/activate
```

and execute the command:

```
python3 createcase.py  <case_prefix> <case_name> ../path/to/artefacts/folder/ <os_type>
```

additional optional flags are:

```
--signing-key path/to/private_key.pem (e.g certificates/private_key.pem)
--verify-key path/to/public_key.pem (e.g certificates/public_key.pem)
-y : automatically start the processing without asking prompt
--local : If set, use ollama models instead of gemini
--skip_copy : skips the copy of the artefacts into the case directory
--no_hash : skips the hash and manifest phase
```

The framework will then generate the report in the following folder:

```
FrameWork/<os_type>/<case_prefix>_<case_name>/04_reports/final_report.md
```

A.1.1 Guide for Polito HPC

To use the framework with the PoliTo HPC, you need first to save the image as a .tar file and upload it to the login node. To do this use the following commands:

```
docker save -o framework.tar autonomus_framework:latest
scp framework.tar username@hpc-legionlogin.polito.it:/home/username/
```

Also upload the artefacts you need to analyze to the login node:

```
scp -r path/to/artefacts/folder/ \
username@hpc-legionlogin.polito.it:/home/username/artefacts/
```

Since the computing node can only communicate with the login node, you need to also install ollama on the login node and run it on the computing node. To install the binary of ollama do the following commands on the login node:

```
mkdir -p ollama/bin
cd ollama/bin
wget https://github.com/ollama/ollama/releases/latest/download/ollama-linux-amd64.tgz
tar -xvzf ollama-linux-amd64.tgz
chmod +x ollama
```

Now you can request a computing node with GPU (e.g):

```
srun --partition=gpu_a40 --gres=gpu:1 --cpus-per-task=8 \
--mem=48G --time=04:00:00 --pty bash
```

Then setup the environment variables for ollama:

```
export PATH=$HOME/ollama/bin:$PATH
export OLLAMA_MODELS=$SCRATCH/ollama/models
export OLLAMA_FLASH_ATTENTION=true
export OLLAMA_HOST=http://0.0.0.0:11434
export OLLAMA_CONTEXT_LENGTH=262144   # max context lenght of your choosing
export OLLAMA_KV_CACHE_TYPE=q8_0      # optional
export OLLAMA_MAX_LOADED_MODELS=1     # optional
```

You can start the ollama service and pull the models you want to use for the framework with the following commands:

```
ollama serve &
ollama pull <model_name>
```

Now use apptainer to build the image and run the framework:

```
apptainer build framework.sif docker-archive://framework.tar
apptainer shell --writable-tmpfs --bind \
/home/username/artefacts:/app/artefacts framework.sif
```

once inside the sandbox, you can run the framework as in the section above, for example:

```
cd /app/FrameWork/

source venv/bin/activate

export VT_API_KEY=<your_VirusTotal_API_key>

python3 createcase.py --signing-key certificates/private_key.pem \
--verify-key certificates/public_key.pem 2025 linux_local_test \\
../artefacts/linux/second_sample_b/ linux --skip_copy -y --no_hash --local
```

to modify the configuration file, you can use the "nano" command already installed in the docker image.

A.2 Programmer's Guide

This guide is intended to Programmers that wants to understand, mantain or extend the framework.

The general architecture of the project can be seen in A.3.

The diagram that explains the workflow of the framewrok can be seen in figure 4.1.

The full prompts feeded to the AI models can be seen in A.4.

A general description on how each part of the framework works, can be seen in chapter 4 of this thesis.

The following information will describe which part of the framework can be expanded and how.

A.2.1 Disk Module

The **Disk module** is responsible for the analysis of file system artifacts extracted from disk images or mounted evidence directories. It integrates tools such as `The Sleuth Kit (TSK)` and `Plaso` to extract relevant Indicators of Compromise (IoCs) from low-level file metadata and timeline data. This module can be extended (for example) by:

- Adding new parsers for additional file system formats or artifact types. The detection of the operating system and its corresponding artifact patterns is managed by the `detect_os_and_artifacts()` function:

  ```
  def detect_os_and_artifacts(fs_type: str):
      if "ext" in fs_type or "linux" in fs_type:
          return "linux", LINUX_ARTIFACT_PATHS
      elif "ntfs" in fs_type or "windows" in fs_type:
          return "windows", WINDOWS_ARTIFACT_PATHS
  ```

  New file system types (e.g., `APFS`, `exFAT`, or `ReFS`) can be supported by adding corresponding branches to this function and defining new lists of artifact paths in the configuration file (e.g., `config.py`):

  ```
  APFS_ARTIFACT_PATHS = [
      "/private/var/db/launchd.db/*",
      "/Library/LaunchAgents/*.plist",
  ]
  ```

  Once declared, these paths will automatically be processed by the `process_volume()` function using the `FileSystemSearcher` API provided by dfVFS.

  Similarly, to support new artifact categories (for example, *browser profiles* or *scheduled task caches*), the developer can simply expand the relevant artifact path list and the recursive extraction logic will handle copying and local storage transparently.

- Integrating additional timeline correlation logic. Currently, the module focuses on the extraction and static collection of persistence artifacts. To add a temporal or behavioral correlation stage (e.g., cross-referencing file modification times with system event logs), developers can hook into the end of the `process_volume()` function or extend it with an additional correlation routine.

  For example, after the call to:

  ```
  logger.info(f"Extraction for {os_type} artifacts completed in '{output_dir}'")
  ```

  a function such as `correlate_artifacts_timeline(extraction_map)` could be invoked. This function might parse file timestamps (from `file_entry.GetStatObject().mtime`) and align them with other sources (e.g., logon events, prefetch execution traces, or Plaso timelines).

  To maintain modularity, it is recommended that any new correlation logic be implemented in a separate module (e.g., `timeline_correlator.py`) and imported at runtime.

- Expanding the normalization logic to include new IoC types or STIX/OpenIOC mappings. Normalization of the extracted artifacts is currently managed within the `save_extraction_map()` function, which writes the extraction map to CSV and ensures path consistency through the `normalize_path_slashes()` helper.

  To introduce a richer normalization stage aligned with IoC standards (such as STIX 2.1 or OpenIOC), developers can extend this step by:

  - Adding a new function, e.g. `convert_extraction_map_to_stix()`, that converts each extracted file or registry entry into a standardized object.
  - Implementing a mapping layer that classifies each artifact by IoC type (e.g., `File:Path`, `Registry:Key`, `Process:Name`).
  - Optionally exporting a JSON-LD or STIX 2.1 bundle that can be consumed by other forensic or threat intelligence tools.

  Such functions can be called after `save_extraction_map()` within `extract_artifacts()`, preserving the current workflow order and maintaining compatibility with the AI analysis pipeline.

- Implementing deleted file recovery and integration with the artifact extraction pipeline.

  Another useful extension to the Disk module is the ability to recover deleted files (so-called "trash" or "unallocated" data) from disk images. This feature is crucial in forensic investigations, as deleted files can still contain traces of persistence mechanisms or indicators of compromise.

  A dedicated function, for instance `recover_deleted_files()`, can be added and called within `process_volume()` after the extraction of known artifacts:

  This function should use the dfVFS API to iterate through deleted file entries that are still physically recoverable within the file system. Recovered files can be saved in a separate folder (e.g., `recovered/`) under the current case directory. They can then be automatically included in the AI analysis stage by extending the `run_gemini_ai()` function to parse the additional recovery folder.

  To facilitate integration, the function may also log recovered file metadata (file name, size, and recovered path) to the same extraction map CSV, ensuring that they are normalized and included in the IoC generation workflow.

A.2.2 Memory Module

The **Memory module** handles the processing of volatile memory dumps and focuses on extracting IoCs related to process activity, network connections, and in-memory artifacts. It uses tools such as `Volatility3` for data extraction and provides structured outputs that can be automatically parsed by the AI analysis component. Future developers can extend this module (for example) by:

- Implementing new Volatility plugins or analysis scripts. The plugins executed on each memory dump are defined in the configuration file (`config.py`) under the dictionary `PLUGINS_BY_OS`. For example:

  ```
  PLUGINS_BY_OS = {
      "windows": ["windows.pslist", "windows.registry.hivelist", "windows.malfind"],
      "linux": ["linux.pslist", "linux.netstat", "linux.lsof"]
  }
  ```

  To add new plugins or custom analysis scripts, developers simply append the plugin names to the relevant OS list in `config.py`. The task-building functions (`build_tasks`, `build_tasks_auto`, `build_tasks_from_folder`) automatically detect these new entries and schedule their execution. Output for each plugin is saved in a case-specific folder, with the plugin name used as the output filename.

- Adding correlation functions between memory artifacts and disk/network IoCs. To integrate memory analysis with disk or network IoCs, developers can implement correlation functions that combine the outputs of Volatility plugins with extraction maps from the Disk and Network modules.

  A suggested approach:

  - After plugin execution (`execute_tasks_in_parallel()`), parse the text output files to extract relevant indicators such as processes, network connections, or registry keys.
  - Cross-reference these indicators with disk artifact maps (`extraction_map.csv`) or network logs.
  - Store correlated results in a standardized format for AI analysis or report generation.

  Such correlation logic can be implemented in a separate module, e.g., `memory_correlator.py`, and called after memory plugin execution.

- Comparing plugin output with a clean baseline.

  An effective approach could be to to compare the outputs of Volatility plugins against a baseline memory dump obtained from a known clean (non-infected) system.

  The workflow can be as follows:

  - Maintain a set of baseline memory dumps for each supported OS, ideally corresponding to standard system configurations.
  - After running plugins on the target memory dump, parse the outputs (e.g., processes, DLLs, network connections).
  - Compare each parsed indicator with the corresponding plugin output from the baseline machine.
  - Highlight differences that may indicate malicious activity, such as unexpected processes, injected code, anomalous loaded modules, or unusual network connections.
  - Store the anomalous findings in a structured format for AI-assisted analysis or reporting.

  Implementing this approach requires a dedicated comparison module (e.g., `baseline_comparator.py`), which can be called after the parallel execution of all Volatility plugins. This method allows the framework to detect deviations from normal behavior rather than relying solely on known IoCs, providing a more flexible detection mechanism for previously unseen malware.

A.2.3 Network Module

The **Network module** manages the collection and analysis of network traffic data. It supports parsing of PCAP files and integration with external sources such as `ANY.RUN` or sandbox reports to extract and correlate network-based IoCs. Future developers can extend this module (for example) by:

- Adding new parsers for alternative report formats or sandbox APIs. Currently, the module supports PCAP files and VirusTotal API queries. Developers can extend it by implementing new parsers for alternative network sources or sandbox outputs:
  - Create a new parser class or function that extracts IoCs from the desired file format (e.g., Zeek logs, Cuckoo sandbox JSON reports).
  - Ensure that extracted IoCs are returned as a standardized set of string representations for compatibility with the rest of the pipeline.
  - Integrate the new parser into the main workflow, similar to `extract_iocs_from_pcap()`, so that the subsequent VirusTotal or correlation functions can process them seamlessly.
- Integrating automated enrichment with external threat intelligence feeds. The module demonstrates integration with VirusTotal via `query_virustotal()`. To add new sources:
  - Add API credentials and base URLs to `config.py`.
  - Implement a query function that accepts an IOC and returns a structured dictionary containing at least: type, value, and a classification or score.
  - Include rate-limiting handling and error checking similar to the VirusTotal function.
  - Modify `process_virustotal_results()` to iterate over all configured threat intelligence sources and merge results.

  The enriched IoCs are saved both as a CSV summary and, if malicious or suspicious, as a full JSON report for forensic inspection.
- Expanding the IoC correlation model with temporal or behavioral patterns. Beyond individual IoC reputation, the module can correlate network artifacts with temporal or behavioral patterns to identify suspicious activity:
  - **Temporal correlation:** Compare the timestamps of captured network events or IoCs against known attack timelines or the system's disk/memory artifacts.
  - **Behavioral correlation:** Identify patterns such as repeated connections to uncommon IPs, abnormal port scanning, or malware C2 communication sequences.
  - Implement dedicated functions (e.g., `correlate_temporal_patterns()`) that analyze IoC frequency, sequence, and context across multiple cases or PCAP files.
  - Store correlated events in structured tables to feed AI models for automated threat assessment or to highlight anomalies in forensic reports.

A.2.4 AI Analysis and Report Generation

The AI-based components of the framework perform the automatic interpretation of extracted IoCs and the generation of forensic reports. Developers can extend this part by modifying or adding new AI models, updating the prompt templates (see Appendix A.4), or connecting the system to other data sources for cross-domain analysis.

Also, expanding the report generation to not just one but multiple report each one tailored to a forensic team specific role could be useful. To generate reports tailored for specific roles (e.g., malware analyst, incident responder, network forensic specialist):

1. Define different AI prompt sets for each role in `config.py`, emphasizing the information relevant to that specialist.
2. Duplicate the `generate_summary()` call for each role, passing the role-specific instructions.
3. Assemble separate Markdown reports or append role-specific sections within a single document.
4. Include a unified Table of Contents and summary per report to maintain readability.

This approach ensures that each forensic team member receives a report focused on their operational needs while maintaining a coherent overall narrative.

A.2.5 Extensibility Guidelines

To maintain modularity and readability, each module follows a clear interface and uses a shared configuration file for adjustable parameters. All new code should respect the existing structure and logging conventions to ensure compatibility with the report generation pipeline. Developers are encouraged to:

- Keep logic separated from configuration.
- Maintain a consistent logging level and output format.
- Follow the naming conventions and modular design outlined in Appendix A.3.

A.3 Full Project Structure

```
.
+-- Framework/
|   +-- config.py                 # Global configuration (lists, variables, etc.)
|   +-- createcase.py             # Main orchestrator: creates case folder, copies
|                                  artifacts, hashes, verifies, starts processing
|   +-- process_acquisitions.py   # Called by createcase.py, runs analysis scripts
|                                  for disk/memory/network, then report generation
|   +-- verify_signatures.py      # Called by process_acquisitions.py, verifies
|                                  manifest file against signatures
|   +-- patch_tshark.py           # Fixes tshark incompatibility for Python < 3.7
|   +-- requirements.txt          # Python dependencies
|   |
|   +-- certificates/             # Keys used to sign/verify manifests
|   |   +-- private_key.pem
|   |   +-- public_key.pem
|   |
|   +-- scripts/
|       +-- final_report_generation.py   # Generates final report with AI verdict
|       |
|       +-- disk/
|       |   +-- scan_disk.py            # Scans disk images, extracts persistence
|       |                                  artifacts, AI analysis with Gemini
|       |   |
|       |   +-- AI_agent/
|       |       +-- disk_agent.py       # LangGraph AI agent for IOC detection,
|       |                                  registry hive extraction
|       |       |
```

```
|       |       +-- actions/
|       |           +-- cat_file.py      # AI action: view file contents (cat)
|       |           +-- folder_scan.py   # AI action: list folder contents (ls/dir)
|       |
|       +-- memory/
|       |   +-- ask_ai.py               # AI report generation from memory artifacts
|       |   +-- scan_memory.py          # Volatility plugins on memory dump
|       |
|       +-- network/
|           +-- scan_network.py         # Extract IPs/domains from PCAP, query
|                                      # VirusTotal threat intelligence
|
+-- windows/
|   +-- casefolder/                     # Example autonomous analysis on Windows
|       +-- 00_meta/
|       |   +-- case_info.txt           # Metadata about the current case
|       |
|       +-- 01_acquisitions/
|       |   +-- hdd_image_flat.vmdk     # Disk image
|       |   +-- memorydump.raw          # RAM dump (or memorydump.lime)
|       |   +-- network_traffic.pcap    # Network capture
|       |
|       +-- 02_raw_checksums/
|       |   +-- manifest.txt            # Manifest file
|       |   +-- manifest.txt.sig        # Signature of the manifest
|       |
|       +-- 03_processing/
|       |   +-- disk/
|       |   |   +-- extraction_map.csv  # Map: extracted path <=> case folder path
|       |   |   +-- 00_agent_reports/   # AI agent outputs (.md, .json)
|       |   |   +-- [extracted files & folders...]
|       |   |
|       |   +-- memory/
|       |   |   +-- 00_agent_reports/   # AI reports on plugin outputs (.md)
|       |   |   +-- clean_examples/     # Reference outputs from clean system
|       |   |   +-- [volatility plugin outputs...]
|       |   |
|       |   +-- network/
|       |       +-- vt_ioc_results.csv  # Extracted IOCs + VirusTotal results
|       |       +-- [full reports of malicious IPs/domains in .json]
|       |
|       +-- 05_reports/
|           +-- forensic_report.md      # Final AI-generated forensic report
|
+-- linux/
|   +-- casefolder/                     # Example autonomous analysis on Linux
|
+-- README.md                           # Project documentation
```

A.4 Forensic Agents Prompts

This appendix reports the prompt templates used for guiding the forensic Large Language Model (LLM) agents. They are grouped by functionality: disk analysis, memory analysis, and final reporting.

A.4.1 Disk Agent Prompts

Summarization Prompt. This prompt reduces the context length of the main forensic agent's history. It enforces a strict output format to ensure compatibility with the main agent.

Listing A.1. Disk agent summarization prompt

```
DISK_AGENT_SUMMARIZE_PROMPT = """You are a subagent that assist a main forensic llm
    agent. you task is to reduce the context lenght of its history. The main agent
    accepts only this format, do not add reasoning to your response:
LLM: -
Observation: -
Final Answer: -
Each on a newline.
Here is the history you have to manage:""" #+ "\n".join(history)
```

File Analysis Prompt. This prompt instructs the agent to analyze a single file for malicious behavior. The output must follow a strict JSON format including suspiciousness, reasoning, and potential Indicators of Compromise (IoCs).

Listing A.2. Disk agent file analysis prompt

```
DISK_AGENT_ANALYZE_FILE_PROMPT = """You are a digital forensics expert. Analyze the
    following file for potential malicious behavior, suspicious indicators, or
    anomalies.

Instructions:
1. Examine the content carefully.
2. Decide if the file is suspicious or benign.
3. Identify any indicators of compromise (IoCs) you can find.
4. Return your response in EXACTLY this JSON format:
5. Look out for the dates, today is {current_date}

{{
  "file_name": "{file_to_examine}",
  "suspicious": true/false,
  "reasoning": "<brief explanation of why it is suspicious or benign>",
  "indicators": ["<list of any IoCs or suspicious patterns found>"]
}}

File content:
{file_content}
"""
```

Folder Analysis Prompt. This prompt enables recursive analysis of a folder structure from a disk image. It guides the agent in identifying anomalous files or directories and optionally triggers further scripted analysis.

Listing A.3. Disk agent folder analysis prompt

```
DISK_AGENT_ANALYZE_FOLDER_PROMPT = """You are a forensic agent that can use a Python
    script to analyze a folder.

Your task:
   - The input is the output of an ls (Linux/macOS) or dir (Windows) command, if
       linux use "/" if Windows use "\\"".
   - Analyze this input to identify any anomalous files or directories, mind the
       dates: today is {current_date} (the LAST MODIFIED field of the input has date
       format <year>-<month>-<day>).
   - For files, if any seem anomalous, return their names as a dictionary (with name
       + reason) in the final answer.
      scan all files that may be household of malware persistance.
      If you are unsure whether a file is anomalous or not, you may use the
          following action to inspect its contents:
      {SCRIPT2_PATH} {HDD_IMAGE_PATH} "{{STARTING_FOLDER}}\\\\<file>"
   - For directories, if any seem anomalous, respond with:
     Action: script_name.py <directory_name>
     (This will trigger a deeper analysis on the selected directory.)
   - GO AT LEAST 1 LEVEL DEEP FOR ALL FOUND FOLDERS IN THE INITIAL INPUT
```

```
WARNING: Ignore files usually classified as folder or viceversa, they are not
    anomalies but maybe corrupted.

Use the following response formats:
   - LLM: <your analysis> - to provide your reasoning or analysis
   - Observation: <your observation> - to log any findings or observations
   Then follow with:
   - Final Answer: <your final answer> - to provide the result or conclusion, include
       the full paths
   - Action: {SCRIPT_PATH} {HDD_IMAGE_PATH} "{{STARTING_FOLDER}}\\\\<directory_name>"
       - if further action is needed, Note: only one action at a time, remember to
       add the hdd image path, DONT WRAP IT IN text or anything.Current
       history:contextUser input:user_inputPlease provide your analysis and next
       steps."""
```

A.4.2 Memory Agent Prompts

Chunk Memory Prompt. Used when processing large Volatility3 outputs, this prompt filters and retains only anomalous or relevant rows from memory analysis.

Listing A.4. Forensic memory chunk filtering prompt

```
FORENSIC_CHUNK_MEMORY_PROMPT = """You are processing a long input of a Volatility3
    plugin output.
Your job is to cut the output on only the rows that you find anomalous/relevant to a
    forensic case.
Context of previous chunks:
context
RESPOND STRICTLY AND ONLY WITH THIS OUTPUT FORMAT:
'''
* Final Output (Only Relevant/Anomalous Rows):
OFFSET (V) PID TID PPID COMM UID GID EUID EGID CREATION TIME File output
'''

input:
'''
{chunk_text}
'''
"""
```

Memory Analysis. This prompt analyzes memory dumps. It highlights anomalies, suspicious processes, and potential malicious behavior while avoiding false positives due to forensic workflows.

Listing A.5. Forensic memory analysis prompt

```
FORENSIC_MEMORY_PROMPT = """You are a digital forensic assistant. Your task is to
    generate a forensic report section in Markdown format.

Context

You are provided with the outputs of a Volatility plugin executed on the machine
    under investigation:
{target_content}

Task

1. Examine the plugin output carefully for anomalies, suspicious entries, or
    deviations from expected system behavior.
2. Interpret forensic significance, explaining why findings may indicate malicious
    activity or abnormal system behavior.
3. Maintain professional forensic reporting style: concise, factual, and clear. Avoid
    speculation unless grounded in evidence.
```

```
4. Ignore and do not flag as anomalous any actions consistent with forensic/analyst
    workflows.
   Examples:
   - Extracting password-protected ZIPs with password "infected" (common for sandboxed
       malware samples).
   - Renaming or saving extracted malware as files like "malware".
   - Loading kernel modules with insmod or modprobe when attributable to forensic
       memory acquisition tools (e.g., LiME).
   - Using sudo/su to run those tools.
5. Handle plugin malfunctions carefully: do not interpret them as malicious
    indicators.
   Examples:
   - Extremely large or non-standard values for PID, PPID, UID, GID, EUID, or EGID.
   - Corrupted or garbled process names.
   - Impossible timestamps (e.g., year 1930 or similar).
   These must be treated as artifacts of plugin parsing, not as rootkit or malware
       evidence.
6. Ignore any processes or artifacts that are clearly part of forensic or analyst
    workflows.
   Examples:
   - Memory acquisition and analysis tools (e.g., LiME, Volatility, Rekall).
   - Network sniffers used for legitimate analysis (e.g., Wireshark) during
       investigation.
   - Common analyst scripts or debugging tools (e.g., Python scripts, WinDbg).
   These should not be flagged as suspicious, anomalous, or malicious.

Output format (Markdown)

- Begin with a section title (e.g., ## Memory Analysis - [PluginName])
- Provide a short overview of what the plugin analyzes.
- Provide a narrative analysis describing the findings.
- Conclude with a forensic interpretation (possible implications, next steps).

Warning: insmod is not an IoCs because Lime was use to dump the memory of the machine
    by a forense agent.
Respond with just the section, no intro.
"""
```

A.4.3 Final Report Prompts

Main Forensic Instruction. This prompt aggregates findings across disk, memory, and network analysis. It emphasizes conservative evidence-based reporting and provides a structured forensic conclusion.

Listing A.6. Main forensic reporting instruction

```
MAIN_FORENSIC_INSTRUCTION = """You are analyzing forensic evidence.
Input could be given in multiple JSON chunks if too long.
Summarize the key findings across disk, memory, and network.
Rules:
1. Ignore and do not flag as anomalous any actions consistent with forensic/analyst
    workflows.
   Examples:
   * Extracting password-protected ZIPs with password "infected" (common for
       sandboxed malware samples).
   * Renaming or saving extracted malware as files like "virus.elf".
   * Loading kernel modules with insmod or modprobe when attributable to forensic
       memory acquisition tools (e.g., Dumpit, LiME).
   * Using sudo/su to run those tools.

2. Plugins could not work correctly. Do not use those evidences for a verdict.
   Examples:
```

```
    * 99% of the times, extremely large, non-standard values for UID, GID, EUID, and
      EGID in process dumps, does not indicate memory structure manipulation, but
      the plugin not working correctly.
    * Corrupted in memory process names.
3. Only highlight artifacts that cannot be explained as part of forensic workflows.
    Examples:
    * Unexpected binaries with obfuscation or irregular permissions.
    * Network communications with malicious IPs/domains not tied to forensic activity.
    * Registry or configuration artifacts that match known malware patterns and cannot
      be explained by legitimate activity.
4. ALWAYS consider the possibility of false positives, the reports of disk and memory
    are NOT perfect and 100% reliable. Be always open to suggest the forense agent to
    double-check things instead of declaring the machine compromised.
5. Findings must be evidence-based, and verdicts must be conservative:
     _A single unusual artifact (e.g., oddly named registry key or unknown file) is_
         *not enough** to declare suspicious without corroboration.
     _Corroborating evidence (e.g., matching malicious network traffic, known malware
         hash, or suspicious memory behavior) is required for a_ *Suspicious**
         verdict.
     _If findings are weak or inconclusive, state_ *Not Compromised** or
         **Inconclusive** depending on context.
Format your response in the following sections:
* **Suspicious/malicious activity (report)**
At the end, provide a concise forensic conclusion in one of the following formats:
**Final Verdict: Suspicious**
**Final Verdict: Not Compromised**
**Final Verdict: Inconclusive**
Include a brief reasoning and suggest next steps for the forensic team.
"""
```

Chunk Processing Template. This template ensures long inputs are processed in chunks while preserving forensic consistency. Each chunk is summarized in strict JSON format.

Listing A.7. Chunk processing template

```
CHUNK_PROMPT_TEMPLATE = """You are processing a long input in multiple chunks.
This is chunk {chunk_number}/{total_chunks}.

{context}
Chunk content:
\"\"\"
{chunk_text}
\"\"\"

- Summarize the key findings in this chunk.

When evaluating artifacts, ignore and do not flag as anomalous any actions consistent
with forensic acquisition or analyst workflows. Examples include:
  - Use of memory acquisition tools (e.g., DumpIt, LiME, WinPMEM, AVML).
  - Execution of insmod/modprobe when attributable to LiME or other acquisition tools.
  - Extraction of password-protected malware samples (e.g., ZIPs with password
      "infected")
    and saving/renaming them (e.g., virus.elf) as part of controlled analysis.
  - Privilege escalation via sudo/su when clearly used to run acquisition or analysis
      tools.
  - Use of analyst scripts for unpacking or staging samples.

Highlight only suspicious or malicious activity that cannot be explained as part of
forensic or analyst workflows.
- DO NOT explain or provide methods.
- You MUST return the output in EXACTLY this JSON format:
{{
  "chunk_number": {chunk_number},
```

```
  "summary": "<readable extracted content from this chunk>"
}}
- Do NOT change the keys.
- Maintain this format consistently for all chunks.
- Return ONLY the JSON object, nothing else."""
"""
```

Final Prompt Template. This template merges multiple chunk summaries into a unified forensic analysis, following the main instruction set.

Listing A.8. Final prompt template

```
FINAL_PROMPT_TEMPLATE = """You have received {number_of_summaries} chunk summaries:

{summaries}

Main Task:
{main_instruction}
"""
```

A.4.4 Disk Agent for Local LLM Prompts

Chunk Disk Prompt Template. This prompt processes lengthy `ls` or `dir` outputs, retaining anomalous rows and full folder structures for forensic inspection.

Listing A.9. Chunk disk prompt template

```
CHUNK_DISK_PROMPT_TEMPLATE = """You are processing a long input of a ls output of a
    program.
Your job is to cut the output on only the rows that you find anomalous/relevant to a
    forensic case.
Keep all the folders in your output.
Context of previous chunks:
'''
{context}
'''
RESPOND STRICTLY AND ONLY WITH THIS OUTPUT FORMAT:
'''
* Final Output (Only Relevant/Anomalous Rows + All Folders):
PERMISSIONS UID GID SIZE LAST_MODIFIED NAME
'''
input:
'''
{chunk_text}
'''
"""
```

Local Disk Analysis Prompt. This prompt guides the forensic agent in analyzing a folder listing from a mounted disk image. It specifies anomaly criteria, recursive inspection, and controlled triggering of deeper analysis.

Listing A.10. Disk prompt for local analysis

```
DISK_PROMPT_LLM_LOCAL = """Current history:
{context}
User input:
{user_input}
---

You are a forensic agent that can use a Python script to analyze a folder.
```

```
Your task:
    - The input is the output of an ls (Linux/macOS) or dir (Windows) command, if
        linux use "/" if Windows use "\\"".
    - Analyze this input to identify any anomalous files or directories, mind the
        dates, today is {current_date}.
    - For files, if any seem anomalous, return their names as a dictionary (with name
        + reason) in the final answer.
       scan all files that may be household of malware persistance.
       If you are unsure whether a file is anomalous or not, you may use the
           following action to inspect its contents:
       {SCRIPT2_PATH} {HDD_IMAGE_PATH} "{{STARTING_FOLDER}}\\\\<file>"
    - For directories, if any seem anomalous, respond with:
      Action: script_name.py <directory_name>
      (This will trigger a deeper analysis on the selected directory.)
    - GO AT LEAST 1 LEVEL DEEP FOR ALL FOUND FOLDERS IN THE INITIAL INPUT
WARNING: Ignore files usually classified as folder or viceversa, they are not
    anomalies but maybe corrupted.

Use the following response formats:
    - LLM: <your analysis> - to provide your reasoning or analysis
    - Observation: <your observation> - to log any findings or observations
    Then follow with:
    - Final Answer: <your final answer> - to provide the result or conclusion, include
        the full paths.
    - Action: {SCRIPT_PATH} {HDD_IMAGE_PATH} "{{STARTING_FOLDER}}\\\\<directory_name>"
        - if further action is needed, Note: only one action at a time, remember to
        add the hdd image path, DONT WRAP IT IN text or anything.Please provide your
        analysis and next steps."""
```

Bibliography

[1] National Institute of Standards and Technology, "Nist computer security resource center glossary: Digital forensics", https://csrc.nist.gov/glossary/term/digital_forensics, 2025, [Online; accessed 3-September-2025]

[2] J. Caballero, G. Gomez, S. Matic, G. Sánchez, S. Sebastián, and A. Villacañas, "The rise of goodfatr: A novel accuracy comparison methodology for indicator extraction tools", Future Generation Computer Systems, vol. 144, 2023, pp. 74–89

[3] Splunk Inc., "Ioc: Indicators of compromise", 2025, Accessed 2025-08-14

[4] A. Villalón-Huerta, I. Ripoll-Ripoll, and H. Marco-Gisbert, "Key requirements for the detection and sharing of behavioral indicators of compromise", Electronics, vol. 11, no. 3, 2022, p. 416

[5] Picus Security, "Mitre att&ck framework: Beginner's guide", 2025, Accessed 2025-08-14

[6] Terrabyte Group, "The pyramid of pain in cybersecurity: A strategic approach to ...", 2025, Blog post

[7] OASIS Open, "Stix™ 2.x documentation", 2025, Accessed 2025-08-14

[8] WatchGuard Technologies, "Indicators of compromise overview", 2025, Accessed 2025-08-14

[9] Wikipedia contributors, "Indicator of compromise", 2025, Accessed 2025-08-14

[10] MITRE Corporation, "Mitre att&ck®", 2025, Accessed 2025-08-14

[11] National Institute of Standards and Technology (NIST), "Guide to integrating forensic techniques into incident response", special publication 800-86, NIST, 2006. Accessed 2025-08-14

[12] Trusted Institute, "File system forensics", 2025, Accessed 2025-08-14

[13] E. Casey, "Digital evidence and computer crime", Academic Press, 4 ed., 2019

[14] D. Quick and R. Choo, "Forensic collection and examination of digital evidence", Springer, 2020

[15] W. Zhong and J. Zhang, "Forensic analysis of ext4 journaling and delayed allocation", DFRWS USA, 2020

[16] D. Garcia and M. Alvarez, "Robust timeline reconstruction from file system journals", Digital Investigation, vol. 36, 2021, p. 301112

[17] Forensic Science Anthology, "Memory forensics: The analysis of volatile system memory", 2025, Accessed 2025-08-14

[18] M. H. Ligh, A. Case, J. Levy, and A. Walters, "The art of memory forensics, updated techniques", Wiley, 2020

[19] J. Levy and A. Case, "Detecting process hollowing and code injection in modern windows", Digital Investigation, vol. 40, 2022, p. 301401

[20] V. Foundation, "Volatility 3 framework documentation", https://www.volatilityfoundation.org/volatility3/, 2024

[21] Volatility Foundation, "Volatility framework for memory analysis", 2025, Accessed 2025-08-14

[22] Kentik Inc., "Network forensics", 2025, Accessed 2025-08-14

[23] C. Sanders, "Practical network defense and monitoring", Rural Technology Fund, 2020

[24] R. Sommer and M. Vallentin, "Modern network telemetry for threat hunting", ACM Queue, vol. 20, no. 2, 2022

[25] Z. Project, "Zeek documentation and project overview", https://www.zeek.org/, 2024

[26] W. Foundation, "Wireshark user's guide, version 4.7.0", https://www.wireshark.org/docs/wsug_html_chunked/, 2025

[27] VirusTotal, "Virustotal threat intelligence api documentation", 2024, API v3 reference and usage

[28] Forensics Colleges, "Automation in digital forensics", 2025, Accessed 2025-08-14

[29] T. D. Le, D. Dinh, P. Nguyen T. H., A. Muthanna, and A. Abd El-Latif, "Exploring common malware persistence techniques on windows operating systems (os) for enhanced cybersecurity management", pp. 107–149. 11 2023

[30] Hadess, "The art of linux persistence", 2023

[31] Freedesktop.org, "systemd manual pages", 2025. Accessed online: https://www.freedesktop.org/software/systemd/man/

[32] Freedesktop.org, "systemctl manual pages", 2025. Accessed online: https://www.freedesktop.org/software/systemd/man/systemctl.html

[33] C. Sanders and J. Smith, "Linux persistence mechanisms and forensic analysis", Digital Forensics Magazine, vol. 18, no. 3, 2020, pp. 45–56. Focus on systemd, cron, and user-level persistence

[34] Linux Foundation, "Filesystem hierarchy standard", 2015

[35] ANY.RUN, "Any.run malware analysis report: Worm:win32/ludbaruma", 2025, Available at: https://any.run/report/9ed5e265c25e1e592bff48865e83b0980f578f29622cc1c0268a323dfac15639/96a054fc-9100-483e-89e7-66f2414cd108, Accessed: 2025-09-19

[36] ANY.RUN, "Any.run malware analysis report: conhoz loader", 2025, Available at: https://any.run/report/0512c44f7b16ba8a25913f5e698b79203cf38bdc5c766e23b9fef0d3d1cd8833/e668e173-cef4-4c52-ba7d-f0a557dcfef9, Accessed: 2025-09-19

[37] ANY.RUN, "Any.run malware analysis report: Prometei", 2025, Available at: https://any.run/report/d5baf5e8574d7a9eff594b5c5805eef22df2de646a6fddc5ebd7cba2446ff4a3/9d71b311-2d7b-4aff-b544-bed896fc8193, Accessed: 2025-09-19

[38] ANY.RUN, "Any.run malware analysis report: gh0st rat", 2025, Available at: https://any.run/report/b560f76f7603e3ec88a874085f15499ec043917d93e306b3b0fb7a913b54f287/552f2c6c-d8e8-4655-8a34-5c136938bd9f, Accessed: 2025-09-19

[39] Abyss-W4tcher, "Volatility 3 symbols repository", https://github.com/Abyss-W4tcher/volatility3-symbols, 2024, Accessed: 2025-08-31

[40] Hack The Box, "How to create linux symbol tables for volatility", https://www.hackthebox.com/blog/how-to-create-linux-symbol-tables-volatility, 2023, Accessed: 2025-08-31

[41] "Hpc@polito", Technical infrastructure / computing facility, 2025, Computational resources provided by HPC@POLITO (www.hpc.polito.it)