



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



Vilnius
University

Nicolò Pellegrinelli

Telegram-Based Commodity Infostealers: Operator-Side IoC Extraction and Victim-Side Delivery-Vector Taxonomy

Master Dissertation

Master's on International Cybersecurity and Cyberintelligence

Work developed under the guidance of:

Alberto Casares Andrés

Gabriel Maciá Fernández

June 2026

COPYRIGHT AND TERMS OF USE OF THE WORK BY THIRD PARTIES

This is an academic work that third parties may use, provided that internationally accepted rules and best practices regarding copyright and related rights are respected.

Therefore, this work may be used under the terms of the license indicated below.

If the user needs permission to use the work under conditions not covered by the indicated license, they should contact the author through the RepositoriUM of the University of Minho.

License granted to users of this work.



This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>

Braga, 30th June 2026

A handwritten signature in black ink, which appears to read 'Nuno Pereira', is written above a horizontal line.

(Student Name)

Abstract

Commodity infostealers increasingly shift from dedicated Command-and-Control panels to the Telegram Bot API, embedding the operator's Bot Token and destination Chat ID directly within the distributed binary. These identifiers bind a sample to both the operator behind a campaign and the victims it infects, yet the operator-facing binary and the victim-facing log have been studied only in isolation so far. This thesis presents an empirical measurement of the Telegram-based commodity infostealer ecosystem from both sides at once. On the operator side, a reproducible host-only pipeline detonates a corpus drawn from a public malware repository and recovers Telegram Indicators of Compromise (IoC) through combined static and dynamic analysis, supported by a synthetic responder that defeats the connectivity and geolocation checks samples perform before exfiltrating. Telegram indicators are recovered from 173 samples, reducing to 110 distinct bot tokens, of which 52.7 % remain reachable months after collection. Static and dynamic recovery are found to be interdependent rather than redundant. On the victim side, a second pipeline reconstructs infection vectors for each victim by analyzing stealer log browsing histories. It then groups these vectors into a taxonomy of delivery methods using host-transition graphs and community detection, with most linked to cracked software and firmware lures. The two modalities represent separate segments of the same ecosystem: the families that reveal operator identifiers via Telegram exfiltration are distinct from those that dominate the victim logs. This inherent disjointness defines the boundary that any unified operator-to-victim view must bridge.

Index

1	Introduction	1
2	Background and Related Work	3
2.1	The Commodity Infostealer Ecosystem and Telegram-Based C2	3
2.2	Underground Markets for Stealer Logs and Stolen Credentials	4
2.3	Stealer Log Analysis and Victim Telemetry	5
2.4	Web-Based Malware Delivery and Victim-Path Reconstruction	6
2.5	Telegram as Cybercrime Infrastructure	6
2.6	Empirical Malware Measurement Methodology	7
2.7	IoC Extraction, Configuration Extraction and Normalization	8
2.8	Malware Analysis and Anti-Analysis Techniques	9
2.9	Gap Analysis	10
3	Research Design and Methodology	12
3.1	Analysis Environments	12
3.1.1	Ethical Isolation and the Evasion Trade-off	12
3.1.2	Synthetic Response Strategy	12
3.2	Corpus Selection and Construction	13
3.2.1	Source Rationale	13
3.2.2	Family Selection and Query Design	14
3.2.3	Scope Definition and File-Type Filtering	17
3.2.4	Sample Admission	18
3.3	Sandbox Orchestration	19
3.3.1	Role and Scope	19
3.3.2	Archive Containers and Payload Identity	19
3.3.3	Two-Pass Detonation	20
3.3.4	Viable Passes and Evidence Preservation	20
3.3.5	Layered Analysis Stages	20
3.3.6	Resumability and Fault Isolation	21
3.3.7	Containment and Hygiene	21
3.4	IoC Extraction Methodology	21
3.4.1	Extraction Target and Unit	22
3.4.2	Evidence Sources and Their Roles	23
3.4.3	Candidate Extraction and Normalization	25

3.4.4	Confidence Assignment	25
3.4.5	The Diagnostic Ladder: Making Absence Measurable	26
3.4.6	Measurement Integrity Safeguards	26
3.5	Stealer Log Ingestion Methodology	27
3.5.1	Role, Data Source, and Ethical Posture	27
3.5.2	Ingestion Design	28
3.5.3	Format Detection and Common-Schema Extraction	28
3.5.4	Install-Filename Normalization and Ranking	29
3.5.5	Stealer-Family Identification	29
3.5.6	History-Host Aggregation	30
3.5.7	Host-Path Trajectories	30
3.6	Delivery-Vector and Point-of-Infection Methodology	31
3.6.1	From Trajectories to Infection Vectors	32
3.6.2	Host-Transition Graph	32
3.6.3	Ecosystem Grouping via Leiden Community Detection	32
3.6.4	Shared Delivery Infrastructure	33
3.6.5	Sequential Infection-Vector Patterns	33
3.6.6	Family-to-Vector Attribution	34
3.6.7	Operator-Brand Correlation	34
3.6.8	The Last-Non-Sink Hop: Direct Delivery-Vector Recovery	34
3.6.9	Validating Communities as Delivery Vectors	35
3.6.10	Structural Bias and Coverage Limits	35
4	System Design and Implementation	37
4.1	Manual Analysis Lab	37
4.1.1	Victim and Observer	37
4.1.2	Network Simulation and Traffic Capture	37
4.1.3	Malware Analysis Hygiene	37
4.2	CAPE Sandbox Environment	38
4.2.1	Network Routing and Isolation	38
4.2.2	Synthetic Responders	38
4.3	Sample Corpus Construction	39
4.3.1	Architecture and Execution Model	39
4.3.2	Data Model and State Tracking	39
4.3.3	Query Orchestration and MalwareBazaar Client	40

4.3.4	File-Type Filtering	40
4.3.5	Download Module	41
4.3.6	Admission and Sanity Checking	41
4.3.7	Reporting and Reproducibility	42
4.4	CAPE Orchestrator Module	42
4.4.1	Modular Architecture and Data Flow	42
4.4.2	Configuration Model	43
4.4.3	State Model and Persistence	43
4.4.4	Sandbox Client	44
4.4.5	Job Execution and Main Loop	44
4.4.6	Static and Payload Analysis Stages	44
4.4.7	Command-Line Interface	45
4.5	IoC Extraction Module	45
4.5.1	Module Architecture and Data Flow	45
4.5.2	Token & Chat-ID Pattern Matching	46
4.5.3	Per-Source Extractors	48
4.5.4	Per-Sample Orchestration	50
4.5.5	Reconciliation and Classification	50
4.5.6	Persistence	51
4.5.7	Command-Line Interface	51
4.5.8	Database Schema	51
4.6	Stealer Log Ingestion Pipeline	53
4.6.1	Package Architecture and Data Flow	53
4.6.2	Log Iteration, Format Detection and Per-Format Parsing	53
4.6.3	Single-Pass Accumulation	54
4.6.4	History Collection and URL Reduction	55
4.6.5	Common-Token Filters	55
4.6.6	YARA Family Identification	55
4.7	Delivery-Vector Analysis Implementation	55
4.7.1	Community Detection and the Two Graph Views	56
4.7.2	Pattern Mining and Family Attribution	57
4.7.3	Delivery-Vector Experiment Scripts	57
4.7.4	Command-Line Interface	58
4.7.5	Stealer-Log Database	58

5	Results and Analysis	59
5.1	Corpus and Detonation Coverage	59
5.2	Telegram IoC Extraction	61
5.2.1	Extraction Outcomes Along the Diagnostic Ladder	61
5.2.2	Extraction Yield by Family	62
5.2.3	Indicator Yield and Confidence	62
5.2.4	Static Versus Dynamic Distribution	64
5.2.5	Pass Contribution and Corroboration	65
5.2.6	Distinct Command-and-Control Infrastructure	66
5.2.7	Operator Clusters from Shared Infrastructure	66
5.2.8	Threats to Validity	69
5.2.9	Live-Infrastructure Validation	70
5.3	Stealer-Log Results	71
5.3.1	Ingestion Coverage and Denominators	72
5.3.2	Install-Filename Ranking	72
5.3.3	Stealer-Family Distribution	73
5.4	Delivery-Vector and Point-of-Infection Results	74
5.4.1	The Transition Graph and the Component Collapse	74
5.4.2	Ecosystem Communities and the Resolution Sweep	75
5.4.3	Shared Delivery Infrastructure	76
5.4.4	Sequential Infection-Vector Patterns	77
5.4.5	Operator-Brand Correlation	78
5.4.6	Direct Lure Recovery: the Last-Non-Sink Hop	79
5.4.7	Method Convergence	80
5.4.8	Structural Bias and Threats to Validity	80
6	Conclusions and Future Work	83
6.1	Summary of Contributions	83
6.2	Linking the Two Modalities	84
6.3	Limitations	84
6.4	Future Works	84
	Bibliography	85
	Appendix I - YARA Rules	93
I.1	Telegram Indicator Taxonomy Rules	93

I.2	Stealer-Family Identification Additions	96
-----	---	----

Figures Index

Distribution of the 3,035 analyzed samples across the diagnostic ladder.	61
Recovered endpoints by confidence tier.	64
Operator clusters as connected components. Circles are samples, distinct by family label; squares are bot tokens; diamonds are destination chats, abbreviated to their first four digits. The full SHA-256 of every clustered sample is listed in Table 5.14.	68
Live versus revoked (or malformed) bot tokens per malware family.	71
Share of each family's YARA-labeled logs that yielded a browsing trajectory.	74
The same graph under two groupings, by descending unit size (log-log).	75
Operator-brand composition of each curated community.	79
Cumulative share of victim last-hops covered by the top last-non-sink lure domains.	80

Tables Index

Malware family acquisition allowlist.	14
File Type Admission Categories.	17
File Type Exclusion Categories.	18
Corpus construction pipeline steps.	39
File type rejection codes.	41
Submodules of the orchestrator and their responsibilities.	43
Modules of the extractor and their responsibilities.	45
Correspondence between the <code>FailureLabel</code> enumeration in <code>models.py</code> and the rungs of the diagnostic ladder defined in Section 3.4.5, in descending order of Telegram-evidence strength.	47
Aggregated result tables of sandbox analysis.	52
<code>poi_ingest</code> submodules and their responsibilities.	53
Per-family corpus coverage.	60
Per-family <i>Indicator of Compromise</i> (IoC) extraction yield.	63
Endpoint recovery by detonation pass.	65
Full SHA-256 hashes of the samples in each operator cluster of Figure 3.	67
Per-signal ingestion coverage. The trajectory population is the denominator for the entire delivery-vector analysis.	72
Head of the installer-filename ranking.	73
Leiden+CPM resolution sweep over the 34,501-host projection.	75
Largest community per lure theme at $\gamma = 0.05$, by representative (highest-PageRank) host. Paths are truncated for width.	76
Top shared delivery infrastructure by the number of distinct communities that link into each host.	77
Head of the last-non-sink lure ranking, by distinct victims, with a manual role label.	81
Three victim-assignment methods converge on the same dominant community and the same dominant build.	81

Acronyms

C&C *Command-and-Control.*

CPM *Constant Potts Model.*

IaaS *Impersonation-as-a-Service.*

IoC *Indicator of Compromise.*

MaaS *Malware-as-a-Service.*

Pol *Point-of-Infection.*

1 Introduction

Commodity infostealers are among the most frequently submitted malware types on public repositories, and are increasingly shifting from dedicated panel-based *Command-and-Control* (C&C) to existing benign infrastructure for data exfiltration. In particular, Telegram-based exfiltration has emerged as the dominant channel of this kind. Creating a Telegram bot is free and operates on a global, durable infrastructure that resists takedown, providing token-based authentication with minimal configuration. Through a single HTTPS request, an entire credential archive can be sent to the operator, and without traffic decryption, the exfiltration is indistinguishable from regular Telegram activity, concealing it from network-based detection systems. Telegram becomes the easiest path for an operator with no budget and no technical operational security expertise. This ease of use does not come without a weakness, though. The credential that identifies the bot, the Bot Token, along with the destination Chat ID, must be embedded directly in the sample's binary. This pair has a recognizable shape, composed of a numeric bot identifier, a colon, and a secret of URL-safe characters for the Bot Token, and a positive or negative number from 10 to 14 digits for the Chat ID. Anyone who recovers these credentials gains access to the operator's channel, since they allow querying the bot and observing past conversations. For this reason, operators frequently rotate these credentials and republish these logs on other Telegram channels and underground markets. These credentials, therefore, bind the Telegram Bot and the Chat not only to the operator behind the campaign but also to the population of victims that the sample infected, through logs sent by the malware itself.

These two perspectives on the ecosystem, the operator-facing binary and the victim-facing log, have both been studied in isolation. Research on the operator side focuses on reverse-engineering samples and extracting configurations, while work on the victim side involves analyzing leaked credentials and their subsequent abuse. Both rely on the same Telegram infrastructure; however, there is no transparent academic methodology that systematically measures the two together using indicators that permit cross-domain correlation. The artifacts with the highest correlation, such as operator identifiers embedded in binaries and infection traces in logs, are still studied in isolation within different research domains.

This thesis approaches the problem as an empirical measurement of the Telegram-based commodity infostealer ecosystem from both sides simultaneously. On the operator side, a host-only pipeline processes and detonates a corpus of samples drawn from public malware repositories and recovers their Telegram IoCs using both static and dynamic techniques, measuring how much operator infrastructure withstands packing and sandbox evasion, and how recovery differs across various malware families. On the victim side, a dedicated pipeline ingests stealer logs and reconstructs the route by which the victim got infected, using the install names and browsing history. This allows for grouping infections by delivery vectors and creating a taxonomy of those vectors, which are then examined against the attributed family and operator brand. Underlying these questions, a broader aspiration drives the work to determine whether the operator side and the victim side can be linked as part of the same campaign.

To guide this research, the following research questions are defined:

- **RQ1:** Across a corpus of commodity infostealer samples, selected for documented Telegram C&C capability, what is the extent of Telegram IoCs (Bot Tokens, Chat IDs) that a combined static-dynamic pipeline can recover and validate, and how does recovery vary by family?
- **RQ2:** To what extent can per-victim log artifacts reconstruct and classify an individual victim's delivery vector?
- **RQ3:** What taxonomy of delivery vectors and shared delivery infrastructure emerges across commodity infostealer victims?

This work concerns Windows commodity infostealers that use Telegram for exfiltration. Stealers that rely only on web-panel C&C, or infect different operating systems, are considered outside of this document's scope. On the victim-side, any handling of sensitive victim data, such as credentials, also falls outside of scope.

The remainder of the thesis is organized as follows. Chapter 2 reviews the commodity infostealer ecosystem, Telegram's role as both a C&C channel and a wider cybercrime infrastructure, the stealer-log markets, and the measurement, extraction, and ethical frameworks the work builds on. Chapter 3 describes the research design, covering corpus construction, the host-only laboratory, and the IoC extraction, log ingestion, and delivery-vector methodologies. Chapter 4 describes the implementation of the analysis pipelines, and Chapter 5 reports the results. Chapter 6 concludes and outlines future work.

2 Background and Related Work

Over the past decade, the commodity infostealer ecosystem has shifted from tools that individual threat actors developed and deployed toward a *Malware-as-a-Service* (MaaS) supply chain [1]. Organized teams distribute and coordinate payload deployment, log collection, and revenue sharing through Telegram channels and groups [2], while also using the platform’s Bot API as a free, persistent, and authenticated exfiltration channel. Without deep packet inspection, the resulting traffic is indistinguishable from ordinary Telegram usage [3]. The resulting infrastructure embeds operator identifiers such as Bot Tokens and Chat IDs directly into the malware binary, and no study has systematically examined how these operator-side identifiers relate to the victim-side telemetry captured in stealer logs. On the victim side, the credentials, cookies, and machine fingerprints harvested from victims feed a structured secondary market whose dynamics have been examined in recent academic work. Both sides of the ecosystem run on a common infrastructure layer, yet they have not been connected empirically. This chapter surveys the literature behind that question. It establishes the ecosystem context in Sections 2.1 through 2.5 and the state of the art in IoC extraction in Section 2.7. Sections 2.6 through 2.4 set out the methodological foundations the study builds on: the empirical-measurement tradition, the analysis and anti-analysis techniques the dynamic methodology must overcome, and the web-delivery and victim-path literature underlying the victim-side analysis. Section 2.9 synthesizes the gaps that position the contribution of this thesis.

2.1 The Commodity Infostealer Ecosystem and Telegram-Based C2

Early password stealers were typically monolithic tools created and used by individual threat actors. In contrast, modern infostealers are mainly distributed via organized MaaS platforms. Patsakis et al. [1] describe this ecosystem, documenting in detail a multi-layered supply chain that includes developers who sell or rent stealer builders, affiliates who tailor and distribute payloads, initial-access brokers providing victim environments, and monetization services that turn stolen credentials into revenue. Since MaaS reduced the technical difficulty of operating an infostealer, there has been a significant increase in both the number and variety of active stealer families.

Empirical research provides a broader understanding of the scope. Avgetidis et al. [4] examined HTTP-managed stealer campaigns over 20 months and found operator profit margins above 90 %, which they attribute to the low cost of MaaS subscriptions relative to the market value of exfiltrated credentials. Their taxonomy of stealer management interfaces differentiates between web panel-based families, such as Formbook and AZORult, which exfiltrate data to attacker-controlled HTTP endpoints, and an emerging class of stealers that avoid traditional web infrastructure entirely by using instant messaging platforms as Command-and-Control channels. Sekoia’s Traffers research [5], based on eight months of monitoring major cybercrime forums, further documented this shift by identifying organized distribution teams with over 1,000 members that coordinate malware activities via Telegram channels. KELA [6] reports 4.3 million machines infected in 2024, with Lumma, StealC, and RedLine responsible for over three-quarters of infections.

Bitsight [3] monitored approximately 1,800 Telegram bots known for infostealer exfiltration usage and identified over five million victim records transmitted through this channel, with activity beginning as early as 2020 and intensifying from 2022 onward. Since each bot is uniquely identified by a Bot Token and a Chat ID specifying the destination channel or group, together these two artifacts constitute high-value operational identifiers that link individual samples to shared threat actor infrastructure. Cofense [7] reports that Agent Tesla alone makes up 77.7 % of the 2024 samples observed using the Telegram Bot API for exfiltration, making it the most prominent example of this threat type.

Agent Tesla is among the most widely deployed families exploiting this exfiltration pattern. It is a .NET-based keylogger and credential stealer that supports four distinct exfiltration protocols: FTP, SMTP, HTTP, and Telegram Bot API. A recent hybrid analysis of Agent Tesla [8] describes its multi-stage loader and decryption routines. The broader academic literature on Telegram’s role in cybercrime has been advanced by Roy et al. [2], whose large-scale analysis of 339 cybercriminal channels with over 23 million subscribers found that 38 % of shared executables were bundled with malware, with Telegram serving simultaneously as a distribution platform, a management interface, and a C&C communication channel. Beyond academic studies, the per-family literature is dominated by vendor reporting: Sekoia [9] presents an in-depth teardown of Raccoon Stealer v2, ESET [10] documents a 369 % detection increase for Lumma between the first and second half of 2024, and AhnLab ASEC [11] analyses 28.2 million infection cases between August and October 2024 across RedLine, Raccoon, Vidar, and Meta. Since peer-reviewed, per-family measurement studies remain limited, academic research on commodity stealers falls behind operational developments and leaves room for the contribution this thesis aims to make.

On the operator side, identical Tokens and Chat IDs are hardcoded into every sample generated by a given builder configuration. Thanks to this characteristic, their C&C infrastructure can systematically be extracted. On the victim side, instead, the artifacts that bots collect appear in the stealer logs they exfiltrate. Recovering these identifiers across a corpus makes it possible to study the ecosystem as a whole rather than one sample at a time.

2.2 Underground Markets for Stealer Logs and Stolen Credentials

Stealer logs extend beyond the initial operator. The credentials, cookies, and machine fingerprints harvested from victim hosts become commodities that circulate within a structured secondary market.

Campobasso and Allodi’s analysis of Genesis Market (referred to pseudonymously in their work as ‘IMPaaS.ru’) [12] is the foundational academic study of structured stolen-credential markets and the phenomenon they call *Impersonation-as-a-Service* (IaaS). Full victim profiles are sold at prices affordable enough to enable large-scale session impersonation, greatly reducing the barriers created by multi-factor authentication. The same authors further extend this work in [13], analyzing 161 days of active purchase activity on “IMPaaS.ru” market. Of the 11,357 observed profiles, 1,193 were sold, and a regression model achieved $R^2 = 27.8\%$ in explaining purchase decisions based on observable profile attributes. The study highlights the almost total lack of Russian

and CIS victims in the sold population, a pattern consistent with the typical behavior of Russian-speaking cybercrime groups. These two papers outline both the demand-side dynamics and the empirical methods used to analyze the stolen credential market.

The most directly relevant academic precedent for this work is Nurmi et al. [14], who examine three published datasets, including 245 stealer log dumps from 14 malware networks, totaling 1.8 million victim files. Their research traces the supply chain from infostealer infection to marketplace sale, reporting a median Genesis Market profile price of around \$5.

Soska and Christin [15] examine 16 anonymous marketplaces over more than two years, show how the ecosystem survives law-enforcement takedowns, and set out a longitudinal measurement method that later studies reuse. Van Wegberg et al. [16] extend the approach across eight marketplaces over six years, measure the commoditization of cybercrime, and single out stolen accounts as a prominent retail category. Industry reports provide insights into the key marketplaces for commodity infostealers: Sekoia [17] surveys the Russian-speaking log-trading ecosystem and documents log anatomy and pricing ranging from \$5 to \$100. Meanwhile, Secureworks [18] reports a 670 % increase in stealer logs sold on the Russian Market from June 2021 to May 2023, and notes that stolen credentials were the initial-access vector in 34 % of ransomware incidents during the same period.

2.3 Stealer Log Analysis and Victim Telemetry

Thomas et al. [19] ran a large empirical study of credential theft, measuring 788,000 keylogger victims, 12.4 million phishing victims, and 1.9 billion breached credentials over one year using Google account telemetry. They describe a clear risk hierarchy: phishing has the highest likelihood per victim of leading to an account compromise, followed by keyloggers, then breach corpora. The authors note that 7 % to 25 % of exposed passwords correspond to active Google accounts. This study establishes the current empirical baseline for credential theft research and emphasizes the importance of differentiating the collection method from subsequent misuse. Although Thomas et al. operate at a scale and with data access unavailable to independent researchers, their approach of measuring the entire pipeline from collection to abuse is directly applicable to the present study.

A related area of research involves using honey accounts to monitor how leaked credentials are exploited downstream. Onalapo et al. [20] released 100 Gmail credentials via paste sites, underground forums, and malware-infected devices, then monitored attacker activity over seven months. Their study shows that different channels attract distinct attacker groups and that the subsequent use of the credentials can be partly anticipated based on how they were initially exposed. Bermudez Villalva et al. [21] extend this methodology to compare credential exploitation on the dark web and surface web, identifying distinct patterns of malicious activity by distribution channel. Both studies are important precursors: they show that victim-side telemetry holds recoverable information about the upstream supply chain. Both, however, work from infection toward abuse rather than

backward from log to binary.

Holz et al. [22] studied the architectural predecessor of the modern stealer-log dropzone by analyzing 70 keylogger dropzones, 33 GB of harvested data, and over 173,000 victims across seven months. Despite the technological advances since 2009, hardcoded HTTP dropzones function similarly to Telegram bots today. The core process of regularly exfiltrating organized victim data to operator-controlled storage has remained the same.

2.4 Web-Based Malware Delivery and Victim-Path Reconstruction

The victim-side analysis in this thesis reconstructs how a victim reached the payload from the browsing history recorded in the stealer log. This restates a question from web security: given the visited web pages, which one was the source of the malware? Provos et al. [23] introduced the concept of malware delivery trees, representing the sequence of sites a browser visits before reaching the distribution host, modeled as a tree rooted at the payload instead of a simple list of pages. This perspective justifies viewing victim trajectories as a graph rather than a partition.

Later work recovers the delivery structure from the browsing of many victims rather than from a single trace. Stringhini et al. [24] identify malicious pages by observing how many users follow similar redirection paths, indicating collective behavior. This approach relies on pooling partial trajectories from different victims to reveal common infrastructure. Taylor et al. [25] apply tree-based similarity to analyze 688 million redirection chains, demonstrating that graph-structured delivery analysis can scale to large victim populations.

The reconstruction also uses web usage mining, in which navigation sequences are grouped and analyzed for common patterns. Banerjee and Ghosh [26] introduced order-preserving similarity over clickstreams, while Pei et al. [27] developed the PrefixSpan algorithm for sequential pattern mining, enabling the discovery of frequent ordered subsequences without exhaustively enumerating all candidates.

2.5 Telegram as Cybercrime Infrastructure

Beyond its role as a Command-and-Control channel, Telegram is also a marketplace, a distribution channel, a community space, and a coordination layer within the cybercrime ecosystem.

Bijmans et al. [28] collect 70 phishing kits from the underground economy and detect 1,363 associated phishing domains in the wild, which shows Telegram working as a distribution channel for ready-to-deploy attack tooling. Roy et al. [2] present the most comprehensive published characterization of cybercriminal activity on Telegram to date, tracking 339 channels with over 23 million aggregate subscribers across five content categories: credential compromise, pirated software, pirated media, social-media manipulation, and blackhat resources. Two of their structural findings expand the operational understanding beyond the C&C channel role. First, they identify a layer of purpose-built bots that mediate between channels and users in three functional roles: payment gateways built

on the Telegram Payment API, content-access bots that deliver files outside the public channel feed, and follow-to-access bots that enforce join-multiple-channels funnel conditions. This demonstrates that operator-side Telegram infrastructure extends significantly beyond the Bot API as C&C pattern that is the focus of this thesis. Second, they empirically assess platform resilience: of the 339 channels reported to Telegram through evidence-based disclosure, only 64 (19 %) were removed. Among the removed channels that posted alternates, a median of 43.8 % of subscribers migrated within a week. Taken together, these findings show that Telegram-based cybercrime is deeper and more resilient than its C&C channel role alone would suggest.

Garkava et al. [29] analyze 16 Telegram channels and groups functioning as stolen-data markets, detailing their operational rules, advertising strategies, and trading procedures. This work offers a valuable counterpoint to vendor stories that suggest Telegram trading is unstructured. Marjanov et al. [30] conduct a one-year study of 1,521 manually curated stolen-data channels, 14 million messages, and 3.6 million shared files. Their analysis identifies infostealer logs as a distinct and growing commodity on the platform, increasing from approximately 15 % of offerings in 2021 to a sustained 30 % by 2022 with further growth through mid 2025. The study also highlights a layer of subscription-based "Cloud" channels managed through payment and delivery bots, which form the consumption side of the same Telegram Bot API infrastructure seen on the operator side of the ecosystem. To better understand Telegram's role in the wider cybercrime environment, Allodi et al. [31] analyze the functions and trust-signaling mechanisms within 543 Telegram channels and groups. They find that Telegram's internal trust systems are too weak to support the independent trade of high-impact, innovative technological threats. However, the platform serves as an important auxiliary infrastructure for traditional underground forums. Instead of replacing these markets, Telegram acts as a service layer: bots handle delivery, private channels facilitate vendor-client communication, and public channels promote marketplace updates.

Telegram-as-infrastructure abuse is not limited to commodity actors. Cisco Talos [32] documents Operation Blacksmith, in which the North Korean Lazarus Group deploys NineRAT, a Dlang-based Remote Access Trojan (RAT) that uses the Telegram Bot API for C&C. Recently, cyber actors linked to Iran's Ministry of Intelligence and Security (MOIS) have been actively using Telegram as C&C infrastructure to deploy multi-stage malware targeting Iranian dissidents, journalists, and opposition groups worldwide [33]. These cases show that this technique is not limited to commodity malware and that operator-side Telegram use is robust enough to support even high-stakes APT campaigns. This same robustness also explains why Telegram remains so common in the commodity cybercrime ecosystem.

2.6 Empirical Malware Measurement Methodology

The measurement study presented in this thesis follows an established tradition with explicit standards for corpus construction, infrastructure analysis, and reporting. This section documents the methodological references the work adopts.

The primary reference is Rossow et al. [34], whose survey of 36 malware-experiment publications found that a quarter relied on problematic dataset assumptions, including selection bias, unstated provenance, and family imbalance, and that 71 % reported no containment or safety measures. From this survey, the authors derive four guidelines for sound malware experiments: transparency, realism, correctness, and safety.

The corpus construction follows the model set by Malpedia [35], which includes 607 families and 1,792 samples, curated with an explicit allowlist, documented inclusion criteria, and an auditable rejection log. The same authors' earlier study of domain-generation algorithms [36] shows how a single malware class can be measured at the corpus scale through family-by-family analysis.

Lever et al. [37] analyzed network communications from 26.8 million samples over five years, revealing that malware often reuses C&C infrastructure across distinct samples and operators. In this thesis, the present corpus is smaller by several orders of magnitude, and clustering is performed using exact identifier matches rather than the statistical infrastructure-overlap measures that Lever applies at the internet scale.

This thesis does not advance the methodology of malware measurement. It applies these templates to the commodity Telegram-infostealer ecosystem, measured from the operator side and the victim side at once, a setting in which they have not previously been combined.

2.7 IoC Extraction, Configuration Extraction and Normalization

An IoC is a forensic artifact, such as an IP address, domain name, URL, email address, or file hash, that, when present on a network or operating system, indicates a potential compromise, intrusion, or breach. These data points help cybersecurity professionals identify, detect, and respond to security incidents [38]. Automated extraction of configuration data and IoCs from malware samples has developed along two tracks: extraction from unstructured textual sources, such as threat intelligence reports and security blogs, and extraction from binaries and dynamic analysis outputs.

The text-based track is illustrated by iACE [39], which applies Natural Language Processing techniques to security blog articles to identify IoC tokens and their surrounding context. By using dependency parsing and graph mining to validate the grammatical and positional relationships between IoC tokens and their context terms, iACE effectively filters out the high volume of false positives associated with basic regular expression matching. This method achieved a precision of 95 % across 900,000 extracted OpenIOC items from 71,000 articles over a 13-year period. Caballero et al. [40] benchmark seven open-source IoC extraction tools against a ground-truth dataset of 618,217 indicators collected from 472,891 reports over fifteen months, and identify the handling of defanged and obfuscated indicators as the principal factor determining tool effectiveness.

A similar issue affects the binary track. Bot Tokens and Chat IDs in malware samples can be encoded in Base64, obfuscated with XOR, encrypted, or split across configuration structures. Naive regex matching on raw binaries often fails to detect tokens hidden in packed or encrypted payloads unless preprocessing steps are taken. As a result, the

binary analysis usually requires running samples in sandbox environments or using unpacking heuristics to uncover the underlying configuration data before performing string matching.

Although peer-reviewed literature on binary-side configuration extraction is limited, several systems have become de facto standards. The DC3 Malware Configuration Parser framework [41] from the U.S. Department of Defense Cyber Crime Center offers per-family parser modules, standardized JSON output, YARA-triggered execution, and a REST API, and is the longest-established open-source configuration extraction framework. CERT Polska’s Karton framework [42] provides production-grade, microservice-orchestrated extraction that processes samples from unpacking through extraction to storage. Within CAPEv2 [43], the integrated config-parser library covers many families and operates from YARA-triggered breakpoints during dynamic analysis.

VMware’s published study of Emotet [44] demonstrates the analytical value of large-scale extraction. It shows that 2,181 extracted payloads can provide an overall understanding of C&C infrastructure, including IP distributions, JARM fingerprints, port distributions, and AS-number clusters. This pattern, extracting structured configuration data across many samples and then analyzing the aggregate infrastructure it reveals, is what the configuration-extraction track makes possible, yet the peer-reviewed literature has rarely applied it systematically to any single malware family.

2.8 Malware Analysis and Anti-Analysis Techniques

The measurement study in this thesis depends on recovering operator identifiers from samples that actively resist inspection. This section reviews the two analysis paradigms the work relies on and the evasion techniques that constrain them.

Malware analysis is divided into two complementary paradigms. Static analysis examines a binary without executing it, and dynamic analysis observes behavior at runtime in a controlled environment [45]. Static analysis is fast and carries no execution risk, but it cannot read code that is encrypted or packed until that code is unpacked in memory. Dynamic analysis exposes the runtime behavior that static methods miss, but it depends on persuading the sample to execute its payload, which an evasive sample refuses to do once it detects the analysis environment [46].

The limits of static analysis are well established. Christodorescu and Jha [47] showed that simple obfuscation defeats signature matching, and Moser et al. [48] proved that constructs such as opaque constants place fundamental limits on what static analysis can determine about a program. For the commodity Telegram infostealers examined here, the effect is straightforward: packing typically strips Bot Tokens and Telegram strings from the distributed binary. As a result, static analysis of the shipped sample does not reveal any operator identifiers, and the configuration remains hidden until the sample unpacks itself during runtime.

Dynamic analysis addresses this by executing the sample in an instrumented sandbox and recording its interaction with the operating system. Egele et al. [49] survey the available techniques, which range from function-call monitoring through information-flow tracking and full instruction tracing to the monitoring of

persistence points. Function-call monitoring through API hooking is the technique most relevant to this work, since it underlies the instrumented detonation used to capture behavior and extract configuration.

Instrumentation is also the property that evasive malware targets. Afianian et al. [50] survey the techniques commodity malware uses to detect analysis: checks for virtual-machine artifacts, detection of userland API hooks and injected monitoring modules, and probes of the surrounding environment for signs that it is artificial. A sample that identifies any of these suppresses or alters its payload, so the analysis observes benign behavior rather than the malicious path. Environmental realism, therefore, determines whether dynamic analysis succeeds at all. Miramirkhani et al. [51] demonstrated that sandboxes lacking the wear-and-tear of genuine use, such as browsing history, installed applications, and user files, are reliably distinguishable from real machines. These evasion behaviors shape the laboratory design in Chapter 3: a hardened and realistic analysis guest, a second uninstrumented detonation pass that removes the hooks anti-instrumentation samples look for, and a synthetic responder that returns plausible answers to the connectivity and geolocation probes a sample issues before releasing its payload.

2.9 Gap Analysis

The literature reviewed in this chapter highlights three observations that together define the contribution of this thesis.

Recovering Bot Tokens and Chat IDs from binaries is a common industry practice, but it mostly appears as case studies analyzing individual samples [3, 52]. These reports confirm that operator identifiers are embedded in stealer configurations and binaries and can be extracted, but no studies systematically apply this extraction across a labeled, multi-family corpus with a reproducible pipeline and measure the recovery success. The operator-side measurement in this thesis addresses this.

Second, academic work on stealer logs generally focuses either on tracing the infection to theft and subsequent marketplace sale [14, 19, 20], or analyzing the posts and metadata of trading channels [2, 30]. Marjanov et al. explicitly exclude the binary-producing side of the infrastructure from their dataset, and Roy et al. classify channel posts rather than extracted operator identifiers, so neither anchors its observations to specific samples. The reconstruction of per-victim delivery vectors from the browsing telemetry inside the logs is not addressed. The closest study, Ruellan et al. [53], recovers delivery vectors from infection screenshots and itself notes that the reactive analysis of stealer logs and their artifacts remains a gap. The victim-side measurement in this thesis addresses this.

Third, the operator side and the victim side are studied apart from each other. The one effort that bridges them, Bitsight’s collection of victim telemetry from live bots using tokens harvested from binaries [3], appears in a blog post with an undisclosed collection methodology and a proprietary corpus. No transparent and reproducible academic methodology measures both sides on a common basis. This thesis measures the two modalities together and reports, as a structural finding, the extent to which the operator-side and victim-side corpora can and cannot be

joined.

The contribution consists of three parts: a reproducible method for operator-side loC recovery, a taxonomy of delivery vectors from the victim's perspective, and an analysis of the current boundary separating the two.

3 Research Design and Methodology

This chapter presents the research design and methodology of this empirical measurement study on the Telegram-based commodity infostealer ecosystem. The study operates across two complementary data modalities that characterize the same criminal infrastructure: *operator-side* binary samples sourced from public malware repositories, and *victim-side* stealer logs provided by Constella Intelligence ¹. These logs were collected from public Telegram Channels. Each data source provides a different analytical perspective on the same underlying criminal infrastructure: the binary corpus provides operator-facing identifiers extracted through static and dynamic analysis, while the stealer log dataset supplies victim-side telemetry.

3.1 Analysis Environments

This study uses two analysis environments: a manual analysis lab and an automated sandbox lab built on the CAPE framework. The two are not interchangeable. Three samples are used in the manual lab as a methodological baseline. These samples are manually analyzed to demonstrate how Telegram is used as a C&C channel and to demonstrate the extraction of relevant IoCs through both static and dynamic analysis. This manual analysis grounds the design of the automated extractor, establishing which artifacts carry the bot token and chat identifier and how they are recovered. The CAPE-based environment then operationalizes this logic by automatically processing the entire corpus using a uniform and reproducible pipeline. Therefore, the manual lab illustrates the depth of analysis achievable on individual samples, while the automated lab quantifies the extent to which these findings persist at the corpus scale.

3.1.1 Ethical Isolation and the Evasion Trade-off

Both environments operate in a host-only and air-gapped configuration. This setup is required for ethical reasons to prevent samples from interacting with live attacker infrastructure. However, this kind of isolation poses a major challenge for dynamic orchestration. Commodity infostealers frequently conduct reachability and anti-analysis checks before releasing their payloads. Therefore, they can easily detect sandboxes that silently drop outbound traffic or resolve all queried names, thereby suppressing the malicious behavior on which IoCs are based.

3.1.2 Synthetic Response Strategy

To prevent malware from detecting the sandbox environment, a synthetic responder intercepts and responds to IP geolocation and network-based VM detection queries, providing well-formed responses that allow samples to reach

¹<https://constella.ai/>

the exfiltration stage. Exfiltrated payloads are captured and stored for subsequent IoC extraction. This mechanism allows for the recovery of Telegram Bot Tokens and Chat IDs across the corpus without any live network contact.

The manual lab and the CAPE lab serve two different purposes; therefore, the synthetic responder is implemented with different tools. INetSim, PolarProxy, and nginx were selected for the manual lab for a deep, exploratory analysis of individual samples; mitmdump and a pair of custom DNS and HTTP responders were selected for CAPE to efficiently process a corpus of infostealers at high throughput.

3.2 Corpus Selection and Construction

3.2.1 Source Rationale

During the design phase, several publicly accessible malware repositories were evaluated as potential binary sources: VirusTotal [54], Hybrid Analysis [55], MalwareBazaar [56], Malpedia [35], VirusShare [57], and MalShare [58]. Three criteria drove the selection: (i) structured family-level metadata accessible via API, (ii) unrestricted bulk download without the need for a commercial agreement, and (iii) sufficient community attribution to support a reproducible corpus construction.

VirusTotal offers the broadest coverage among public repositories; however, sample downloads require a paid Intelligence subscription. It was therefore excluded. Hybrid Analysis provides sandboxed behavioral reports and a public API, but its family attribution metadata is limited, and downloads are subject to a vetting policy. Malpedia is a well-curated family intelligence resource with invitation-only registration. VirusShare and MalShare both maintain large binary collections but lack sufficient structured metadata, necessitating manual classification of each candidate sample and introducing reproducibility risks.

MalwareBazaar was selected as the exclusive binary source. Operated by abuse.ch, it is freely accessible and community-curated, providing active malware samples along with rich metadata, including family signatures, community tags, file types, and cryptographic hashes [56]. The MalwareBazaar REST API supports direct filtering by both `signature` and `tag` fields, which enabled the 33-strategy query design described in Section 3.2.2: per-family tag queries, per-family signature queries, and a community Telegram tag. Each sample record contains a `first_seen` timestamp, facilitating the temporal analysis of the corpus. Samples are distributed as password-protected ZIP archives using a standardized passphrase (`infected`), which makes automated bulk downloads secure and simple. Any researcher with an abuse.ch account and the same family allowlist can reconstruct the candidate pool using identical API queries, thereby ensuring corpus reproducibility.

A notable limitation of this selection must be explicitly acknowledged. MalwareBazaar is community-driven, meaning that sample submissions reflect the priorities and capabilities of its contributors rather than a systematic representation of commodity infostealers' distribution in the wild. In fact, certain families or geographies may be over-represented relative to their actual prevalence. While this submission bias affects the generalizability of prevalence

findings, it does not invalidate the study’s primary contribution.

MalwareBazaar’s holdings change over time, so the same queries run at a later date return a different set of candidates. The corpus was queried on *April 30th, 2026*.

3.2.2 Family Selection and Query Design

The corpus is constructed based on two inclusion criteria. The primary condition requires family-level confirmation of the Telegram Bot API as an exfiltration channel. This confirmation is accepted at two evidence levels, corresponding to the two classes in Table 3.1. Class T1 (*Telegram-native*) requires at least one authoritative source that confirms Telegram Bot API exfiltration as a family feature, including published or leaked source code, threat intelligence reports, vendor blog posts, or security advisories. Class T2 (*Telegram-capable*) applies when no family-level documentation exists, but individual samples attributed to the family carry direct sample-level evidence of Telegram capability. T2 families are retained because the absence of family-level reporting would discard samples with demonstrable Telegram activity. The second criterion applies uniformly to both classes: the family must be offered as Malware-as-a-Service (MaaS) to a broad customer base, rather than as a tool attributed to a named threat actor or Advanced Persistent Threat (APT) group.

Applying these criteria to the commodity infostealer landscape yielded a final allowlist of 16 families, shown in Table 3.1, along with the inclusion rationale and supporting references.

Families that rely primarily on panel C&C infrastructure (i.e., RedLine, Vidar, Raccoon, and their successors) were intentionally excluded because their exfiltration mechanism falls outside the scope of the Telegram IoC extraction pipeline. In fact, families without a confirmed Telegram exfiltration channel would inflate the corpus with samples that are structurally incapable of producing the signals under investigation.

Table 3.1 Malware family acquisition allowlist.

Family	Inclusion rationale
<i>Class T1 – Telegram-native Bot API exfiltration documented at family level</i>	
Agent Tesla	.NET RAT; Telegram is one of four configurable C&C protocols (HTTP, SMTP, FTP, Telegram). Cofense reports Agent Tesla accounts for 77.7% of 2024 Telegram-Bot-API samples. [7, 59]
Snake Keylogger	.NET keylogger and credential stealer; Telegram, SMTP, and FTP exfiltration modes selectable at builder time. [60]
404 Keylogger	Snake Keylogger Autolt-based variant that retains Telegram-bot and SMTP exfiltration methods. [61]
Continued on next page	

Table 3.1 – continued from previous page

Family	Inclusion rationale
Blank Grabber	Open-source Python stealer; builder exposes Telegram-bot and Discord-webhook exfiltration alternatives. Cross-validated with a manually analyzed sample. [62]
ToxicEye	Telegram-native RAT written in C#. [63]
Masad Stealer	Autolt scripts compiled into a Windows executable. Identified as the first commodity infostealer to operationalize Telegram for C&C in 2017. [63]
StormKitty	Open-source .NET stealer with Telegram-only exfiltration; parent codebase for multiple variants such as Prynt, WorldWind, and others. [64]
Prynt	.NET stealer derived from AsyncRAT and StormKitty that uses Telegram exfiltration. [65]
WorldWind	Open-source Prynt derivative with Telegram-only exfiltration. [65]
Phemedrone	Open-source C# stealer; builder selects between HTTP panel and Telegram bot. Telegram is the primary observed C&C across the analyzed sample sets. [66, 67]
XWorm	.NET trojanized RAT; uses Telegram as C&C. [68, 69]
Phantom Stealer	.NET-based infostealer sold commercially as part of the "Phantom Project"; exfiltrates through Telegram or SMTP. Over a period of five months, a researcher operating under the handle <i>0x6rss</i> empirically collected telemetry from commodity infostealers, finding that over 39% of all victims were from Phantom Stealer. [70]
<i>Class T2 – Telegram-capable, retained on sample-level evidence despite non-primary Telegram attribution</i>	
AsyncRAT	C# RAT. Default C&C uses a custom TCP/TLS protocol; however, AsyncRAT-tagged logs are observed in the Bitsight Telegram-bot dataset, and modified variants (e.g. the Desert Dexter campaign) implement Telegram exfiltration. [3]
DeerStealer	Multi-stage infostealer written in Delphi that exfiltrates to a separate C&C over HTTP POST. Campaigns embed a Telegram bot token in the binary and POST execution-stage notifications via curl, producing the same analysis artifacts as Class T1 families. [69]
Lumma (LummaC2)	C++ and ASM stealer with primary exfiltration via HTTP POST to operator-controlled panels. Retained because recent Lumma samples are observed retrieving C&C configuration from Telegram channels, therefore producing detectable Telegram-API artifacts during analysis time. [69]
Continued on next page	

Table 3.1 – continued from previous page

Family	Inclusion rationale
Umbral Stealer	C# infostealer. Public builder exfiltrates via Discord webhooks. Retained because a manually analyzed sample carrying the MalwareBazaar tag <code>UmbralStealer</code> exhibited Telegram-Bot-API content, evidencing forked variants in the wild. [71]

Most families in the list are implemented in .NET or C#. The .NET runtime dominates the commodity infostealer market because accessible tooling and widely available builders lower the barrier for new operators. The remaining families are developed in Python, Autolt, Delphi, and even C++ and ASM.

The list also contains a lineage cluster centered on StormKitty. Along with Prynt and WorldWind, these families share a Telegram-only exfiltration design but correspond to distinct operator populations, so all three are retained. The same applies to Snake Keylogger and its variant 404 Keylogger, which is an Autolt-packed re-implementation. Because these labels come from MalwareBazaar's own tags and signatures, and code relatives of this kind are often cross-attributed by reporters, any per-family result reported later is read as conditional on that labeling rather than as ground-truth family behavior.

Telegram usage patterns vary across families. Some families use Telegram as the sole exfiltration channel, while others expose it as one of several options that can be configured at builder time. Within Class T2, the pattern is even more diverse: some AsyncRAT variants route traffic through Telegram instead of the default TCP/TLS channel; DeerStealer embeds Bot Tokens for execution-stage notification rather than for data exfiltration but produces the same analysis artifacts; LummaStealer retrieves its C&C panel configuration by querying Telegram.

The allowlist is deliberately narrow. Families were excluded if Telegram usage was unconfirmed at both the family and sample levels, or if they fell outside the Windows user-mode scope of this study. The resulting selection covers the dominant commodity infostealer families active during the study period, as corroborated by concurrent threat intelligence reports [3, 7].

The 16-family allowlist maps to a 33-strategy query design on MalwareBazaar. Two independent queries are issued per family: a `tag:<family>` query retrieves samples where community reporters applied the family label as a tag, and a `signature:<family>` query retrieves samples where MalwareBazaar's attribution pipeline matched the family by signature. This dual approach is necessary because MalwareBazaar is inconsistent across its reporter base: a sample may carry a `signature` attribution without the corresponding community `tag`, or vice versa, and relying on either field alone would silently exclude a portion of the attributed samples. A thirty-third strategy, `tag:telegram`, captures all samples reporters have tagged with the platform keyword, regardless of family attribution. This provides a catch-all for Telegram-attributed samples that fall outside the allowlist or lack a family label.

Each binary is uniquely identified by its SHA-256 hash, so a sample returned by multiple strategies is recorded

only once, preventing the corpus from being inflated by query overlap. The set of strategy identifiers that discovered each sample is preserved as provenance metadata, making it possible to quantify the marginal coverage of individual strategies. In particular, it is useful to determine how many samples the `tag:telegram` cross-tag query contributed that the sixteen per-family sweeps would have missed entirely.

3.2.3 Scope Definition and File-Type Filtering

The scope of this study is limited to samples targeting the Windows user-mode execution environment, which represents the predominant platform for commodity Telegram-based infostealers and aligns with the constraints of the analysis infrastructure. File-type filtering is implemented as a secondary step following candidate discovery, and prior to binary download. This sequence preserves API quota, avoids downloading files that would be unconditionally rejected, and maintains a complete audit trail of every exclusion decision.

Four file categories, as shown in Table 3.2, are retained for analysis. Native Windows executables and libraries (`exe`, `dll`, `msi`) are admitted directly to the CAPE sandbox without preprocessing. Windows script formats (`ps1`, `js`, `jse`, `vbs`, `vbe`, `bat`, `cmd`, `hta`, `wsf`, `lnk`) are included because commodity infostealers frequently use script-based droppers and loaders as delivery-stage components, and the analysis environment supports their execution. Python and Python-compiled formats (`py`, `pyc`) are retained to facilitate the analysis of PyInstaller-packaged Python stealers. Archive formats (`zip`, `rar`, `7z`, and others) are flagged with `needs_unwrap` but still admitted, and are submitted to the sandbox unmodified, which extracts the contained payload when it detonates them (Section 3.3.2).

Table 3.2 File Type Admission Categories.

Category	Extensions	Pipeline Handling / Environment Context
Native Windows	<code>exe</code> , <code>dll</code> , <code>msi</code>	Directly executable by CAPE.
Windows Scripts	<code>ps1</code> , <code>js</code> , <code>jse</code> , <code>vbs</code> , <code>vbe</code> , <code>bat</code> , <code>cmd</code> , <code>hta</code> , <code>wsf</code> , <code>lnk</code>	Common delivery-stage components supported by CAPE.
Python	<code>py</code> , <code>pyc</code>	PyInstaller-packaged supported by analysis VM.
Archives	<code>zip</code> , <code>rar</code> , <code>7z</code> , <code>arj</code> , <code>tar</code> , <code>gz</code> , <code>xz</code> , <code>cab</code> , <code>lzh</code> , <code>ace</code> , <code>z</code> , <code>r00</code> , <code>r09</code> , <code>iso</code> , <code>img</code> , <code>vhd</code> , <code>x</code>	Flagged with <code>needs_unwrap</code> ; submitted to the sandbox unmodified, which extracts the payload at detonation.

Seven file categories are excluded as out of scope, as detailed in Table 3.3. Android packages (`apk`), MacOS binaries (`macho`, `dmg`), and Linux executables (`elf`, `sh`) are omitted because the analysis infrastructure is designed exclusively for Windows, and no allowlisted family is known to deploy cross-platform payloads within the same campaign. Java archives (`jar`) are excluded because the sandbox configuration does not include the Java Virtual Machine. Browser extensions (`crx`) and web documents (`html`, `pdf`, `chm`) are considered low-signal delivery

artifacts rather than primary payloads. Microsoft Office formats are similarly excluded.

Table 3.3 File Type Exclusion Categories.

Category	Extensions	Rejection Reason
Java	jar	JVM not in scope; sandbox configuration lacks a Java Runtime Environment.
Android	apk	Out of scope: analysis infrastructure targets Windows user mode exclusively.
macOS	macho, dmg	Out of scope: analysis infrastructure targets Windows user mode exclusively.
Linux	sh, elf	Out of scope: analysis infrastructure targets Windows user mode exclusively.
Browser Extensions	crx	Low-signal delivery artifact rather than a primary payload.
Web Documents	html, shtml, pdf, chm	Low-signal delivery artifacts rather than primary payloads.
Office	doc, docx, xls, xlsx	Office execution is not present in the current analysis environment.

3.2.4 Sample Admission

Due to MalwareBazaar's community-driven approach, every downloaded sample undergoes three file-type-agnostic sanity checks before being admitted to the evaluation corpus. These checks are designed to eliminate trivially malformed or operationally implausible files.

1. **Minimum file size.** The extracted file must be at least 512 bytes. Samples below this threshold are unlikely to be functional malware and are more likely to be corrupt downloads or stub artifacts; therefore, they are rejected as `too_small`. The threshold was set at 512 bytes rather than a more aggressive value to avoid excluding legitimate small script files: `.lnk` shortcuts and minimal `.bat` launchers can be a few hundred bytes while still being analytically meaningful.
2. **Maximum file size.** The second check imposes a maximum file size of 100 MB. Samples above this threshold are rejected as `too_large`. The 100 MB ceiling is a safeguard against corrupted or mislabeled archives, and files above this threshold would exhaust analysis infrastructure resources.
3. **Readability.** The file must be readable in binary mode without raising an I/O error. Files that fail this check

are often corrupt or truncated and therefore unprocessable, regardless of their assigned family.

All three checks apply equally to PE binaries, Windows scripts, and other files, ensuring a uniform admission criterion across file types. The samples that pass all three sanity checks are admitted in the corpus for analysis; those that fail are moved to a rejection directory and recorded as rejected samples with their respective reason (`too_small`, `too_large`, or `unreadable`).

3.3 Sandbox Orchestration

3.3.1 Role and Scope

The orchestration stage operates after corpus selection: it receives an admitted sample and detonates it within an instrumented sandbox under a fixed analysis policy to generate a comprehensive archive of the resulting analysis artifacts. This stage does not interpret the artifacts; it leaves that to the extraction stage. The orchestrator ensures only that the necessary raw data is present, systematically organized, and accompanied by sufficient provenance to support downstream trust.

This separation of duties is reflected in an architectural decision: the sandbox is a black box that receives submissions and returns a verdict, while the orchestrator lives outside it and communicates with it only via the REST API. The analysis policy, the archive structure, and failure handling are centrally defined at the controller level rather than spread across the various sandbox configurations. This also ensures that the orchestration logic does not need to be changed during sandbox restarts or upgrades.

An admitted sample may be an archive container rather than a directly executable file. Such samples are submitted to the sandbox without modification and detonated directly, with payload extraction delegated to CAPE rather than handled in a separate preprocessing stage.

3.3.2 Archive Containers and Payload Identity

Detonation of an archive within the sandbox produces two distinct identities: one for the archive itself and one for its content. This happens because the corpus identifies the sample by its SHA-256 hash at admission, which refers to the archive's hash, while the sandbox executes the internal payload with its own unique hash. Without explicit linkage between these two, the dynamic analysis results pertain to a payload unrecognized by the corpus, and multiple archives containing the same payload appear unrelated.

To establish this linkage, the analysis processes for each archive sample a record of the hash of the executed payload and the outcome of the extraction attempt. Extraction outcomes vary across the corpus. The sandbox can open most containers and execute the internal payload; however, if the archive is password-protected and the password is unavailable, the sandbox detonates the container itself, which doesn't produce the desired results.

3.3.3 Two-Pass Detonation

Within the CAPE lab, each sample undergoes a two-pass analysis protocol. The first pass, *standard*, runs under full CAPE instrumentation, enabling API call monitoring, behavioral logging, and targeted memory extraction. AMSI buffer interception captures deobfuscated script content at runtime, while the unpacker module extracts and reconstructs injected or unpacked payloads for subsequent static analysis. This pass generates behavioral telemetry and collects IoCs from samples that tolerate monitoring. The second pass, *no_monitor*, operates without any instrumentation, allowing unrestricted execution and capturing evidence only from network traffic and files dropped to disk.

This design addresses two complementary limitations of instrumented analysis. Instrumentation techniques such as API hooking and DLL injection may disrupt sample execution, resulting in crashes, altered control flow, or incomplete detonation attributable to the analysis environment rather than the sample itself. Furthermore, a significant proportion of commodity infostealers employ sandbox-awareness techniques to detect userland hooks, injected monitoring modules, or other instrumentation artifacts. Upon detection, these samples suppress or redirect execution, resulting in incomplete behavioral profiles under full instrumentation. The two-pass protocol mitigates these limitations by collecting behavioral telemetry in the first pass for cooperative samples and in the second pass for non-cooperative samples, for network-observable IoCs.

3.3.4 Viable Passes and Evidence Preservation

Each pass has a minimum evidence requirement that defines what constitutes a successful analysis. The standard pass is considered successful if a CAPE report is produced; the second pass is considered viable if any of the following are present: a network capture, an HTTP archive, or a dropped-files directory. A pass that does not meet its minimum evidence requirement is distinguished from a sample that simply produces no output.

Even if a detonation fails, any partial output is archived. Usable captures or dropped payloads often occur in failed passes, and these fragments are still considered valid evidence. Therefore, each archived pass contains a metadata record that describes the sample, the pass, and relevant timestamps, so that the archive is interpretable without reference to the controller's database.

3.3.5 Layered Analysis Stages

In addition to the two dynamic passes, two further stages of analysis are defined. The static stage is performed once per sample prior to either dynamic passes. Static indicators such as file hashes, string extraction, YARA matches, and capability assessment are recorded regardless of how the sample behaves under dynamic analysis, so a sample that crashes or evades the sandbox still contributes static evidence. Moreover, running the static analysis tools once outside CAPE avoids duplicating expensive computation across both passes.

Following each dynamic pass, a dedicated payload analysis stage processes every file produced by that pass,

including files dropped to disk, process dumps, and memory-extracted blobs. To do so, the same string extraction, YARA scanning, and capability assessment used in the static stage are applied. Commodity Telegram-based infostealers frequently store their operator-facing identifiers inside encrypted or packed payloads that are only decrypted at runtime. These identifiers are not present in the raw sample binary and are therefore invisible to the static stage, but they become accessible after the dynamic pass, when the sample unpacks and decrypts itself, and drops the malicious payloads to disk or extracts them into memory. The payload analysis stage is therefore the primary source for a whole class of IoCs that neither static analysis nor execution-level analysis would otherwise capture.

The pipeline structure across all four stages is as follows: static analysis of the original binary, then each dynamic pass with its payload analysis run immediately after.

3.3.6 Resumability and Fault Isolation

A corpus run can take several hours of wall time, and failures can occur in the sandbox, the virtual machine, or the host system during that time. This means that each run must be treated as a restartable batch process with a durable record of per-pass state. Each pass goes through a lifecycle: queued, submitted, running, and finally a terminal outcome. State changes are persistently recorded in the controller's database, which serves as the ultimate source of truth. In-progress work is reconciled with the sandbox on restart. If the sandbox does not recognize a pass, it is recorded as an infrastructure failure.

Failures are isolated to the smallest possible failure unit. Transient sandbox and network errors are retried with exponential backoff within a given time budget, while malformed requests are not retried. Any failed pass is re-queued a limited number of times before it is marked as a permanent failure. If a detonation process becomes unresponsive and exceeds its wall-time budget, it is terminated, but the entire corpus run is not stopped. The failure is noted for that pass, and the whole process continues.

3.3.7 Containment and Hygiene

Detonation is performed in an air-gapped laboratory with simulated network services (Section 3.1.1), and each pass starts from a known-clean virtual machine snapshot. The controller can verify snapshot hygiene before submission and either warn or refuse, depending on the configuration.

3.4 IoC Extraction Methodology

3.4.1 Extraction Target and Unit

A sandbox detonation produces a broad and heterogeneous set of artifacts, including decompiled or dumped strings, network capture files, decrypted configuration blobs via family-specific parsers, proxied HTTP transcripts, behavioral logs, and signature-scanning results. On the corpus level, the quantity of data is simply too great to consider direct inspection. The extraction stage is therefore reductive: it extracts artifacts from a single sample into a compact record that lists the actionable network indicators revealed by the sample, along with sufficient provenance to justify each entry.

Rather than extracting all the indicator types exposed by the sandbox, we limit our methodology to a single class: *Telegram C&C endpoints*. Because the Bot API is free to use, encrypted, resistant to takedown efforts, and blends in with legitimate traffic to a trusted host, it has become a primary exfiltration channel for commodity information-stealers. All endpoints in this channel follow the same structure: each endpoint is identified by a *bot token* and, if applicable, a *chat identifier*. This structural consistency permits high-precision systematic extraction, which is used by the rest of the methodology.

Telegram Bot Token Telegram bot tokens have a fixed and recognizable format: a numeric bot ID of multiple digits, a colon, and a 35-byte secret composed of URL-safe characters². This structure is unique enough to be reliably recognized in arbitrary binary data with a low false-positive rate. Moreover, the token is a *secret key* that identifies the bot: having a valid token is essentially equivalent to controlling the attacker's bot. It is therefore the single most operationally significant artifact recoverable from a detonation, and the basis upon which all subsequent analysis is built.

Telegram Chat ID Rather than an independent indicator, the chat identifier that specifies the destination for exfiltrated data is treated as a *satellite* of the token. This intentional asymmetry arises because a chat ID alone is merely an integer with limited discriminatory power and a high false-positive rate. However, the chat ID becomes highly relevant when linked to the corresponding bot. From a threat intelligence perspective, the chat ID facilitates the collection of critical information about malware operators and infected individuals. Therefore, our methodology does not produce any chat identifier that cannot be associated with a token. The output is therefore a *(token, chat)* pair, where the chat component may be absent, but the token component is always present.

The design choice also determines how deduplication is performed. The methodology must define when two observations refer to the same indicator, as the same operator infrastructure is often seen multiple times (across detonation passes, static strings, dynamic traffic, and different dropped payloads). This identity is established by the token and, where applicable, the associated chat: observations sharing a token are aggregated into a single endpoint,

²Based on empirical evidence, as no official documentation states the formal composition of a Telegram Bot Token

and multiple sightings are treated as *evidence* for that endpoint rather than as separate results. So deduplication is defined by operator identity, not by the file or offset where a string is found.

3.4.2 Evidence Sources and Their Roles

Our methodology is built around an explicit notion of *evidence*. An observation is never regarded as a complete indicator, but only as a *contribution* to one, with a complete record of its provenance. Each contribution is tagged with the type of source it was taken from, the detonation pass and analysis stage in which it was found, the file and locator within that file, and a short, human-readable snippet of the surrounding context. The reconciled endpoint includes the full list of contributions.

This evidence-centered design has three purposes. First, it makes every claim *auditable*: any endpoint reported can be traced back to the exact artifact and location that produced it. Second, it makes confidence a *derived* property rather than an asserted one, since the strength of an indicator can be computed from the types of sources that vouch for it. Third, it allows the same indicator to accumulate confirmation from independent sources. A token that appears in multiple locations is more credible than one seen only once. The evidence structure is designed to reward that convergence.

As the reliability of detonation artifacts as indicators of a live (C&C) endpoint varies, we grouped them into three different epistemic levels.

Observed traffic and parsed configuration. Network activity is the cleanest source of activity: if the sample is observed making a request to the Telegram API host with a token and a chat identifier, the endpoint is automatically confirmed to be in use. Configuration data is equally obvious: tokens and chat IDs found in typed, named fields are the malware author's explicit declaration of the C&C channel, which is recovered after unpacking. The configuration data is handled by a family-specific parser.

Static and unpacked strings. The presence of a token and chat in the printable strings of the original sample, or in the strings of a payload dropped or decrypted by the sample, constitutes moderate evidence. This suggests that the operator's infrastructure is embedded in the artifact, but it is not used during execution. Chat IDs are associated with the respective token based on textual proximity, not on observed activity or structured data.

Behavioral reports and logs. The lowest tier contains tokens that are found only in behavioral report files or binary behavior logs. These artifacts are often noisy and may contain strings unrelated to actual C&C activity. Such sources can reveal a true endpoint not identified elsewhere, but any endpoint based on this evidence alone is reported with the lowest confidence.

Static recovery involves inspecting printable strings without running the sample. This approach often fails for the commodity Telegram-stealer corpus, as packing and obfuscation remove Telegram strings from the distributed binary (Section 2.8). In contrast, dynamic recovery investigates the sample during execution. The proxied HTTP transcript captures endpoints in use, the family-specific parser extracts typed configuration from the running process, and behavioral blobs document runtime output.

The static string-scanning technique is applied at two distinct points in the analysis. The strings of the distributed sample are examined before detonation, whereas the same routine is later applied to the strings of any payload dropped or decrypted by the sample, artifacts that exist only after execution. The technique is identical in both cases; only its input differs, as the second application depends on the dynamic pass to first expose the unpacked payload. In Section 2.8, we discuss how packing usually removes operator infrastructure from the distributed binary. For instance, some samples include built-in endpoints that do not exercise their network path during detonation. Scanning unpacked payloads can recover this information, which neither static inspection of the original file nor dynamic traffic capture alone would reveal. The pass and source type for every observation are retained in its evidence record, so the contribution of unpacking remains recoverable from the provenance, even though it is not reported as a separate indicator axis.

A separate class of signals that do not constitute an endpoint is also collected. They are used solely for diagnostic taxonomy and include: the presence of the Telegram API host in the network, as detected through DNS queries; the server name in a TLS handshake within the raw capture; or the activation of Telegram-related signature rules.

The signature signals referenced above originate from six YARA rules developed for this study. These rules were applied to the static strings of each unpacked payload. Instead, they are not applied to the distributed sample, where packing generally removes Telegram references from the static strings (Section 2.8); its pre-detonation scan uses the sandbox's own rule set. The rules are divided into one specific rule and five less stringent ones. The specific rule, `tg_bot_api_specific`, matches both the Bot API URL and the token format. Therefore, even if no token is recovered from a sample, a match still indicates the presence of the API endpoint in the artifact. This signature assigns such a sample to the API tier of the diagnostic ladder (Section 3.4.5). The other five rules target broader references: `tg_generic` matches bare `t.me` or `telegram.org` mentions and common Bot API method and field names; rule `tg_supergroup_id` matches negative channel identifiers; `tg_deep_link` and `tg_invite_link` match `tg://` and private-invite URLs; and `tg_user_api` matches Pyrogram or Telethon credential pairs. A match on any of these five, in the absence of a token or API evidence, places the sample on the weaker reference tier. These rules never really produce an endpoint because tokens are retrieved through extraction routines rather than signatures. Additionally, chats are identified solely through the proximity procedure, making the process more continuous than finite. The rules are intentionally designed to be non-evidential, since the weaker patterns may trigger on incidental Telegram references unrelated to operator channels. The complete text of the six rules is provided in Appendix I.1.

3.4.3 Candidate Extraction and Normalization

The consistent format of Bot Tokens makes them easy to extract in most analysis artifacts. However, tokens embedded in obvious schema or markup contexts are excluded, as these often introduce token-like noise rather than real credentials.

For Chat IDs, the extraction is more complicated: in unstructured sources such as static strings, payload strings, and behavioral blobs, typed fields are absent, thus preventing the extractor from distinguishing whether an integer represents a chat identifier and to which token it belongs. This is partially solved by using a token-anchored proximity approach. This reflects the layout of data in memory and configuration, where an operator's token and its destination chat are typically stored close together.

The token-anchored procedure locates every token in the blob and searches a bounded neighborhood around each token for potential chat-identifier candidates, assigning the closest candidate to that token. Neighborhood size can be adjusted for each chat identifier type based on its risk of false positives. Negative groups and supergroups have characteristic signatures and lengths, so a wider search window can be used with little risk of false positives. In contrast, bare positive integers, which represent private chats, occur more frequently and are therefore searched in a narrower window, immediately around the token to minimize accidental matches.

Furthermore, because a token's numeric bot identifier shares the digit-only form of a chat ID, no candidate that overlaps with a token's own bot identifier is accepted. This prevents the bot identifier from masquerading as its own chat. If no Chat ID candidate is identified near a token, the token is still saved as a token-only endpoint, preserving the most valuable IoC even when its destination is unknown.

3.4.4 Confidence Assignment

All contributions collected from all sources and all passes are combined and reconciled to produce a final set of endpoints. Reconciliation groups aggregate contributions by token. Accompanied by the token, distinct chats lead to distinct paired endpoints, while contributions lacking a chat get attached as supporting evidence to those pairs. If the token were never paired, it produces a single token-only endpoint. Operator infrastructure observed multiple times across various artifacts of the same sample is condensed into the smallest set of unique endpoints that explains all the evidence.

Confidence is assigned to each endpoint on the three-level scale. An endpoint obtains high confidence when at least one contribution comes from observed traffic or parsed configuration; medium confidence requires a complete token-and-chat pair obtained from static or payload strings; and everything else is classified as low confidence: a token-only endpoint or a pair found only in behavioral reports and logs. Confidence is therefore directly related to the origin or source of information, and the credibility of any indicator is determined by the reliability of the sources that produce it.

3.4.5 The Diagnostic Ladder: Making Absence Measurable

Every sample is placed on an ordered five-tier *diagnostic ladder* that records the observed level of Telegram involvement, even when no endpoint is recovered. From strongest to weakest, the tiers are:

1. a complete token-and-chat pair was recovered;
2. a token was recovered, but no chat could be bound to it;
3. no token was recovered, but the sample demonstrably interacted with, or referenced, the Telegram *API* specifically (through observed network presence, an API host substring, or an API-specific signature);
4. no token and no API interaction, but some weaker Telegram reference was present (a deep link, an invite link, or a generic signature);
5. no Telegram involvement of any kind was detected.

The binary extraction result is converted into a graded diagnostic result, with each sample assigned the highest level it meets. The ladder separates samples that have no Telegram-related signal at all (the lowest tier) from samples that have some Telegram-related signal but yield no recoverable endpoint (the middle tiers). What puts a token-free sample in the middle tier rather than the bottom is the non-evidential network and signature signals, which, by themselves, do not generate an indicator. The signals are indicative, not definitive. An observed contact to the API host, or a signature rule match, suggests that the sample is Telegram-related in some way, but neither proves that an operator channel was present and missed. The middle tiers show where recovery could, in principle, improve. The lowest tier records the apparent true negatives, subject to a separate status check that distinguishes them from artifacts of analysis.

3.4.6 Measurement Integrity Safeguards

The host-only design requires that each sample receive plausible answers to its network probes so that its execution proceeds far enough to reveal its configuration (Section 4.2.2). These responses are from a lab responder that generates a mixture of Telegram, geolocation, and file-upload responses and necessarily includes synthetic identities: a benign bot and a destination chat that stand in for the operator's own. Because these identities are structurally indistinguishable from real ones, without safeguards, they would be extracted and reported as operator infrastructure, contaminating the corpus and inflating downstream measurements that interpret recovered endpoints as evidence of operator activity. To prevent this, the responder's identities are removed from the recovered evidence prior to reconciliation. The number of such removals is recorded for each sample, ensuring that the filtering process remains auditable.

As described in Section 3.3.3, our analysis methodology involves two detonation passes of the same sample, treating both symmetrically. The evidence from either pass is valid, and each endpoint records the specific passes in which it appeared. The two passes provide complementary evidence, increasing recall because an endpoint missed in one regime may be found in the other.

Since the analyzed artifacts tend to be adversarial (whether from deliberately malformed software or from best-effort sandbox tooling), robustness to malformed inputs is prioritized. A corrupt capture or missing directory affects only the result for that particular source, and the failure is recorded in the corresponding sample summary. Individual failures do not stop the process. At the corpus level, this approach enables extraction as a restartable batch process. Partial failures are logged, and the pipeline continues to run without any interruptions.

3.5 Stealer Log Ingestion Methodology

The victim-side modality of this study draws on a corpus of commodity infostealer *stealer logs* obtained from Telegram: the per-victim archives that stealer operators publish on Telegram channels and underground markets. The binary corpus analyzed in the previous sections reveals the infrastructure an operator *declares*, while stealer logs reveal what the malware actually *collected* from infected machines. This section defines the methodology that transforms a heterogeneous collection of operator-formatted victim archives into three privacy-preserving signals: a ranking of installer filenames, a ranking of common visited hosts, and an ordered per-victim browsing trajectory. Section 3.6 then builds the delivery vector and *Point-of-Infection* (PoI) analysis on top of those signals.

3.5.1 Role, Data Source, and Ethical Posture

A stealer log is a per-victim bundle containing a system-information file written by the stealer along with the data it collected during the infection. By nature, the harvested material is stolen personal data, including saved passwords, browser cookies, autofill records, cryptocurrency wallets, payment card details, browser history, game credentials, and documents present on the victim’s PC. Such data contains sensitive information of the victim, which, for ethical reasons, is *never opened*. The stealer log ingestion reads only two classes of artifact: the system-information document and the browser history. Both still contain sensitive information, such as the victim’s IP address or the device specifications, but the browser history requires some careful handling. Therefore, browser history is reduced immediately and irreversibly to host and path, while full URLs, query strings, page titles, or visit timestamps are never stored.

This restriction is enforced not only for ethical reasons, but also for structural ones. Only paths whose shape is history-like are accepted (a file inside a directory named `history`, a root-level `history.txt`, or a Chromium History SQLite database under a `Raw` or `Default` parent). Files that store credentials do not match any of these requirements, preventing them from being read. An analogous exclusion is applied to the family-identifier, refusing

by basename any file that looks like a password, cookie, autofill, wallet, card, history, bookmark, download, or token dump.

3.5.2 Ingestion Design

Stealer logs are bundled into nested .7z archives, and the primary cost of ingestion is decompression, as each victim archive must be expanded before any internal files can be examined. Given that the corpus contains tens of thousands of archives, the methodology is structured as a single-pass approach, where each victim log is accessed exactly once to update three independent accumulators: the installer-filename ranking, the history-host ranking, and the per-victim trajectory. Stealer-family identification is also calculated during this pass.

The design includes selective execution, which allows for restricting a run to each function independently. However, traversing the archives is always required, as it constitutes the most expensive computation. Each victim is identified by a unique `log_id`, which corresponds to the SHA-256 digest of the inner archive bytes. This identifier is the join key that enables reconciliation of the install filename, family label, and trajectory for each victim without necessitating additional corpus reads.

3.5.3 Format Detection and Common-Schema Extraction

Commodity stealers do not share a single log format, but each family (and each different builder version of the same family) constructs its archive differently and assigns its system-information file differently (`information.txt`, `System.txt`, `system_info.txt`, a capitalized `Information.txt`, and others). To ensure that no credential store is ever opened, each log is assigned to a known format using only the archive manifest, the list of file paths and their sizes, never their contents. They are classified into one of a small number of formats, in order from most specific to least specific: logs originating from MacOS contain keychain artifacts, then two Windows variants, conventionally named as formats A and B, include different files and directories, and a fallback to an unknown class when no signature matches.

Once classified, a per-format parser extracts information from the system-information file to populate a *common schema*. From the system-information file, each parser retrieves only the fields required by the downstream install-side and attribution analyses and discards the rest. Each parser recovers the installer filename and the log date. Subsequently, the install path is decomposed into an anonymized directory pattern, in which both high-entropy random segments and the user-account segment are masked, and a boolean flag indicates whether the filename itself appears randomly templated. Three additional install-independent signals are collected, if present: the build version, the build identifier, and the operator brand. As per the ethical posture, no field identifying victims persists.

3.5.4 Install-Filename Normalization and Ranking

Installer filenames are the most direct victim-side evidence of the social-engineering pretext used to deliver a payload (e.g., cracked-software installer, game cheat, document lure, etc.). They are normalized conservatively so as to be countable across victims: they are reduced to their basename, stripped of surrounding whitespace, and lowercased. The extension is retained because it is discriminating, while download-collision suffixes such as a trailing (1) are folded so that re-downloads are aggregated into a single name.

Names are primarily ordered by their cross-victim count, with ties broken by the Shannon entropy of the filename stem, computed in ascending order. The idea is that a name seen by thousands of victims is more important than a rare one. Entropy helps only to sort names that have a common count, putting the meaningful low-entropy names (`setup.exe`, `crack.exe`) above the high-entropy random strings in a tie group. For each `log_id`, a per-victim side table is created containing the normalized installer name, so that the install-side signal can later be joined onto any per-victim output by identifier alone.

3.5.5 Stealer-Family Identification

Attributing each victim log to the originating stealer family allows subsequent analysis to determine whether a specific delivery vector is unique to a family or shared across families. Because each family generates a specific system-information file, family identification is obtained through signature-matching on this file. To do so, we used the open-source *what-is-this-stealer* rule set³ in which each rule corresponds to a single family by requiring the concurrent presence of approximately 6 family-specific ASCII strings.

This method is driven by two design decisions that enhance robustness and ethical integrity. Candidate files are prioritized in tiers: canonical information-file names are considered first, followed by files with information-like names, and, as a limited fallback, the smallest remaining plain-text files. This prioritization ensures that, under first-match-wins semantics, the canonical information file determines the outcome, while the fallback tier only applies when no recognizable information file is matched. Additionally, the exclusion of credentials and personal data is enforced at every tier. The procedure extracts only the family name and the matched file's path from each file, and it records both the primary family and the complete set of triggered rules.

To guarantee reproducibility, the rule set is tied to a specific upstream commit (2c30cd2, vendored on 2026-05-28) and maintained as a static snapshot. Two families found in the stealer log corpus, *SHubStealer* and *MacSyncStealer*, weren't included in the original set. To address this, we created one rule for each family, bringing the total number to 28. These additional rules were created after manual inspection of logs that were not matched by any existing vendored rule. Each of these families adds its own name and some fixed labels, such as a `Build Tag:` line, to its system-information file. As a result, the new rules encourage the simultaneous presence of these

³<https://github.com/MalBeacon/what-is-this-stealer>

specific strings and follow the one-rule-per-family approach of the vendored set. The two new rules are provided in Appendix I.2; the 26 vendored rules are not reprinted and are available unchanged from the pinned commit.

3.5.6 History-Host Aggregation

The host-side analysis addresses a population-level question: for how many distinct victims does a given web host appear in browsing history? To answer it, each URL from a victim's history is reduced to three signals:

- **URL-stripped:** the full host-and-path URL with query parameters and fragments removed, and path depth capped (e.g., `youtube.com/c/<channel_name>/videos`). This is the finest-grained signal and serves as the unit of counting.
- **host-path-token:** the host together with its first non-empty path segment (e.g., `youtube.com/c`), a coarser rollup that groups every page under the same site section.
- **bare host:** the host alone (e.g. `youtube.com`), the coarsest rollup, answering only whether the site was visited at all.

The host-path-token and bare host are kept as denormalized columns on each URL-stripped row, so the data can be aggregated at any of the three granularities without reparsing. Counts are per victim at the stripped URL level: if a victim revisits the same stripped URL multiple times, it is counted as a single increment. Having each unique stripped URL as its own row prevents a signal path (e.g., a given creator channel) from being averaged out by noise paths under the same host.

Two refinements enhance the signals. First, a static denylist flags common benign hosts, such as major search, social, streaming, productivity platforms, and a wildcard for Google's various country domains as *common*. This is implemented as a flag rather than a deletion, allowing analysis to include or exclude common hosts as needed. This denylist is deliberately broad, matching by host suffix and including any subdomain. Such breadth, however, inevitably overreaches; therefore, a small allowlist overrides the denylist for those cases where a typically benign host serves as a delivery vector.

The second refinement introduces a recency signal. A single "top" URL that represents the most recently visited URL is recorded for each victim. This is selected from the maximum-timestamp row in the Chromium history database when available, or otherwise from the first link in document order in the text history. This recency signal, together with the host-frequency ranking, was the first evidence that victims converge on shared endpoints

3.5.7 Host-Path Trajectories

The richest victim-side signal is temporal: the ordered sequence of hosts a victim visited in the browsing session that *culminated in stealer execution*, read as "the path leading to the payload". Stealer history files are stored most-recent-first, so they are reversed into oldest-to-newest order, so that the stored sequence reads in the direction of

the victim's journey. The unit of a trajectory is the *URL-stripped* token, identical to the finest-grained signal of the host ranking. This granularity preserves the *specific resource* a victim reached, such as a particular creator channel, a specific hosted file, or a named redirect endpoint. A controlled comparison against that coarser host-path-token empirically confirmed the trade-off: the finer token surfaces specific cross-campaign delivery resources that the coarser one cannot express, while the host-path-token yields cleaner and more compact frequent-pattern chains. Because recovering concrete indicators is the priority of this study, the finer granularity was adopted as the single trajectory representation. This trade-off is recognized as a known limitation and is left for future development. Full URLs that include query parameters and timestamps are still never stored, preserving the privacy posture.

Trajectories are built *per history file*, not per victim. A victim can have multiple browsers, each producing its own history. Within one history file, the visit order is recoverable, but there is no shared timeline across browsers, so merging a victim's histories into a single sequence would impose a cross-browser ordering that the data cannot support. The idea is that trajectories that do not contain infection indicators do not recur across the corpus and fall below the frequency thresholds, leaving only the recurring delivery chains as signals.

Each stream is then cleaned through four steps:

1. common tokens are dropped;
2. consecutive duplicates are collapsed;
3. the sequence is truncated to its most recent 75 tokens;
4. only sequences of at least three tokens survive, since shorter streams carry no signal.

Removing common tokens drops the high-traffic benign hosts that victims usually start a browsing session on, such as search engines and the major social and streaming platforms. Therefore, the reconstructed trajectory begins just downstream of the original referral, focusing on the lure and its immediate referrer rather than the initial entry point. This approach is intentional because the recurring signal across victims is found in the delivery hosts, and maintaining the standard entry points would cause common traffic to all browsing sessions to obscure the frequent-pattern output. The allowlist in Section 3.5.6 includes a few benign hosts that can act as delivery vectors. The result is a short, ordered, URL-stripped record of each browsing session.

3.6 Delivery-Vector and Point-of-Infection Methodology

The *PoI* is the answer to the questions of how and from where a victim acquired the stealer: the lure page, the cracked-software portal, the file host, or the redirect chain that terminated in the payload. This PoI can be inferred from the trajectories defined in Section 3.5.7, which record the host sequence leading to infection. This is an inference based on the timing assumption that the most recent history entries in a victim's browser reflect pre-infection browsing rather than activity that continued after compromise. This assumption is based on the execution

model of the family that influences the trajectory-bearing population. Most infostealers execute once and exfiltrate at run time, so the history it captures is a snapshot taken at the moment of infection, which usually follows some internet browsing. For a family that persists or re-exfiltrates in a second moment, this assumption would not hold.

This section defines inference as a sequence of complementary instruments: a transition graph and its community structure to recover the *ecosystems* of delivery infrastructure, sequential patterns to recover recurring *infection-vector chains*, and a direct last-hop instrument to rank the individual *lures*. The section concludes with validation experiments and an honest statement of the corpus's structural bias.

3.6.1 From Trajectories to Infection Vectors

Each stored trajectory is modeled as a walk on a directed graph, with nodes representing the URL-stripped tokens and edges representing the URL transitions made by the victim. By aggregating these walks over the entire corpus, we obtain a transition graph where the weight of each edge is the number of victim streams that pass through that edge. Because the trajectories are already minimized at ingest, no additional preprocessing, length trimming, or filtering is performed prior to counting; instead, noise control is implemented as a post-aggregation pruning.

3.6.2 Host-Transition Graph

The working graph includes only edges that are traversed by at least three distinct victim streams, assuming that transitions containing fewer than three victim streams are browsing noise rather than a shared route. The threshold considers streams rather than distinct victims, so a victim running several browsers can contribute jointly. We then apply a standard suite of structural measures to this pruned graph. We compute weakly connected components, sorted by size (the largest component gets component id 0). Weighted PageRank provides a global prominence score and is used to designate a representative host for each community, which is its highest-ranked node, for labeling and summary.

However, weakly connected components are ineffective as units of campaign analysis in this corpus: a small number of generic delivery platforms, such as public file hosts, cloud drives, and link shorteners, are accessed by victims of almost every campaign, and these shared endpoints serve as bridges that merge otherwise unrelated campaigns into a single large component encompassing the majority of hosts. As a result, it does not identify the specific campaign to which a host belongs. This limitation motivates the following methodological steps.

3.6.3 Ecosystem Grouping via Leiden Community Detection

To recover campaign-scale structure, hosts are grouped into *communities* using the Leiden algorithm [72] configured with the *Constant Potts Model* (CPM) objective. The Leiden refinement phase ensures that every returned community is internally connected, while the CPM objective imposes no resolution limit, so the long tail of

small campaigns is not forced into a few large communities. Community identifiers are relabeled by descending size for stable cross-run interpretation, mirroring the convention used for components. Communities are written as $C_0, C_1, \dots$, where C_0 is the largest community.

This step deliberately uses two different views of the graph. Community detection runs on an undirected version of the transition graph: each pair of hosts is joined by a single edge whose weight is the sum of the transitions in both directions, and an edge is kept only if the combined weight is at least 5. Both choices rest on the same assumption about campaign structure: that it is carried by transitions several victims share, and that the direction of any single transition is uninformative. Everything else (PageRank, sequential patterns, and the bridge analysis) continues to use the full directed graph.

A CPM resolution parameter γ acts as a granularity dial: a larger γ produces many smaller communities and shrinks the giant component, while a lower γ merges them into fewer, larger communities and grows the giant component. A range of γ values is evaluated, and the full per-host assignment is stored for each. A single value, $\gamma = 0.05$, is then chosen as the operating point for the downstream joins. This value is not claimed to be optimal, but is a good tradeoff granularity dial. The full scan is kept so that the sensitivity of the downstream conclusions to γ can be examined directly.

3.6.4 Shared Delivery Infrastructure

The bridge hosts that undermine component analysis matter in their own right: they are multi-tenant delivery infrastructure reused across several campaigns. These hosts are identified on the directed graph by counting, for each one, the number of distinct communities among the hosts that link into it. A host is classified as a bridge when it belongs to two or more communities, and as shared infrastructure when it spans three or more: a cross-campaign delivery sink. This rule identifies public-file hosts, cloud drives, and redirectors, and measures how often each is reused. The bridge analysis remains on the directed graph and uses the community labels only as input; the Leiden step changes only the grouping and leaves every other measure unchanged.

3.6.5 Sequential Infection-Vector Patterns

Communities show which hosts are linked, while sequential patterns display the order in which victims move between them. Closed sequential patterns are extracted from the raw trajectories with the PrefixSpan algorithm [27]. A pattern is kept only if it surpasses an absolute support threshold and includes at least three tokens. Both filters keep the output to a small set of recurring infection-vector chains, well short of everything PrefixSpan could enumerate. Limiting the analysis to closed patterns removes redundant prefixes; a shorter chain is reported only if it has strictly greater support than its extensions. Each resulting pattern is an ordered chain of URLs that many victims followed en route to infection.

3.6.6 Family-to-Vector Attribution

The mined patterns are associated with the family labels described in Section 3.5.5 to determine whether an infection-vector chain is unique to a single stealer family or shared among multiple families. The distribution of the originating family is calculated for each pattern based on the distinct victims whose trajectories include that pattern. A victim is counted as *following a pattern* if the pattern's tokens appear in their browsing in the same order, even if other elements fall in between. Each victim is counted only once, even if several of their browser histories match, and a victim that is not identified by any family falls in an explicit *unlabeled* group. Family counts are therefore expressed in unique-victim units, which cannot exceed the pattern's stream-level support.

3.6.7 Operator-Brand Correlation

The family label is a weak discriminator in this corpus, because the trajectory-bearing population is dominated by a single family (Section 3.6.10). This limitation is addressed by a second orthogonal attribution axis: the operator brand, identified by the Telegram handle or panel address associated with the stealer's build, subdivides the main family into different operator panels. The brand is obtained only for the logs whose system-information file or build metadata exposes it, which means the brand-based analysis is scoped only to that subset. Raw brand strings are normalized before counting.

Each brand is attached to its victims and then examined against both the stealer family and the community assignment. Since a single brand often covers multiple communities, it is considered an operator-identity axis, which is orthogonal to the delivery-vector axis represented by the communities. Together, the two axes produce operator-by-vector pairs that neither can recover on its own. The operator brand subdivides the dominant family into operator-specific panels, while the minority families remain separable on the family axis. Because the brand is extracted independently of the family signature, it can also label victims whose signature scans leave them unclassified.

3.6.8 The Last-Non-Sink Hop: Direct Delivery-Vector Recovery

Community detection and pattern mining are performed only on the count-thresholded graph, which limits them to hosts shared by sufficient victims. They are therefore missing the long tail of one-off lures. In a separate step, these are recovered by taking, for each victim, the last token in the browsing trajectory and the host nearest to the moment of infection, and skipping backward over any host classified as generic delivery infrastructure until it reaches a meaningful one. This host is the victim's inferred delivery vector, also known as a lure.

The exclusion set consists of a curated, γ -independent denylist targeting multi-tenant public delivery infrastructure. It includes generic file hosts, cloud drives, public link shorteners, open redirectors, and third-party advertising-redirect networks. This set is distinct from the shared-infrastructure flag of the graph. That flag depends on the selected gamma and includes two types of hosts: generic sinks used by all, and theme-specific

lure hosts that host their own downloads. Reusing it would delete exactly the delivery hosts this step is trying to recover. The exclusion isn't applied universally. Instead, multi-tenant hosts are retained because they are often identified not just by being shared hosts, but by specific per-tenant paths on them, like storage buckets, link-aggregator pages, or redirector slugs. These paths serve as the key indicators. The last identified non-sink hosts are then grouped together from all victims, which creates a ranked list of common, recognizable lures, along with counts for the more unique, less common ones in the long tail. This process helps to distinguish hosts that belong to a larger community from those niche lures that community-based methods might miss.

3.6.9 Validating Communities as Delivery Vectors

Three experiments evaluate whether the recovered structure functions as a delivery vector rather than representing a clustering artifact. Since the corpus does not include per-victim ground truth for the delivery vector, recovery must be inferred. In contrast to studies that directly observe the download process, these experiments support an interpretation rather than measuring against a known answer.

Each victim is assigned to a community using three methods: plurality across hosts, terminal host, and last non-sink hop. Each assignment is then linked to the installer filename, stealer family, and operator brand, based on the expectation that a genuine vector will concentrate victims under a dominant label. These three assignments are not independent, as they utilize the same community map and trajectories, differing only in the voting token. Therefore, their agreement serves as a consistency check rather than triangulation. A reverse join examines which communities deliver each installer filename, revealing builds distributed across multiple vectors.

The only check that does not utilize the community map is the last-non-sink lure ranking described in Section 3.6.8, which identifies delivery hosts directly from each victim's trajectory.

3.6.10 Structural Bias and Coverage Limits

The delivery-vector analysis depends on browsing trajectories, so only stealers that exfiltrate browser history can be part of it. This introduces a structural bias because, in this corpus, families that do not exfiltrate browser history contribute no trajectories. Although the family-identification step labels victims across several families, it's important to note that the subset with trajectories is primarily influenced by a single family that exfiltrates history. Cross-family questions, such as whether different families share delivery vectors, can be answered only as far as the minority families are represented. The operator-brand axis, explained in Section 3.6.7, helps diversify this monoculture by separating the main family into distinct operator panels and adding a discriminating variable that the family label alone doesn't capture. A second, smaller limitation comes from the last-non-sink instrument discussed in Section 3.6.8: a sizeable share of victims end on a host outside the count-thresholded projection. These are off-projection niche lures that the graph cannot capture and that only the direct last-hop ranking can recover.

A further limitation is the counting of victims. Victims are keyed by the SHA-256 of their inner archive, so

deduplication is byte-exact: a log re-shared and recompressed in transit, common in the Telegram log economy, could get a different digest and be counted as a different victim, inflating victim totals.

Another limitation is that several instruments rely on curated denylists instead of learned criteria. The common-token filter, the operator-brand allowlist, and the last-non-sink strip set are manually created, making the recovered lures, communities, and patterns dependent on these lists. A host missing from the strip set may be mistaken for a victim's lure, while it is actually just generic delivery plumbing. Consequently, the extended list of individual lures might include unlisted sink artifacts.

4 System Design and Implementation

4.1 Manual Analysis Lab

The manual environment runs on a Type-2 hypervisor over a strictly host-only network and consists of two guests, REMnux and Flare VM.

4.1.1 Victim and Observer

The observer is a REMnux guest. It contains multiple tools for reverse engineering and static analysis. It is used to obtain basic information about the analyzed sample, such as the programming language used for development, whether it is packed, and which packer is used. It is also used to dissect and unpack the sample to analyze its logic. Finally, it acts as the victim's default gateway and DNS server during dynamic analysis.

The victim is a Windows 10 guest provisioned with the Flare VM toolkit distribution for in-memory, registry, and binary inspection. It is also used for malware detonation; therefore, common hardware metadata values, such as BIOS and chassis identifiers, are overridden to suppress the most obvious hypervisor artifacts.

4.1.2 Network Simulation and Traffic Capture

Simulated internet services are provided by INetSim. Since INetSim does not handle TLS, PolarProxy is placed in front of it to terminate HTTPS, with `iptables` redirect rules pivoting the corresponding ports to its listener. PolarProxy issues leaf certificates on demand from a custom Root Certificate Authority installed in the victim's trust store. The Root CA is issued without CRL Distribution Point or Authority Information Access extensions, and revocation checking is disabled on the guest, so that the offline chain does not trigger validation lookups that would fail and reveal the simulated environment.

A thin `nginx` layer sits ahead of INetSim to satisfy some connectivity-awareness checks that were identified during the static analysis of the samples.

Network evidence is assembled from two sources: non-intercepted traffic captured with `tcpdump` on the observer, and decrypted flows logged by PolarProxy. The two are merged into a single file, avoiding double-counting TLS sessions that were already recorded by PolarProxy in cleartext.

4.1.3 Malware Analysis Hygiene

Both REMnux and Flare VM guests are captured in a clean snapshot after provisioning is complete and prior to the analysis and execution of any sample. Each analysis session begins from these snapshots, and the guests are

reverted to this state before processing subsequent samples. This approach ensures that no analysis residue or artifacts are carried over to subsequent analyses.

4.2 CAPE Sandbox Environment

The automated pipeline is built on CAPEv2 using KVM/libvirt as a virtualization framework. The analysis guest is a Windows 10 machine hardened against sandbox detection by removing Windows Defender, debloating the system, and adding environmental realism via mainstream applications and synthetic user artifacts. The resulting profile is nearly indistinguishable from an authentic user machine. The guest sits on an isolated, host-only bridge and all traffic interception occurs on the host.

4.2.1 Network Routing and Isolation

Network isolation and containment are enforced by a static `iptables` configuration on the `PREROUTING` and `FORWARD` chains. Port 53 traffic, over both UDP and TCP, is redirected via DNAT to a local DNS responder on KVM's bridge address, capturing also connections that try to bypass DNS resolution by hardcoding a specific resolver. All other TCP traffic is redirected to the interception proxy, regardless of destination port. The remaining non-DNS UDP traffic is dropped on the `FORWARD` chain so that no traffic of that class can leave the host.

4.2.2 Synthetic Responders

Two custom services provide simulated Internet connectivity: a DNS responder and an HTTP responder.

Selective DNS responder. The DNS responder (`dns_responder.py`) is a system service that maintains a curated allowlist of domains that are relevant to commodity infostealers, including Telegram endpoints, connectivity checks, geolocation services, and a handful of benign domains that a real Windows host resolves at startup. Any query matching the allowlist will resolve to the bridge address, and any other query, including the random names used as anti-sandbox probes, will return `NXDOMAIN`. This selective behavior is a more faithful reproduction of a real resolver than the `INetSim` DNS resolver, which arbitrarily resolves each domain with a positive answer, a behavior that a malware sample recognizes as simulated.

HTTP and TLS responder. HTTP and HTTPS response synthesis is driven by a patched CAPE auxiliary module (`Mitmdump.py`) that runs `mitmdump` in transparent mode with a Python addon (`responder.py`). The proxy only receives a leaf certificate for the requested name and never establishes an upstream connection, so it never contacts the real endpoint. The responder addon handles each request based on host and path. Plausible synthetic payloads are sent to connectivity and IP geolocation requests, while requests to known exfiltration endpoints (the

Telegram Bot API host) are intercepted, and a success status code is returned. A decrypted HAR transcript is generated for each task and is the primary source used by the extractor (Section 4.5.3).

4.3 Sample Corpus Construction

4.3.1 Architecture and Execution Model

The pipeline operates in seven sequential steps, each implemented as a simple shell wrapper around a Python module that interacts with the shared database. Table 4.4 lists these steps and their respective responsibilities. Each step is idempotent: re-running a step processes only the rows that remain pending, allowing interrupted runs to resume without repeating completed work. Before a full run, two auxiliary modes support validation: a *dry-run* tests each strategy at a reduced limit to confirm that every strategy resolves and returns results without writing to the database, and a *smoke_test* executes the entire pipeline on a single known-good SHA-256.

Table 4.4 Corpus construction pipeline steps.

Step	Module	Responsibility
0	<code>db.py</code>	Initialize a fresh database.
1	<code>query.py</code>	Query Malware Bazaar through the 33 strategies.
2	<code>filter.py</code>	Classify and filter out-of-scope file types.
3	<code>download.py</code>	Download samples.
4	<code>sanity_check.py</code>	Extract zipped files and apply the sanity checks.
5	<code>stats.py</code>	Print read-only corpus characterization statistics.
6	<code>export.py</code>	Export the decision tables as the reproducible corpus record.

4.3.2 Data Model and State Tracking

All persistent state lives in a single SQLite database at `data/working.db`. Beyond the database, only two modules write to the filesystem: `download.py` writes password-protected ZIP archives to `data/samples/`, and `sanity_check.py` extracts archives to `data/extracted/<sha256>/raw/` and relocates rejected material to `data/rejected/<sha256>/`. Every module imports from `pipeline/config.py`, which serves as the single source of truth for all configurable parameters: directory paths, API credentials, rate limits, the family allowlist, and the complete list of query strategies. No other module owns configuration state; changes to scope, thresholds, or rate limits propagate automatically to every component simply by editing `config.py`.

Each database connection is opened in write-ahead-logging mode, with foreign key rules enforced and a 32 MB page cache configured to balance multiple read operations with data safety. The main `samples` table contains

one row for each unique SHA-256 value. Rows from this table are never deleted during normal operation, and the progress of each sample is tracked by updating four independent status columns. As a result, the lifecycle is the product of four distinct state machines:

```
filter_status:    pending    -> kept | needs_unwrap | rejected
download_status:  pending    -> downloaded | failed    | skipped
candidate_status: candidate -> downloaded -> admitted | rejected
admitted:         NULL       -> 1 (admitted) | 0 (rejected)
```

A candidate is added in Step 1, classified in Step 2, downloaded in Step 3 when it passes filtering, and finally admitted or rejected in Step 4. Rejections are not ignored: a rejected sample is recorded in a dedicated `rejected_samples` audit table together with its reason code, and its extracted files are moved to a rejection directory. Two append-only logs track every query and download attempt, along with their API or HTTP status. The database additionally contains tables reserved for later phases, such as IoC records, so that cross-phase queries require no data movement.

4.3.3 Query Orchestration and MalwareBazaar Client

The query orchestrator issues the 33 strategies defined in Section 3.2.2 (16 tag queries, 16 signatures queries, and the cross-tag `tag:telegram` query) and consolidates the results.

Each raw result returned by the API is passed through a normalization function that maps each MalwareBazaar field name to the respective database column. An `infer_family` function attributes a family to the sample. When present, the MalwareBazaar signature takes priority; otherwise, the tags or the strategy identifier are used as a fallback.

Multiple query strategies can produce the same SHA-256 hash, and typically, a single sample has a signature attribution and multiple community tags. Therefore, candidates are inserted via an upsert that merges the new strategy identifier into the `discovered_via` JSON array by checking whether the SHA-256 hash already exists before deciding whether to insert or update. A full discovery trail is kept for each sample, without inflating the candidate count.

All API access is routed through a single client that enforces the abuse.ch rate limit. The client throttles to 1 request per second. If failed, requests are retried up to 3 times with a fixed delay, except for HTTP 401 responses, which are treated as permanent authentication errors and returned immediately.

4.3.4 File-Type Filtering

File-type filtering (Step 2) classifies each candidate into one of four outcomes based on the scope described in Section 3.2.3: in-scope formats are `kept`, archives are flagged `needs_unwrap` and passed through, and out-of-

scope formats are rejected with a specific reason code associated with the respective file type (shown in Table 4.5) and an entry in the audit table. Unrecognized files are also treated as rejected.

Table 4.5 File type rejection codes.

rejection_reason	Triggered by file_type
jvm_not_in_scope	jar
android_not_in_scope	apk
macos_not_in_scope	macho, dmg
linux_not_in_scope	sh, elf
browser_extension_not_in_scope	crx
document_low_value	pdf
compiled_help_not_in_scope	chm
web_page_not_in_scope	shtml, html
unknown_file_type	unknown, xw
file_type_not_recognized	Any unrecognized file type not in the map

4.3.5 Download Module

Samples whose filter status is kept or needs_unwrap are selected as pending in the download step (Step 3). The main download loop processes candidates in descending order of their MalwareBazaar first-seen timestamp, placing more recent samples earlier in the queue to provide useful partial results even if the execution is interrupted. This step is resumable and each attempt is logged.

4.3.6 Admission and Sanity Checking

Admission (Step 4) is composed of two stages. The password-protected archive is first extracted using the standard infected passphrase. A sample whose archive cannot be extracted is recorded as a rejection with reason `extraction_failed`. Following the extraction, three file-type-agnostic sanity checks are applied to the binary samples: a minimum size of 512 bytes (`too_small`), a maximum size of 100 MB (`too_large`), and binary readability (`unreadable`). A sample that passes all three checks is admitted, while a sample that fails one or more checks is rejected with the appropriate reason code, and its extracted files are moved to the rejection directory.

4.3.7 Reporting and Reproducibility

Two read-only modules complete the pipeline: a reporting module that characterizes the corpus at any intermediate stage of a run, and an export module that outputs the final decision tables for downstream use.

Reporting. The reporting module prints a series of seven rich corpus-characterization views that are safe to invoke at any point in the pipeline run as a progress check. This module includes, in order, seven tables:

- **Candidate summary.** Overall candidate count across the four status dimensions.
- **Filter stage.** Filter stage results including rejection reason codes.
- **Corpus by family.** A per-family corpus summary with total and admitted counts.
- **Download progress.** A download progress view showing byte counts and an estimated remaining time derived from the average per-sample download duration.
- **Admission summary.** Per-family admission rates and the most frequent sanity-filter rejection reasons.
- **Sample recency.** Number of admitted samples by year-month calculated using the MalwareBazaar first-seen tag.
- **Recent query log.** Query log showing the last 30 strategy runs with their result counts and API response statuses.

Export. The export module saves the two decision tables, `samples` and `rejected_samples`, as CSV or JSONL files, along with a manifest that records the export timestamp and the number of rows in each table.

4.4 CAPE Orchestrator Module

4.4.1 Modular Architecture and Data Flow

The orchestrator is implemented as a Python package, divided into submodules that manage each specific pipeline step. A configuration module loads a YAML file for orchestration settings, and persistence is managed by a single SQLite database that records the state of every sample and job. A `Sample` refers to an admitted binary, while a `Job` represents a single detonation of that sample during one pass, including its sandbox task identifier, status, and archive location.

A thin REST client is the only point of contact with the CAPE Sandbox API. Each stage is coordinated by a central run loop, and a command-line interface exposes each stage as a subcommand, so that individual steps can be driven in isolation, in addition to the full run. The external dependencies are limited to an HTTP client, a YAML parser, and the optional static-analysis tools. Table 4.6 summarizes the submodules and their respective responsibilities.

Table 4.6 Submodules of the orchestrator and their responsibilities.

Submodules	Responsibility
<code>config.py</code>	Configuration loader from YAML file
<code>db.py</code>	Database persistence
<code>cape_client.py</code>	REST client for the CAPEv2 API with bounded retry
<code>manifest.py</code>	Sample-tree scan and SHA-256 verification
<code>submitter.py</code>	Working-copy staging, snapshot check, submission
<code>snapshot.py</code>	libvirt snapshot verification
<code>poller.py</code>	Polling a task to a terminal status
<code>archiver.py</code>	Moves analyses into the archive
<code>orchestrator.py</code>	Main run loop
<code>static_analyzer.py</code>	Per-sample static stage
<code>payload_analyzer.py</code>	Per-payload static stage
<code>tools/</code>	External tool wrappers (metadata, strings, yara, capa)
<code>logging_setup.py</code>	JSON logging
<code>cli.py</code>	Command-line interface

4.4.2 Configuration Model

A single YAML file contains all configurable behavior. This file is parsed and validated at start-up by the `config.py` submodule. The goal is to reject misconfiguration immediately, rather than produce errors during execution. Validation includes checking that every key is present, that values have the expected shape, and that the declared detonation passes are internally consistent. If stages that require external tools are enabled, those tools and their rule directories must actually exist on disk. Some additional checks are performed, such as whether the sandbox is reachable and whether the sample directory is non-empty. The two detonation passes are also defined as configurations, each with its specified timeout and respective monitoring tools enabled.

4.4.3 State Model and Persistence

The `db.py` file handles a lightweight SQLite database with three tables: `samples`, `jobs`, and `orchestrator_runs`. The `samples` lists all admitted binaries, keyed by SHA-256. The `jobs` table represents every single attempt of detonation for each sample under a pass, and is identified by the triple of sample hash, pass name, and attempt number. It also includes the task identifier issued by the sandbox and the job status. The table is append-only, and retries are implemented as new attempt rows; therefore, the table contains the full history of every submission for auditing. Finally, the `orchestrator_runs` table summarizes the orchestration: each row records when a controller run began and ended, how many samples were in scope, and how many jobs

it carried to completion or permanent failure.

A job begins as *pending* and becomes *submitted* when it is posted to the sandbox. Once the sandbox reports execution is underway, it becomes *running*. From the running status, it can resolve to one of several outcomes, depending on whether it succeeds or fails. A clean, successful run ends in *completed*, while failure state depends on the time of reporting: *failed_analysis* when the analysis itself fails; *failed_processing* during post-analysis processing failures; and *failed_reporting* if failed during the reporting stage. Additionally, a *timeout* indicates a job that exceeded its deadline, while *infra_error* indicates a failure of the controller or sandbox infrastructure rather than the analysis itself. Finally, *failed_pending_retry* means that a job has failed but remains within its retry budget, and *failed_permanent* implies one whose retries are exhausted.

4.4.4 Sandbox Client

The `cape_client.py` module provides an interface to CAPEv2 through its REST API. It handles tasks such as sample submission, task viewing, task deletion, and error handling. Connection errors and timeouts are retried using exponential backoff within a defined wall-time budget, but if this budget is exceeded, the sandbox is considered unreachable.

4.4.5 Job Execution and Main Loop

The `orchestrator.py` module contains the primary run loop. If any, it first reconciles any in-flight jobs remaining from a previous execution; then it selects the next runnable jobs and processes them through submission, polling, and archiving stages. The sample is submitted to CAPE as is, without any modifications or unpacking if it is an archive. Following the sample submission, the `poller.py` submodule queries the task returned by CAPE at fixed intervals, recording status changes, until the sandbox reports a terminal status or a hard deadline, such as the timeout. The archiver executes the minimum evidence check described in Section 3.3.4 and moves the analysis directory into the archive directory.

4.4.6 Static and Payload Analysis Stages

The static stage (`static_analyzer.py`) runs once for each sample before CAPE submission. The payload stage (`payload_analyzer.py`) goes through the archived analysis folders and checks every dropped file within a set size range. Both stages use external tool wrappers found in `tools/`. These include string extraction with the system `strings` binary, signature scanning with `yara-python`, and capability detection using the `capa` tool, which runs as a subprocess and only works with PE and DLL files. The process also collects file metadata, such as hashes and entropy analysis results. Both stages are idempotent, so if you run them again, they only redo the steps that did not finish successfully before.

4.4.7 Command-Line Interface

The command-line interface (`cli.py`) breaks the workflow into subcommands: `initialize` and `scan` to prepare the database; `run` to execute the queue; `status` and `resume` to inspect and reconcile; `cancel-task` to terminate an active task; and two analysis commands to run a stage on demand. Individual sample faults are isolated so that a failure in one sample does not affect the rest of the batch.

4.5 IoC Extraction Module

4.5.1 Module Architecture and Data Flow

The extractor is implemented as a Python package structured according to the data flow. A small set of core modules determines the data model, per-sample execution coordination, and final outcome. The `sources` sub-package follows this pattern: each module is a stand-alone parser for a single artifact type and exposes a single entry point. The two persistence concerns, namely the per-sample summary and the corpus database, are managed by the run driver and a dedicated database module, respectively. Table 4.7 presents the modules and their respective responsibilities.

The package has a minimal external footprint. The only third-party dependency is a packet dissection library, which helps parse packet capture files. This keeps the component portable and easy to deploy along with the rest of the pipeline.

Table 4.7 Modules of the extractor and their responsibilities.

Module	Responsibility
<code>models.py</code>	Dataclasses, enumerations, and the summary serialization
<code>patterns.py</code>	Regular expressions, chat-id shape rules, proximity pairing
<code>allowlists.py</code>	Synthetic laboratory identities and their predicates
<code>classify.py</code>	Assignment of the five-rung diagnostic label
<code>reconcile.py</code>	Evidence pooling, de-duplication, confidence scoring
<code>extractor.py</code>	Per-sample orchestration and summary writing
<code>db.py</code>	Idempotent persistence into the corpus database
<code>cli.py</code>	Command-line interface and batch driving
<code>sources/</code>	One parser per artifact kind (see Section 4.5.3)

Inter-module communication is handled by a set of dataclasses defined in `models.py`. Each module imports these types and processes instances generated by the preceding stage to produce instances for the subsequent stage.

Evidence. The `Evidence` dataclass, an immutable value that describes a single observation contributing to an endpoint, represents the atomic record. This record includes the type of source, the stage and pass during which it was observed, the attempt identifier, the pass directory, any optional token and chat produced, and the locating fields: file path, in-file locator, and a short snippet. It also has an open extra dictionary for source-specific information, such as the chat subtype or API method. Because an observation is a historical fact that may be freely shared among aggregating structures after it is recorded, the dataclass is immutable.

Endpoint. The `TelegramEndpoint` dataclass, which includes a token, an optional chat, an integer confidence value, a list of passes in which it was seen, and a list of `Evidence` supporting it, represents the reconciled indicator. This dataclass provides derived properties such as determining if it constitutes a complete pair, and producing both normalized and raw string representations for database storage.

Signals and features. Two additional data structures store the non-evidential material. `TelegramSignals` records Telegram-related signature hits, differentiating the API-specific rule from less robust ones and enabling in-place merging to combine signals from multiple files. `PassFeatures` and `StaticFeatures` contain per-pass and static counters, such as HTTP request counts, network-presence flags, and lists of attempt directories, which inform both the classifier and the summary.

Run outcome. All outputs produced for a single sample, including static and per-pass features, reconciled endpoints, collected signals, substring flags, assigned diagnostic labels, suppression counts, and any errors, are combined into the top-level `RunOutcome` object. To prevent duplicate storage, this object computes summary statistics, such as paired endpoints, high-confidence endpoints, unique token counts, and multiple "Telegram-seen" predicates, as derived attributes.

Additionally, three enumerations mandate the controlled vocabularies: the analysis `Stage` (static, and the two passes), the `SourceKind`, which labels each artifact type, and the `FailureLabel` that encodes the five tiers of the diagnostic ladder in descending order of evidence strength, represented in Table 4.8.

4.5.2 Token & Chat-ID Pattern Matching

The `patterns.py` model includes all the patterns for the Telegram artifacts.

Token and chat shapes. This pattern matches a numeric bot ID, a colon, and a 35-character secret composed of URL-safe characters, as shown in Listing 4.1. Two boundaries truncate the match to keep it tight. To prevent the bot ID from being picked out of a longer number, one ensures that no digit precedes the ID. The other ensures that the secret is not merely the beginning of a larger string by requiring that no URL-safe character appear immediately

Table 4.8 Correspondence between the `FailureLabel` enumeration in `models.py` and the rungs of the diagnostic ladder defined in Section 3.4.5, in descending order of Telegram-evidence strength.

Label	Diagnostic-ladder rung
PAIR_OK	A complete token-and-chat pair was recovered.
TOKEN_ONLY	A token was recovered, but no chat could be bound to it.
API_TELEGRAM	No token was recovered, but the sample interacted with or referenced the Telegram API specifically (observed network presence, an API-host substring, or an API-specific signature).
TELEGRAM_OTHER	No token and no API interaction, but a weaker Telegram reference was present (a deep link, an invite link, or a generic signature).
NO_TELEGRAM	No Telegram involvement of any kind was detected.

after it. The same pattern may operate directly on raw binary data because it is also compiled as bytes. Chat IDs are matched by three byte-level patterns, one for each type: supergroups (a distinctive negative prefix), basic groups (a shorter negative number), and private chats (a plain positive number). Each candidate is then sorted by a helper function into the correct type based on its length and sign.

Each regular expression was validated interactively using regex101.com⁴ with representative input data prior to integration into the codebase. Furthermore, the bot token regular expression was validated using byte-level unit tests with fixtures derived from recovered samples, which exercise the look-behind and look-ahead boundaries against arbitrary surrounding bytes that a browser-based tester cannot represent.

Token-anchored search. The core of the module is the function that realizes the token-anchored proximity methodology over an arbitrary byte blob. It proceeds as described:

1. locates every token in the blob, discarding any that sit in schema or markup context (detected by a small set of XML/XSD keywords in a window around the match);
2. for every token, it searches within a narrow window for private chats candidates (positive integers) and a wider window for groups and supergroups candidates (negative integers);
3. rejects any private candidate whose byte span overlaps a token, or whose value equals a token's numeric bot identifier, so that a token's own digits cannot masquerade as a private chat;

⁴<https://regex101.com/>

Listing 4.1: Bot token and chat-identifier patterns.

```
# Bot token
TELEGRAM_BOT_TOKEN = re.compile(
    r"(?<![0-9])([0-9]{8,16}):([A-Za-z0-9_-]{35})(?![A-Za-z0-9_-])"
)
TELEGRAM_BOT_TOKEN_B = re.compile(
    rb"(?<![0-9])([0-9]{8,16}):([A-Za-z0-9_-]{35})(?![A-Za-z0-9_-])"
)

# Chat identifiers
TELEGRAM_CHAT_ID_SUPERGROUP_B = re.compile(
    rb"(?<![0-9])(-100[0-9]{10,13})(?![0-9])"
)
TELEGRAM_CHAT_ID_BASIC_GROUP_B = re.compile(
    rb"(?<![0-9])(-[1-9][0-9]{8,9})(?![0-9])"
)

TELEGRAM_CHAT_ID_PRIVATE_B = re.compile(
    rb"(?<![0-9])([0-9]{9,10})(?![0-9])"
)
```

4. computes the textual gap between the token and each chat candidate, sorts them by gap, and assigns the nearest distinct candidates;
5. if no chats were selected as candidates, the token is still saved as token-only result.

Each result returned by this function is a dictionary containing the token, the chosen chat ID and its identified subtype, the byte offsets for both the token and the chat ID, and a context window snippet. The caller of this module turns these results into `Evidence`. Token and chat ID offsets make these indicators traceable back to their exact position of the analysis artifact.

4.5.3 Per-Source Extractors

Each module within `sources` is responsible for parsing a specific artifact type and returning `Evidence`, along with feature counters where applicable. A shared helper module supplies the evidence constructor, a path-relativization utility, a snippet truncator, and an ISO-timestamp function, ensuring that all parsers generate consistently structured, archive-relative records. Three parsers—static strings, payload strings, and behavioral

blobs—utilize a common scanning routine based on the token-anchored search described in Section 4.5.2. These parsers differ only in their source type, locator prefix, and whether they perform taxonomy substring scans.

HAR (proxied HTTP), strongest tier. The HAR parser serves as the primary dynamic source. It processes the proxy’s HAR transcript by iterating through request entries. For each request directed to the Telegram API host, the parser extracts the token from the request path and retrieves chat identifiers from the query string, the structured query array, and the post body. The body handler accommodates all three formats observed in samples: URL-encoded forms, JSON bodies (recursively traversed for chat-id fields), and multipart bodies (segmented by the declared boundary). If a request contains a token but no chat, the parser generates token-only evidence that includes the API method. If one or more chats are present, it produces paired evidence for each chat. Additionally, the parser reports the total number of HTTP requests and the number of Telegram-host requests.

CAPE configuration, strongest tier. The configuration parser processes each JSON file generated by the sandbox’s family-specific configuration extractors. It flattens each document into leaf (`path`, `value`) pairs, selects those whose field names indicate a token or a chat and whose values conform to the expected format, and then pairs them accordingly. A single token is paired with each distinct chat in the document, or emitted alone if no chat is present. When multiple tokens are present, they are paired positionally with chats in document order, avoiding the creation of artificial combinations not present in the original configuration. Because the fields are typed and named, this source requires no proximity heuristics.

Static and payload strings, medium tier. The static-strings parser examines the printable strings within the sample itself, while the payload-strings parser applies the same analysis to each dropped or decrypted payload’s strings. Each piece of evidence is attributed to its originating payload by referencing the payload metadata map, with the payload hash and source path recorded in the locator and extra fields. Both parsers report the presence of the API host substring and the weaker Telegram substrings, and these findings contribute to the taxonomy. As both utilize the shared scanning routine, their pairing behavior is identical; only their provenance differs.

Behavioral blobs, weakest tier. Both the binary behavior logs and the full behavioral report are subjected to the token-anchored scan by the behavioral-blob parser. Because these artifacts include analysis-engine text that may otherwise produce misleading Telegram-reference signals, the taxonomy substring scan is intentionally muted for them. Only genuine token matches survive, and they enter at the lowest confidence.

Packet capture, taxonomy only. There is no evidence produced by the pcap parser. The parser checks payloads for either a DNS query to the Telegram API host or a TLS ClientHello with a server-name extension referencing that host after unpacking the link, network, and transport layers if the optional packet-dissection library is present. The parser gives an empty result if the library is not present.

Signature signals, taxonomy only. The YARA-signal reader processes the orchestrator's signature-match JSON for both the static scan and each payload, aggregating Telegram-related rule matches into the `TelegramSignals` structure. The API-specific rule activates the API flag, while other recognized rules set the weaker flag. All matched rule names are preserved for documentation.

4.5.4 Per-Sample Orchestration

The `extractor.py` module is the core of the orchestration. It locates the sample's archive directory and processes it in a predetermined sequence: static artifacts first, followed by two CAPE passes, signature signals, reconciliation, classification, and output.

Passes are identified by directory-name convention: a base pass directory and any suffixed attempt directories sharing the pass prefix. This approach ensures that multiple detonation attempts within a pass are processed, and their attempt identifiers are recorded. During each pass, the orchestrator calls the configuration, HAR, payload, behavioral, and pcap parsers; accumulates their evidence, and integrates their feature counters into the pass's `PassFeatures`.

An essential part of the parser's analysis architecture is fault isolation. Any parser exception that arises during a pass is logged, added to the run's error list, and otherwise silenced to allow the following sources to keep contributing. No single defective artifact should end a sample's analysis, which is operationalized by this procedure. Once all the evidence has been collected, the classifier labels the resulting `RunOutcome`, and a human-readable run log and a summary JSON are saved to the sample's output directory. The run is then saved to the database.

4.5.5 Reconciliation and Classification

Reconciliation. The `reconcile.py` module converts the flat evidence list to final endpoints. It scans the evidence once, discarding contributions containing tokens or chats matching synthetic lab identities. It counts each suppression, and indexes the remaining items by token and by token-chat pair. For each token, it generates one endpoint per distinct chat, attaching both pair-specific and token-only evidence, or a single token-only endpoint if the token was never paired. The endpoints are returned in a stable sort order, together with the suppression count. Confidence is computed here from the source kinds present in an endpoint's evidence: the strongest tier yields high confidence, a complete pair from the medium tier yields medium confidence, and everything else yields the lowest.

Synthetic-identity and response suppression. The laboratory responder synthesizes Telegram, geolocation, and file-upload responses to ensure that samples probing for connectivity continue to execute (Section 4.2.2). Reporting their associated identities as recovered infrastructure would contaminate the corpus with laboratory-specific values, since they are analysis artifacts instead of the operator's artifacts. To prevent this contamination, two mechanisms are implemented: first, the synthetic identities emitted by the responder are

declared in `allowlists.py` along with the predicates that recognize them. The `reconcile.py` module discards any contribution whose chat ID or token matches a synthetic identity. Each suppression is also recorded in the `suppressed_synthetic` field of the per-sample summary, making the filtering process transparent and auditable. Furthermore, the HAR parser analyzes only requests and does not parse the response bodies from the Telegram API host, since these are generated by the responder and contain no operator-side information.

Classification. The `classify.py` module is straightforward: it examines the completed `RunOutcome` and returns the highest applicable level of the diagnostic ladder. The possible outcomes, in order of priority, are represented in Table 4.8.

4.5.6 Persistence

For each sample, the orchestrator produces a summary JSON file and a plain-text run log in an output directory alongside the sample's analysis artifacts. The summary is the canonical serialization of the run outcome, providing a complete, reproducible narrative of the analysis that is independent of the database.

The `db.py` module logs each run into the shared corpus database. Each row is labeled with the extractor's name, and the write operation drops only the previously written rows for that sample under the corresponding tag before inserting the current data. The effect of this procedure is that re-running a sample will cancel out its previous contribution, but rows from other tools will not be affected, and duplicates will be prevented. Endpoints are stored in both normalized and raw values, with a label indicating the extraction stage from the most authoritative source, a confidence score, and a compact JSON context containing the full evidence. The run as a whole is also recorded, with its diagnostic label and a digest of its features. The entire process is executed in a single transaction that commits on success or rolls back on error, so the database is never in a partially written state.

4.5.7 Command-Line Interface

The command-line interface is handled by the `cli.py` module. It takes a single hash or a batch file containing one hash per line as parameters. It is possible to specify the archive root, the database path (or a flag to disable database use), the output override, and the log level. The module calls the orchestrator for each target and prints a short result line for each sample. Failures are isolated so that an exception in one sample does not stop batch processing. This approach enables the extractor to process an entire corpus efficiently while keeping per-sample fault isolation in line with the orchestrator's design.

4.5.8 Database Schema

The IoC extraction pipeline operates in distinct phases, with each phase maintaining its own state throughout execution. The corpus pipeline records data in `working.db` (Section 4.3.2), while the orchestrator writes to a

separate SQLite database (Section 4.4.3). Both databases use the sample's SHA-256 hash as the primary key, which enables records from different phases to be matched to the same sample. The subsequent IoC extraction phase writes the results back into `working.db`, resulting in the corpus database evolving into the central analytical repository. In contrast, the orchestrator's database has an active role as an operational log and is not structured for subsequent analytical processes. This database is updated as the orchestrator's processing queue advances, storing one row for each detonation attempt rather than a single verdict per sample, and it contains only data generated by the controller. The additional information required for analysis and results is distributed across other locations, including the sandbox's own database. Aggregating this information at query time for each query would be inefficient and error-prone.

Results snapshot The results of the sandbox analysis are aggregated into a results database that is constructed once by collecting three sources: job outcomes and timings from the orchestrator's log, per-sample payload counts from the sandbox database, and the completion state of the static and payload stages from the archive on disk. These data are condensed into one row per sample and one row per pass, each keyed by the SHA-256 hash for direct join with the corpus database. Table 4.9 summarizes the contents of this snapshot.

Table 4.9 Aggregated result tables of sandbox analysis.

Table	Contents
<code>sample_results</code>	One row per sample: overall outcome, per-pass status, attempt count, static-stage completion, and payload counts.
<code>pass_results</code>	One row per sample and pass, taken from the final attempt: status, sandbox task identifier, timings, and per-pass payload-analysis counts.
<code>run_summary</code>	Corpus-level totals and coverage figures recorded at build time.
<code>needs_unwrap_map</code>	Archive samples and the payload analyzed on their behalf.

Archive-to-payload map The `needs_unwrap_map` table contains the identity reconciliation described in Section 3.3.2. Each row represents an archive sample, identified by the archive's SHA-256. The table contains the sample's first-seen date and extension, any password recovered from the source metadata, the extraction outcome, the SHA-256 hash and file type of the payload analyzed by the sandbox, a flag that indicates if the payload is the archive or the content of it, and the sandbox task identifiers for both passes.

Archives categorized within a specific family can be traced to the payload responsible for their observed behavior. Sometimes a payload is shared among multiple archives and is classified as a repeat rather than as an unrelated sample.

4.6 Stealer Log Ingestion Pipeline

The victim-side workstream is implemented as a self-contained Python package, `poi_ingest`. It offers two commands through a single command-line interface: `rank`, which is the single-pass extractor explained in Section 3.5, and `graph`, an offline delivery-vector analysis outlined in Section 3.6. The package runs directly from its directory and uses a specific recent Python interpreter. This section explains the ingestion command, and Section 4.7 goes into more detail about the analysis command.

4.6.1 Package Architecture and Data Flow

The package is structured around responsibilities and reflects the data flow from raw archive to organized table. A layout-aware iterator generates a consistent triple for each victim; a manifest classifier and multiple format-specific parsers convert each victim's information file into a standardized schema; and the trajectory builder, in conjunction with shared URL and filter utilities, converts the victim's history into an ordered sequence of URL-stripped tokens. An orchestrator coordinates all components in one pass and manages the database schema. Table 4.10 enumerates the modules and their respective responsibilities. Only the numeric helpers, archive extractor, and optional signature engine interface with external systems. The rest of the components handle data transformations internally.

Table 4.10 `poi_ingest` submodules and their responsibilities.

Module	Responsibility
<code>cli.py</code>	Command-line interface
<code>log_iter.py</code>	Layout-aware iteration
<code>extractor.py</code>	7z archive extraction
<code>format_detector.py</code>	Manifest to format classification
<code>parsers/</code>	Per-format parsers (A, B, macOS, unknown)
<code>rank.py</code>	Single-pass extractor
<code>trajectory.py</code>	Per-history-file ordered URL-stripped trajectory builder
<code>url_utils.py</code>	Shared URL parsing utilities
<code>filters.py</code>	Common-host/token/URL denylists and predicates
<code>family.py</code>	YARA stealer-family identification
<code>db.py</code>	Database connector helper

4.6.2 Log Iteration, Format Detection and Per-Format Parsing

For each inner log, the `log_iter` produces a triple consisting of a stable identifier, a manifest of path-size entries, and a reader closure that returns the bytes of a specified file. The outer archive is extracted once to a

temporary directory using an external 7z binary, while nested archives are accessed directly in memory. The iterator then traverses the directory structure based on its layout. The identifier of a log is the SHA-256 digest of the inner archive bytes, allowing each table row to be traced back to its source.

The `format_detector` module classifies logs exclusively based on file names, without accessing their content. It applies an ordered cascade and returns upon the first conclusive match. The high specificity of macOS keychain artifacts gives them the highest priority, followed by the Windows "format B" variant, which is distinguished by some marker and account-token directories. The Windows "format A" is considered next, and an `unknown` fallback is used if no match is found.

The `parsers` sub-package dispatches each classified log to one of three concrete parsers, while logs classified as unknown are processed using a regular-expression scan of any information-like file. These parsers fill the shared `ParsedLog` dataclass with help from common functions: a key-value reader matches vendor field labels to standard names; an install-path decomposer extracts filenames from directory patterns and notes whether the filename is randomized; and a branding extractor determines the operator brand from embedded Telegram handles, URLs, or specific keyword markers.

The schema is deliberately kept simple and excludes victim-identifying fields like network address, geolocation, hostname, account name, hardware, and similar details. Instead, only essential data is kept for further analysis: the installer filename and log date for install-side ranking, the anonymized directory pattern and templating flag, and three install-independent header signals (the build version, build identifier, and operator brand).

4.6.3 Single-Pass Accumulation

The orchestrator `rank.py` starts by setting up the database and compiling family rules. Then, it traverses each outer archive and every inner log to extract names, identify families, and explore histories to help with host ranking and building trajectories.

Name extraction turns each victim's installer filename into a countable token. The filename is reduced to its basename, trimmed, lowercased, and stripped of a trailing browser download-collision suffix such as `(1)`, and a per-name accumulator records its cross-victim count. In parallel, the host side accumulates, per URL, how many victims visited it and how many had it as their most recent visit. Once aggregated across the corpus, these two accumulators become the installer-filename ranking and the host ranking, respectively.

Each parsed log also contributes up to two per-victim-side rows: one carries the install-path signals; the other carries the install-independent header signals (operator brand, build version).

All data is saved at the end of the processing pass. Each output is ranked or de-duplicated in memory before being written by a dedicated table writer, which drops and recreates its respective table. Per-victim side tables are de-duplicated by identifier to avoid primary key conflicts caused by identical inner-archive bytes appearing multiple times. Therefore, only the first occurrence is kept, and any collision is reported.

4.6.4 History Collection and URL Reduction

The shared history collector is the privacy-critical core of the ingestion. It uses the manifest to filter only history-shaped paths, with the same predicate that structurally excludes credential stores, and distinguishes plain-text histories from Chromium SQLite history databases. Text histories are scanned for both HTTP and HTTPS URLs, while any operator-banner Telegram links near the top of the file are suppressed. SQLite histories are copied to a temporary file, opened in read-only mode, and queried for their URL column, the maximum-timestamp "top" URL, and a bounded stream ordered by most recent first. Each URL is processed through the `url_utils` primitives, which lowercase the host, remove the port, user information, and a leading `www.`, and produce the three formats described in Section 3.5.6. In a single pass, the collector returns the set of per-victim URLs for host ranking, a single recency "top" URL, and one ordered stream per history file for the trajectory builder.

4.6.5 Common-Token Filters

The `filters` submodule manages noise reduction using three deny lists, each operating at a different level of detail and controlled by its own predicate. The host-level list detects widespread platforms via suffix matching and includes a wildcard for Google's country domains; it is accompanied by a brief allowlist of delivery-relevant subhosts, such as Google Drive. The exact host-path list targets specific noise paths, while the exact path-URL list excludes bare landing URLs and retains entries with deeper paths. These three lists enable the detection and filtering of non-delivery traffic across three distinct levels of detail. Both the host ranking and the URL-stripped trajectory process the complete path and utilize a unified composite predicate that removes a token if any of the three lists match.

4.6.6 YARA Family Identification

`family.py` implements lazy compilation rules that ensure that ingestion can proceed without failure even when the signature engine or rules are unavailable. Based on the manifest, three tiers of candidate files are identified for rules matching. Following the order of priority, first those with exact known information-file names are matched; then files with information-like names, and finally, as a fallback, the smallest remaining plain-text files. Family identification is performed by matching each file's bytes against the rules, and the first matching rule determines the primary family, while all triggered rules are recorded. The verdict is determined per victim and, like the install filename, is reconciled into a side table indexed by identifier.

4.7 Delivery-Vector Analysis Implementation

The `graph` command performs offline delivery-vector analysis, as described in Section 3.6, on the stored trajectories. The command loads the stored sequences without preprocessing and counts consecutive token pairs

into directed edge weights. The directed graph keeps only the edges seen by at least three victim streams, and operates at the URL-stripped level of granularity. A set of independent experiment scripts supplements this module. On each execution, the command drops and reconstructs all tables it generates, enabling the analysis to be repeated at any time.

4.7.1 Community Detection and the Two Graph Views

Each stored trajectory is read exactly as it was left by ingestion, without additional preprocessing. The frequency of each ordered pair of adjacent tokens across the corpus is counted, and these counts serve as the weights in a directed edge table, with one row per observed transition. This table enables the derivation of two distinct views of the data, as the applied measures require the graph in two formats.

The first view keeps the directed graph for measures that require edge directionality. An edge is only included if at least three victim streams have traversed it (Section 3.6.3). The nodes correspond to the URL-stripped tokens. It is guaranteed that self-loops are absent because consecutive duplicate visits were collapsed during ingestion.

The second view is an undirected projection used for community detection (Section 3.6.3). Each pair of hosts is connected by a single edge, weighted as the sum of the two directed counts between them. This edge is kept only if the combined weight surpasses a threshold that's even stricter than the one used for the directed graph. We then apply the Leiden algorithm, based on the Constant Potts Model objective, to this projection. For reproducibility, the reference Leiden library is preferred, executed with a fixed random seed and iterative refinement until convergence. If the reference library is unavailable, the native Leiden implementation in the underlying graph library is used; if neither is present, a sentinel is returned, allowing the analysis to proceed without community labels. Communities are relabeled in descending order of size, denoted as $C_0, C_1, \dots$, with C_0 representing the largest, to ensure label consistency across runs.

The resolution parameter is not set to a single granularity, but is varied over a configured range. For each value, we maintain the full per-host assignment and a short summary, which includes the number of communities, the fraction of hosts in the largest community, and the number of singletons. One value from this range is designated as the operating resolution and is used for subsequent joins. At the operating resolution, the analysis returns to the directed graph to identify cross-campaign infrastructure. For each host, we count the number of different communities with incoming edges. A host with edges from two or more communities is classified as a bridge, and a host with edges from three or more communities is classified as shared infrastructure. The community label at the operating resolution is then stored in a per-host table indexed by the graph node, with null entries for hosts that are excluded by the count threshold.

4.7.2 Pattern Mining and Family Attribution

Communities indicate which hosts are associated, while sequential patterns reveal the order in which victims traversed these hosts. The command extracts these patterns from the raw trajectories using PrefixSpan, retaining only closed patterns that meet both an absolute support threshold and a minimum length. These patterns are then recorded in a pattern table, where the auto-incrementing identifier also serves as a record of the discovery order.

Each pattern is attributed to the corresponding stealer families. For each pattern, the primary metric is the distribution of producing families, calculated over the distinct victims whose trajectories include that pattern. Instead of testing all patterns against each trajectory, the implementation creates an inverted index that links every token to the streams where it appears. A reduced candidate set of streams is identified by intersecting the postings for a pattern's tokens. Each candidate is then validated using an order-preserving subsequence test, which permits intervening hosts but requires the tokens to appear in sequence. Only confirmed victims are included, with each victim counted once, even if multiple browsers match. Victims without a family label are assigned to an explicit unlabeled category instead of being omitted. This process produces a per-pattern, per-family table aligned with the pattern table by insertion order.

4.7.3 Delivery-Vector Experiment Scripts

The delivery-vector findings illustrated in Sections 3.6.8 and 3.6.7, along with the validation experiments, are generated by a small set of stand-alone scripts external to the main command. These scripts share a module that defines the exogenous strip set, which is a curated denylist of generic delivery hosts that include public file hosts, cloud drives, link shorteners, open redirectors, and advertising-redirect networks.

The lure-ranking script examines the trajectory of each victim, locates the last hop in each browser's history that is not a generic delivery host, and then de-duplicates those hops across the victim's browsers. This process yields the recognizable head of named lures, the long tail of one-off lures and their size, the proportion of victims accounted for by the top hosts, and the niche lures that fall outside the community projection. The victim-assignment script addresses whether the communities function as delivery vectors by assigning each victim to a community in three ways: by the community containing most of the victim's hosts, by the victim's terminal host, and by the last non-sink hop. Each attribution is then joined with the installer filename, stealer family, and operator brand. These joins show not only the dominant signature in a community but also the distribution of a single build or operator across multiple communities. A brand-coverage probe re-parses a sample of the corpus without modifying the data; it measures the frequency with which the operator brand can be recovered, the number of distinct brands, and the extent to which brands join onto the rest of the data. It also cross-tabulates brand against family to demonstrate that the brand provides the discriminating axis that the family monoculture lacks.

4.7.4 Command-Line Interface

The ingestion and analysis stages are implemented as sub-commands in a single command-line interface. For the `rank` command, the user needs to provide the path to the corpus, the path to the database, a mode selector to restrict execution to a specific signal side, an optional label for the corpus, paths for export, and a switch for family identification. This command initiates the single-pass ingestion orchestrator and generates a summary for each side upon completion. The `graph` command requires the database path, two edge-count thresholds for the directed graph and undirected projection, parameters for resolution sweep and operating resolution, support and length thresholds for pattern mining, and toggles for community detection, pattern mining, and family attribution. Both the rank and the graph analysis are designed to handle faults by logging and skipping any malformed archives, so the process continues without interruption.

4.7.5 Stealer-Log Database

All ingestion and analysis states are stored in a single SQLite database operating in write-ahead-logging mode, without the use of an object-relational layer. Each table is managed, dropped, and rebuilt by the command responsible for writing it. The ingestion command manages the install-filename ranking and its two per-victim side tables (one for install-path signals and one for install-independent metadata), the per-victim family label, the history-host ranking, and the per-history-file trajectories. The analysis command processes the transition graph, the per-node statistics, the community scan and its per-resolution summary, the shared-infrastructure bridges, the frequent patterns, and the per-pattern family attribution. The graph-side tables are indexed by the `host_url_stripped` column containing the URL-stripped graph-node token. The history ranking keeps a separate bare `host` column for grouping. All per-victim tables share the `log_id` join key, which allows any victim to be reconciled to installer filename, install metadata, family, and trajectory. Also, family labels can be attached to the pattern attribution without re-reading the corpus.

5 Results and Analysis

5.1 Corpus and Detonation Coverage

The acquisition queries surfaced 8,113 candidate samples. File-type filtering removed 91 of these (58 Office documents and 33 out-of-scope file types), and the sanity check removed a further 25 as too small, leaving an admitted corpus of 7,997 samples. Sandbox analysis, however, incurs high computational costs, further increased by two-pass detonation and reruns of failed attempts. An analysis of the entire corpus would have exceeded the project's computational budget, so the analysis was reduced to a subset of 3,094 samples (38.7 % of the admitted corpus), selected in reverse chronological order by MalwareBazaar first-seen date. This ordering was intentional to prioritize the recovery of active operator infrastructure, as older samples are more likely to contain revoked tokens, while recent submissions are more likely to yield live indicators. Of the 3,094 submitted samples, **3,035** completed analysis and **59** failed to produce usable reports; the extractor was applied only to the successfully analyzed samples. All extraction rates in this chapter use the 3,035 completed analyses as the denominator. The 59 failures are excluded rather than counted as negatives, since the absence of an indicator in a failed run is considered an analysis artifact rather than a property of the sample.

Table 5.11 shows totals by family. The admission process creates a wide selection, but the recency-based filter can entirely exclude certain families. These families are evaluated based on the proportion of admitted samples with a first-seen date in 2026. As a result, families primarily composed of older members are greatly diminished, if not completely removed, in their representation. The counts analyzed per family serve as the denominators for any family-level rates discussed later in the chapter. The unevenness observed in these counts is also addressed when discussing threats to validity.

In Table 5.11 *Admitted* refers to samples included in the analyzable corpus, while *Detonated (2026)* counts the subset detonated in CAPE under the most recent-first selection, each with a 2026 first-seen date. *Coverage* represents the proportion of admitted samples that have been analyzed. Rows follow the acquisition allowlist from Table 3.1, sorted by the number admitted within each class. Off-allowlist labels, admitted via the `tag:telegram` catch-all, are grouped into a category called *Other*. The *Other* category represents the marginal yield of the cross-tag strategy: 102 samples (1.3 % of the admitted corpus) possess the Telegram tag and lack any allowlisted-family tag or signature. As a result, none of the sixteen per-family sweeps identified these samples, but only the `tag:telegram` query could have obtained them. Under the recency-first selection, two of the 102 samples reached the detonation stage, but only one produced analysis results. The Telegram tag overall exhibits a similar imbalance: among the 721 admitted samples that possess this tag, five have completed analysis. Therefore, the cross-tag query expanded the admitted corpus substantially more than it increased the analyzed yield.

The completed-versus-failed distinction matters again for the dominant NO_TELEGRAM class discussed in the

Table 5.11 Per-family corpus coverage.

Family	Admitted	Detonated (2026)	Coverage
<i>Class T1 – Telegram-native</i>			
Agent Tesla	1,703	908	53.3 %
XWorm	1,033	413	40.0 %
Snake Keylogger	998	163	16.3 %
Phantom Stealer	782	418	53.5 %
Blank Grabber	444	29	6.5 %
StormKitty	302	1	0.3 %
404 Keylogger	190	0	0.0 %
Phemedrone	72	2	2.8 %
ToxicEye	12	1	8.3 %
Masad Stealer	2	0	0.0 %
Prynt	0	0	n/a
WorldWind	0	0	n/a
<i>Class T2 – Telegram-capable</i>			
AsyncRAT	1,089	838	77.0 %
Lumma (LummaC2)	1,066	270	25.3 %
Umbral Stealer	105	27	25.7 %
DeerStealer	97	22	22.7 %
<i>Off-allowlist (tag:telegram catch-all)</i>			
Other	102	2	2.0 %
Total	7,997	3,094	38.7 %

following section. Two verification steps confirm that the samples belonging to this class represent true negatives rather than masked analysis failures. All 2,606 samples in this class originate from analyses that the sandbox marked as completed, and 2,447 of these (93.9%) dropped at least one payload during detonation. These samples executed sufficiently to unpack and write artifacts to disk, indicating that a configured Telegram token would, in most cases, have been accessible to the extractor. Since the corpus is a broad MalwareBazaar collection rather than a Telegram-specific set, a large non-Telegram majority is the anticipated result.

5.2 Telegram IoC Extraction

5.2.1 Extraction Outcomes Along the Diagnostic Ladder

Each analyzed sample is assigned to exactly one of the five diagnostic labels defined in Table 4.8, based on the first match in a descending hierarchy of Telegram-evidence strength. The distribution across the corpus is shown in Figure 1. PAIR_OK and TOKEN_ONLY represent samples from which the extractor recovered at least one indicator; together, these account for **173** samples (5.7 % of the corpus). API_TELEGRAM and TELEGRAM_OTHER categories correspond to samples that demonstrate Telegram involvement of decreasing strength, but from which the pipeline could not extract any Telegram IoCs: either the sample did not expose its malicious behavior, or the extractor was not able to obtain the token from the analysis artifacts. The remaining NO_TELEGRAM category does not yield an extractable token and dominates the corpus, as expected of a broad MalwareBazaar pull.

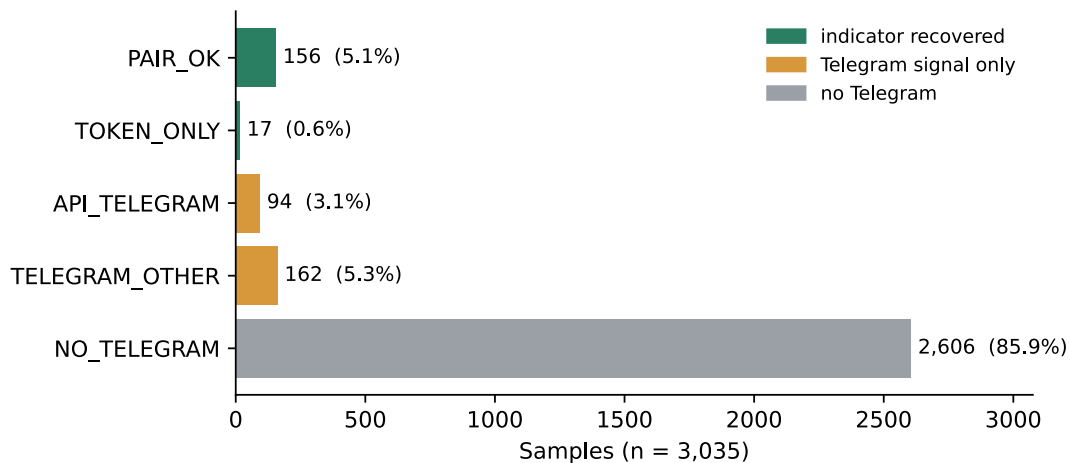


Figure 1: Distribution of the 3,035 analyzed samples across the diagnostic ladder.

The overall 5.7 % rate understates the extractor’s effectiveness on its intended sample set, as it is diluted by the predominance of non-Telegram samples. When aggregating the four Telegram-bearing tiers, **429** samples (14.1 % of the corpus) exhibit a Telegram signal. Among these, the 173 samples in the top two rungs indicate that the extractor recovered a concrete indicator from **40.3 %** of all Telegram-involved samples. This wide percentage difference is taken into consideration by the API_TELEGRAM and TELEGRAM_OTHER samples, which are cases

where Telegram was demonstrably referenced (such as through a contacted API host, an embedded deep link, or a matching signature), but no token survived in any recoverable artifact. It can be hypothesized that, in these cases, the token is built at runtime, or that the sample detects the sandbox and never executes the code that builds it.

5.2.2 Extraction Yield by Family

The overall yield shows considerable variation among different malware families. Table 5.12 details, for each family, the total successful analyses performed⁵, how many samples produced at least one Telegram indicator (PAIR_OK or TOKEN_ONLY tiers), the corresponding recovery rate, and the number of endpoint records collected ((bot_token, chat_id) pair).

Recovery efforts are mainly focused on a few stealer families. Specifically, Phantom Stealer, Snake Keylogger, and Blank Grabber each yield an indicator from approximately one in five analyzed samples (18.4 %, 19.1 %, and 17.9 %, respectively), which is substantially higher than the corpus-wide rate of 5.7 %. Notably, Phantom Stealer alone is responsible for 74 of the 173 indicator-bearing samples and 74 of the 187 endpoint records.

The difference between families that use Telegram natively and those that are capable of using Telegram doesn't align as closely with the observed split as the acquisition classes indicate. Although Agent Tesla and XWorm are classified as Telegram-native, they yield an indicator from only 3.6 % and 2.4 % of analyzed samples, respectively. These rates are comparable to those of Telegram-capable families such as AsyncRAT (2.3 %) and Umbral Stealer (3.7 %). Two families with substantial sample sizes, Lumma (265 analyses) and DeerStealer (22 analyses), produced no tokens. The most likely explanation is that these families usually create the token during runtime, save it within an encrypted configuration, or recognize the sandbox environment and avoid proceeding to the exfiltration stage, rather than simply lacking a Telegram channel. Single-sample families lack statistical significance; for instance, ToxicEye reports a 100 % recovery rate, but this is derived from only one analysis. Among the samples that possess only the tag:telegram query strategy and no other signature or tag in the allowlist, only one produced analysis results but contained no IoCs. If we instead consider those obtained through a different strategy but still containing the Telegram tag, four out of five samples produced IoCs, demonstrating high accuracy in using the Telegram tag to identify infostealers that exfiltrate through Telegram.

As with all per-family data in this chapter, these rates are affected by the recency-heavy coverage in Table 5.11 and should be seen as a lower bound on recall for this subset, not as an intrinsic characteristic of each family.

5.2.3 Indicator Yield and Confidence

Analysis of the 173 token-pair-bearing samples yielded **187** Telegram endpoint records, with each duet (bot_token, chat_id) pair constituting one record, where the chat ID can be null. The record count is

⁵Numbers in this column differ from the ones in Table 5.11 because *Detonated* refers to the samples that were detonated by CAPE, while *Analyzed* considers only the successful analysis.

Table 5.12 Per-family IoC extraction yield.

Family	Analyzed	With IoC	Recovery	Endpoints
<i>Class T1 - Telegram-native</i>				
Phantom Stealer	402	74	18.4 %	74
Agent Tesla	889	32	3.6 %	44
Snake Keylogger	162	31	19.1 %	33
XWorm	410	10	2.4 %	10
Blank Grabber	28	5	17.9 %	5
ToxicEye	1	1	100.0 %	1
Other (T1)	3	0	0.0 %	0
<i>Class T2 - Telegram-capable</i>				
AsyncRAT	825	19	2.3 %	19
Umbral Stealer	27	1	3.7 %	1
Lumma (LummaC2)	265	0	0.0 %	0
DeerStealer	22	0	0.0 %	0
<i>Off-allowlist (tag:telegram catch-all)</i>				
Other	1	0	0.0 %	0
Total	3,035	173	5.7 %	187

higher than the number of indicator-bearing samples because each sample carries one token, but a token can be paired with multiple chat candidates, yielding one record per candidate. Of the 187 records, **170** bind a chat to the token, while **17** are `TOKEN_ONLY`; therefore, 90.9% of recovered endpoints are structurally complete pairs that document both the bot and a destination chat. This figure reflects record completeness rather than the binding's correctness because the chat is attached by bounded proximity and the regular expressions that identify it are prone to false positives, as described in Sections 3.4.1 and 4.5.2. Whether a pair is actually live, with a reachable bot and an accessible chat, is answered by the validation of Section 5.2.9 rather than by this count.

Each endpoint is assigned a confidence score based on the most authoritative source contributing to its identification (Section 4.5.3). Figure 2 presents this distribution. The highest tier, which covers endpoints seen in decrypted HTTPS traffic produced by `mitmdump` (har files) or in family configurations extracted by CAPE, includes **108** records, making up 57.8%. This shows that most extracted IoCs are supported by authoritative evidence rather than simple string matches. The medium-confidence tier that consists of matches in static or unpacked-payload strings is composed of 62 records (33.2%), while the low-confidence tier corresponds exclusively to the 17 token-only endpoints (9.1%).

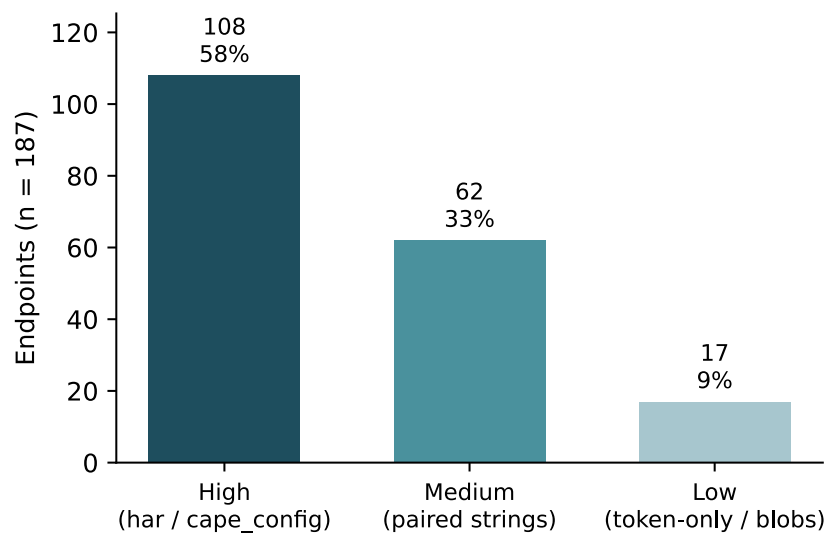


Figure 2: Recovered endpoints by confidence tier.

5.2.4 Static Versus Dynamic Distribution

Of the 187 endpoints, 108 are derived from decrypted HTTPS traffic and 31 from behavioral report blobs, indicating that **139** endpoints (74.3%) originate from dynamic observation. The remaining **48** endpoints were identified through regular-expression scanning of strings, but, in each instance, the relevant string was found in a payload dropped at runtime rather than in the distributed binary. No endpoint in the corpus was detected in the shipped sample by static scanning alone. This outcome can be attributed to the malware's packing process, which

hinders automatic static analysis from extracting relevant IoCs. Despite this limitation, the static routine becomes effective after a dynamic pass has revealed the unpacked payload. Therefore, we can state that static and dynamic analysis do not represent independent methods for identifying the same indicators in this context; instead, they are dependent on each other.

5.2.5 Pass Contribution and Corroboration

Each endpoint documents the detonation passes during which it was observed (Section 3.3.3). Through this data, we can evaluate the two-pass protocol by attributing to each recovered endpoint the `standard` instrumented pass, the `no_monitor` uninstrumented pass, or both. This distribution is presented for the 187 endpoints in Table 5.13.

The two passes provide complementary coverage rather than redundancy. Only **30** endpoints (16.0 %) were observed in both passes, while the remaining 157 were recoverable only in one, since no endpoint was detected by static analysis alone. The uninstrumented pass uniquely contributed **84** endpoints (44.9 %) that the instrumented pass did not produce, which aligns with the sandbox-aware suppression described in Section 3.3.3, and demonstrates that network eavesdropping alone recovers most Telegram IoCs. In these cases, samples detect userland hooks and suppress exfiltration behavior under full monitoring, but reveal it when instrumentation is absent. In contrast, the instrumented pass uniquely contributed **73** endpoints (39.0 %). This pass is as fundamental as it is the exclusive source of behavioral-report and family-configuration evidence, which the uninstrumented pass cannot generate. Endpoints observed only in monitor blobs or CAPE configuration are attributed here. When shared endpoints are counted for both passes, **103** endpoints (55.1 %) were visible to the standard pass and **114** (61.0 %) to the `no_monitor` pass.

Neither pass alone is sufficient, and a single instrumented detonation would have missed the 84 `no_monitor`-only endpoints, while a single uninstrumented detonation would have missed the 73 standard-only endpoints. The same pattern is also present at the sample level, where, among the 173 indicator-bearing samples, 59 yielded an endpoint only under instrumentation, 84 only without it, and 30 under both conditions.

Table 5.13 Endpoint recovery by detonation pass.

Recovery pass	Endpoints	Share
Standard (instrumented) only	73	39.0 %
No Monitor (uninstrumented) only	84	44.9 %
Both passes	30	16.0 %
Total	187	100 %

5.2.6 Distinct Command-and-Control Infrastructure

Since the normalized value for each endpoint corresponds solely to the bot token, the number of distinct C&C endpoints in the corpus directly reflects the number of unique normalized values. The 187 endpoint records reduce to 110 distinct bot tokens and 94 distinct chat identifiers. This reduction indicates infrastructure reuse, in which identical bot tokens appear in multiple samples and individual operator chats are targeted by more than one token. This shared infrastructure is the source material for the operator-cluster analysis discussed in the next subsection.

5.2.7 Operator Clusters from Shared Infrastructure

The reuse seen in the different infrastructure counts can be structured as a clear grouping. The recovered endpoints are shown in a graph, where each node represents indicator-bearing samples, unique bot tokens, and different chat identifiers. Edges connect each sample to the tokens it contains and each token to the chats it targets. Each connected component in this graph represents an *operator cluster*: samples that share a token or have tokens linked to the same destination chat are grouped together, while samples without shared infrastructure remain separate. Family labels or data liveliness are unnecessary for representation, since it relies on normalized indicator values.

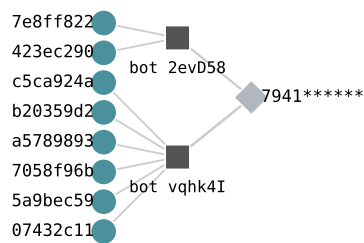
Cluster structure. The 173 indicator-bearing samples constitute **96** distinct components. Of these, **38** components link more than one sample, collectively accounting for **115 samples (66.5 %)**. Thus, two-thirds of the samples from which an indicator was recovered are associated with at least one other sample through shared operator infrastructure. Linkages occur along both available axes. In total, 34 tokens recur across multiple samples, representing the most common form of reuse. Additionally, **11** clusters are connected by a chat that is addressed by more than one distinct token, indicating instances where an operator rotates the bot while retaining the same destination. The largest clusters contain eight samples: one consists of two Phantom Stealer tokens funneling into a single chat, while the other involves a single Snake Keylogger token reused across eight samples. Both clusters are depicted in Figure 3 as clusters A and B.

Clusters that cross family labels. Out of the 38 multi-sample clusters, five span over more than one family label, including 19 samples; however, we need to exclude one of these five clusters from further analysis. This cluster consists of seven samples linked by a single token whose 35-character secret exhibits the lowest entropy among all 110 recovered tokens, measuring 3.12 bits per character over a 12-symbol alphabet, compared to a minimum of 4.19 and a median of 4.61 for the remaining tokens. This low entropy suggests that the token is a templated builder default rather than a configured credential, rendering its cross-label span inconclusive. Although it is included in the headline counts because the extractor recovered it, it is omitted from the figure. The remaining four cross-family clusters, including twelve samples, are presented in Figure 3 as components E to H, together with

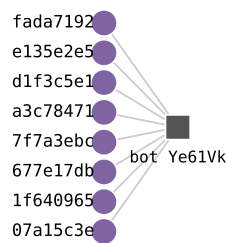
the four largest single-family clusters (A to D) for context. These clusters cross family labels in two distinct ways. In clusters F and G, a single bot token appears in samples from different families, whereas in clusters E and H, two separate bots send messages to a shared chat across family boundaries. The grouping is structural only; whether each clustered endpoint is also active is addressed separately in the validation presented in Section 5.2.9.

Table 5.14 Full SHA-256 hashes of the samples in each operator cluster of Figure 3.

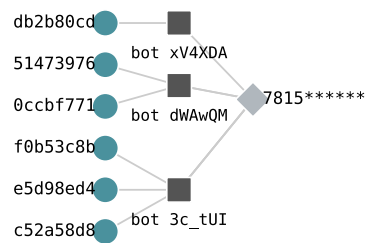
Cluster	Family	SHA-256
A	Phantom Stealer	07432c11f1bb3a5f299aab53022620ece4ca677dc81f18cb1f5ce9c41b6b4553
	Phantom Stealer	423ec290859607655dfce4f545692bae1f1074a8a7275c5bd8303f89429f2219
	Phantom Stealer	5a9bec59f6368482d789e8e369b347a49d42e397133e1a90ebec71b16a58f4f6
	Phantom Stealer	7058f96bcc11886647b2b264535255fe6422555b0cf878ddd2bec4c4a9e1d126
	Phantom Stealer	7e8ff8227296ccf0c4ebb40ac757789ea923ac80cfc4f667dccc04df7bcc673
	Phantom Stealer	a578989343df52383e1009d0f1fb9773a96fbb83d1548c557b2f619ef224d368
	Phantom Stealer	b20359d275dd556b8a25531dd2acc933c945e989855fdc881c438cfb0cb471a0
	Phantom Stealer	c5ca924a6c158fb9d273e2450c3cdb5a92f02433ad87dfdf2977e1aa9fb16348
B	Snake Keylogger	07a15c3e77551af54b6f3d90cdd52cabe4b1a7aea22cab447025569d5bb1874d
	Snake Keylogger	1f6409653cb9163caee32ecc1652b8f7a778f44c492e6c74ca652505c37313d8
	Snake Keylogger	677e17dbef308984f2dd543cdb461377829871b9b958f4963636a51fef53108
	Snake Keylogger	7f7a3ebc3c537133f40c9796460912778f737c90de32e6e8861bc6fb2431ebbe
	Snake Keylogger	a3c7847164f2c9a18c8c6bed01241b84fffbfd6d866195e92f556ba0a596d0428
	Snake Keylogger	d1f3c5e1300ab1be652860818d9f82bc06b91cddad6d0a46eb7ea726524de100
	Snake Keylogger	e135e2e507f3d17a60aba2dac1356350eb90b1194981ed37e1f88022739bee79
	Snake Keylogger	fada71925be3d53eee961507cb43bc2ee409d406770c7ddb7f2f06afae85a456
C	Phantom Stealer	0ccbf7714945cf60d24b6fc417160c6b188bb39cc14537e41063076c7fb28518
	Phantom Stealer	5147397623337963d66ef5d84ae383c9e72c06da6f9836c6ce1771d4d0dff4f4
	Phantom Stealer	c52a58d86ed607a6e49ea632eb609bf08180ceae06249576633c4a6fd63e5488
	Phantom Stealer	db2b80cd7be5c7548932ce10c0ee09907d9fbbfbefb8c17e213d3178492fc4c955
	Phantom Stealer	e5d98ed469c71dbb6e524f31b34d76e05b97f8b03d6c0bcc1c965a68c001b299
	Phantom Stealer	f0b53c8bc902424b42fb165a041f58a713ba427bb2e9b47ce56fe8e061f32a16
D	Phantom Stealer	02eb6ea3994a71eb9d32762d30acfc9f2c38e2fd118043154f6223c10760eb37
	Phantom Stealer	19edf8e473310efee3dcde577c3e76d8f215c1bba6815ffe3952b5001e978c4a
	Phantom Stealer	54fe353ca8d2abb920624cb0314f811ef03f80a57c2199e8f65a25300f86ad3e
	Phantom Stealer	9e57f2f14606901c01d6d1cb6d7ff78ae5615a765f5798887e666b7328cc739b
	Phantom Stealer	c68697eaa5b3e875e4d98fe9380e559289def03d61b141ad860ab78e9ff4ba8
E	Agent Tesla	37a79153d2ff85fdb654c65b12c77fcf8c86272eb2fbc12e39c14fcf50fccd72
	Agent Tesla	478db7c81a60aedfb07b367f7de642fd4b7bcb9b8fdbab2a93ecd91731d869a
	Agent Tesla	a1e7238bffe6f64643869edc5cd52b9c9cb5b080b3a72f2363f8601a82a8b808
	Phantom Stealer	663c470421a43b880641026e8a36204f02249e669ca92c8f1b3fb216cd8306d0
F	AsyncRAT	00fe40a8f1cacf61bc594e3fb3878418792138a670acce546f1ac3c1697c6da6
	AsyncRAT	7aa63a4b896f5e3679e06bd9dd4565a356f2e9fbd95eb6966b0b0eb03c85c24f
	XWorm	dbc31a4c65b7cee1bca3e93df0e2f3573c652a2af20ec93b11d7cd1a737ba2ab
G	Phantom Stealer	1e0bbcaa4d9b3f4c144e10dad6fb9ecbde607e3c48c2d3194195f56852ef8ffb
	Snake Keylogger	a26018fa571c95f994b40c6ff3c5eb99d8f5e35b4ec819819ff559b5fc5ca1ce
	Snake Keylogger	c53e0727ef498dc29d9a90f4b22bab91fd8e8646b9b0dc732738a08d1cd0f45d
H	Phantom Stealer	3c23073583bd33a068be551134983e7958884cb7656a4da03d9cc737b262f1ee
	Snake Keylogger	79e6b2c3d010500745a6a5a68b89b3453e16eca3ff359477718453301c17b034



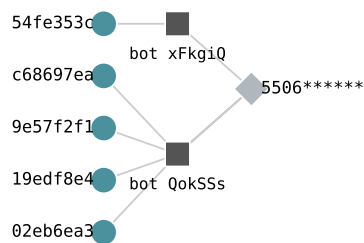
A. Phantom



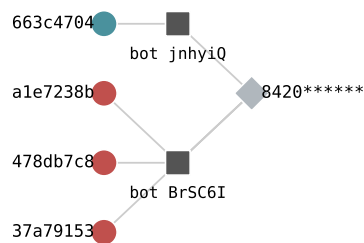
B. Snake



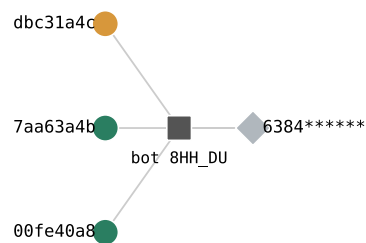
C. Phantom



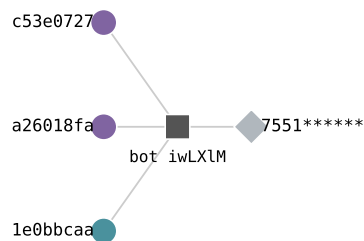
D. Phantom



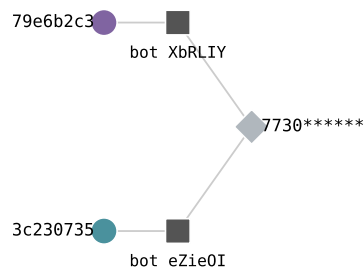
E. AgentTesla/Phantom (cross)



F. AsyncRAT/XWorm (cross)



G. Phantom/Snake (cross)



H. Phantom/Snake (cross)

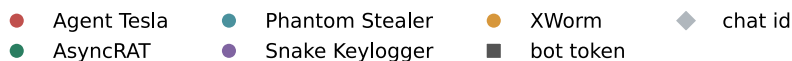


Figure 3: Operator clusters as connected components. Circles are samples, distinct by family label; squares are bot tokens; diamonds are destination chats, abbreviated to their first four digits. The full SHA-256 of every clustered sample is listed in Table 5.14.

5.2.8 Threats to Validity

There are four limitations to consider when interpreting these results. First, the extraction rate shows the minimum level of recall, not the maximum. The combined `API_TELEGRAM` and `TELEGRAM_OTHER` tiers include 256 samples that interacted with Telegram but did not yield a token. An unknown proportion of these may conceal tokens that were not observed, either because they were derived at runtime, present only in undecrypted traffic, or located in a payload stage not reached by the sandbox. The reported yield reflects only the amount recovered in this specific analysis setup, rather than the entire latent token population in the corpus.

Second, the 17 low-confidence endpoints are identified through byte scans of CAPE's report and monitor blobs. This method cannot distinguish between values that appear in real network traffic and those that are just embedded as strings. These endpoints are assigned low confidence for this reason and should be regarded as investigative leads rather than confirmed indicators.

Third, the way the corpus is composed and the recency-based sampling create bias in the family distribution of the recovered infrastructure. Coverage is uneven: AsyncRAT and AgentTesla were analyzed at 77 % and 53 % of their admitted samples, respectively, while StormKitty (1 of 302 admitted) and 404Keylogger (0 of 190) were mostly postponed for later analysis. Both undersampled families are known to exfiltrate data via Telegram, so a comprehensive analysis would likely recover more from these families than the current subset indicates. The family figures reported are based on those with significant coverage and should be understood as specific to this group, rather than representing rates for the entire population.

Fourth, the family labels come from MalwareBazaar's tags and signatures, which also contributed to shaping the selection of the corpus. Because of this, the results for each family might reflect any inaccuracies in these labels. Several families in the code are closely connected, such as the StormKitty and Phantom lineage or the AsyncRAT and XWorm pair. The same bot token might show up under different labels in the recovered data. So, a token's family assignment might depend more on the reporter's labeling choice rather than its actual lineage. This means that family-level recovery and liveness should be viewed as conditional on these labels, not as absolute indicators of family behavior.

These four caveats highlight possible underestimations or the need for further conditioning of the reported figures. In contrast, the chance of over-counting is tightly controlled. The extractor reduces false positives by enforcing structural rules: a token must match the precise Telegram bot-token format, and a chat is included only if it is directly associated with such a token within the same request, configuration entry, or byte sequence. Laboratory-specific values are intentionally excluded because the HAR parser does not analyze response bodies from the Telegram API host, and synthetic identities created by the responder are never added to the evidence stream. Consequently, the suppressed-synthetic counter stays at **0** throughout the corpus (Section 4.2.2). The

live-infrastructure validation in Section 5.2.9 offers an external check, since each endpoint resolving to a reachable bot is guaranteed to be a true positive. As a result, the recovered figures serve as a conservative lower bound, not an overestimated count.

5.2.9 Live-Infrastructure Validation

Scope and ethical framing. All results analyzed so far were generated entirely offline, as the sandbox did not contact the genuine Telegram infrastructure, and the extractor accessed only local artifacts. To verify if the recovered endpoints were active C&C channels instead of outdated or malformed strings, a simple validation step was performed by querying each endpoint against two read-only methods of the Telegram Bot API: `getMe` to verify if a token belongs to an active bot, and `getChat` to check if the associated chat is accessible. Therefore, no messages were sent, no updates retrieved, no content read or written, and there was no interaction with operators or victims; the probe solely observed the reachability metadata provided by the API to the token holder.

Separation from detonation. This step reconciles the validation process with the host-only posture maintained throughout the study. Detonation and validation are distinct procedures with different risk profiles. Detonation involves executing untrusted code, and a sample with access to a live network can perform any action its operator intended, such as fetching additional payloads, contacting third parties, or exfiltrating real data. For this reason, the sandbox remains air-gapped without exception. In contrast, validation does not involve these risks because the request is issued by the analyst using a documented and public API method, and returns only whether the credential is valid and the public metadata of the associated chat. As stated above, no information is sent to the operator, and no message history is accessed. Both results of obtaining a revoked-token error or a live-bot reply are harmless outcomes.

Method. Each distinct recovered endpoint was probed once. A `getMe` success classifies the token as *active*, while an HTTP 401 or 400 response indicates that the token is revoked or malformed and therefore inactive. For active tokens, `getChat` on the bound chat is used to determine whether the destination remains *accessible*. The validation set is composed of 110 distinct tokens recovered by the extractor, which bind to 108 distinct token-chat pairs

Liveness. When probing the endpoints, **58 out of 110 unique bot tokens (52.7 %) resulted as active**, while the other 52 encountered `getMe` authentication or request errors, suggesting they were either revoked or malformed. Among the 108 distinct token-chat pairs, **55** resolved to fully live chats, defined as active bots with accessible bound chats, with the majority of these being private chats (54) and only 1 private group. Approximately half of the recovered tokens stayed active for several months after sample collection. These findings suggest that

Telegram-based C&C endpoints tend to operate for medium-to-long periods, and that token extraction provides actionable indicators rather than solely historical data.

Liveness by family. The distribution of live infrastructure is highly concentrated within specific families. This distribution of active distinct tokens for each family is presented in Figure 4. **Phantom Stealer** is predominant in both the recovered and surviving infrastructure, with 39 of its 46 distinct tokens (85 %) remaining active. This accounts for more than two-thirds of all live tokens in the corpus. On the other hand, the remaining families are mostly inactive: AgentTesla (5 of 18) and SnakeKeylogger (1 of 16) provide many recovered tokens, yet only a few are actively used. This pattern might suggest a higher turnover among operators or that the samples from these families in the collection are relatively older.

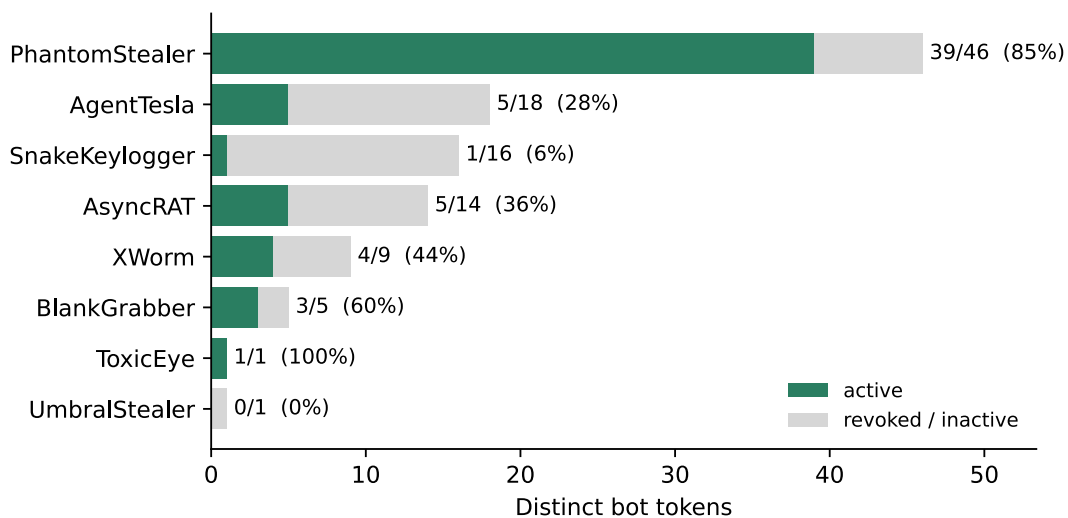


Figure 4: Live versus revoked (or malformed) bot tokens per malware family.

The validation establishes an external lower bound on extractor precision. Since each of the 58 active tokens is, by definition, a syntactically and semantically valid bot credential, inactive tokens are not necessarily false positives, as a revoked token represented a real credential at the time of sampling. As a result, the 52.7 % liveness rate shows ongoing operation rather than just how accurate the extraction was.

5.3 Stealer-Log Results

This section presents the results of executing the stealer-log pipeline described in Sections 4.6 and 4.7 on the complete archive corpus. The analysis starts with identifying the ingestion denominators and install-side signals, then presents a report on the family distribution to guide future analyses. The section details the results for delivery vectors, covering the transition graph and the causes behind the collapse of connected components, as well as the Leiden communities and the resolution sweep. It further characterizes shared delivery infrastructure, examines

sequential patterns, analyzes the operator-brand axis, and ranks direct last-hop lures. The section concludes with validation experiments and empirical confirmation of the structural bias discussed in Section 3.6.10.

All figures are derived from the production database, which is generated by the `rank` and `graph` commands. Throughout the analysis, the trajectory token is url-stripped with a 75-token cap, as described in Section 3.5.7. The operating resolution for each community-conditioned value is set to $\gamma = 0.05$, following the methodology in Section 3.6.3.

5.3.1 Ingestion Coverage and Denominators

The extractor analyzed **31,388** distinct victim logs, each identified by the SHA-256 hash of the embedded archive. Each stealer log influences a particular subset of downstream signals depending on its contents; therefore, the analyses are not based on a shared common denominator. Table 5.15 presents the population associated with each signal. Metadata represents the broadest category, as nearly every parsed system-information file reveals at least one header field. In contrast, admitted browsing trajectories constitute the narrowest category, which requires a history file that persists after the cleaning procedures outlined in Section 3.5.7. The delivery-vector analysis is conducted exclusively on this narrow trajectory population of **6,940** victims, and all subsequent graphs, community analyses, pattern identifications, brand assessments, and lure figures utilize this group as the denominator.

Table 5.15 Per-signal ingestion coverage. The trajectory population is the denominator for the entire delivery-vector analysis.

Signal	Victims	Share of ingested
Logs ingested (distinct log_id)	31,388	100.0 %
System-information header (metadata)	30,957	98.6 %
Stealer-family label (YARA match)	21,510	68.5 %
Usable installer filename	17,698	56.4 %
Admitted browsing trajectory	6,940	22.1 %

5.3.2 Install-Filename Ranking

The installer-filename ranking yielded 2,999 distinct normalized names. The leading entries, presented in Table 5.16, are composed of a small set of randomized build names rather than social-engineering pretexts. The two most prevalent names, `1s81j5.exe` (5,440 victims) and `koc0dj3.exe` (1,450 victims), are opaque alphanumeric stems with low filename entropy. This template recurs throughout the remaining leading entries.

The only descriptive names among the most frequent are the generic default `install.exe` (3,417 victims) and a `licence_version_loader.exe` (85 victims). These findings form the empirical foundation for the negative result mentioned in Section 5.4.8: the installer filename indicates the build that generated the log, not the social-engineered lure used for delivery, and therefore cannot serve as a per-victim delivery-vector label. Web searches were conducted for each of the most common names using both quoted and unquoted syntax. Generally, the queries displayed sandbox analysis results that confirmed install names but did not provide delivery lures.

Table 5.16 Head of the installer-filename ranking.

Normalized name	Victims	Stem entropy
1s81j5.exe	5,440	2.25
install.exe	3,417	2.52
koc0dj3.exe	1,450	2.81
6url4x8.exe	271	2.81
5gfpjxh.exe	237	2.81
youspnj.exe	228	2.81
1g65c4.exe	189	2.58
licence_version_loader.exe	85	3.46
falcon.exe	72	2.58
1j11j3.exe	63	1.46

5.3.3 Stealer-Family Distribution

The YARA scan identified 21,510 logs, with only four malware families present in the corpus: Vidar (13,803 logs, 64.2 %), MacSyncStealer (5,250, 24.4 %), SHubStealer (1,253, 5.8 %), and StealC (1,204, 5.6 %). The distribution shifts significantly when considering only victims whose browsing history was stolen, and who generated a history trajectory. Among the 6,940 trajectory victims, 4,837 (69.7 %) are associated with Vidar, 1,796 (25.9 %) lack a family label, and 307 (4.4 %) are attributed to MacSyncStealer. SHubStealer and StealC do not contribute any trajectories, as neither family exfiltrates browser history in this corpus. Therefore, of the trajectory victims with a family label, 94 % are Vidar. Figure 5 illustrates this mechanism by showing, for each family, the proportion of labeled logs that produced a browsing trajectory. Naturally, the two families that do not exfiltrate history remain at 0 %.

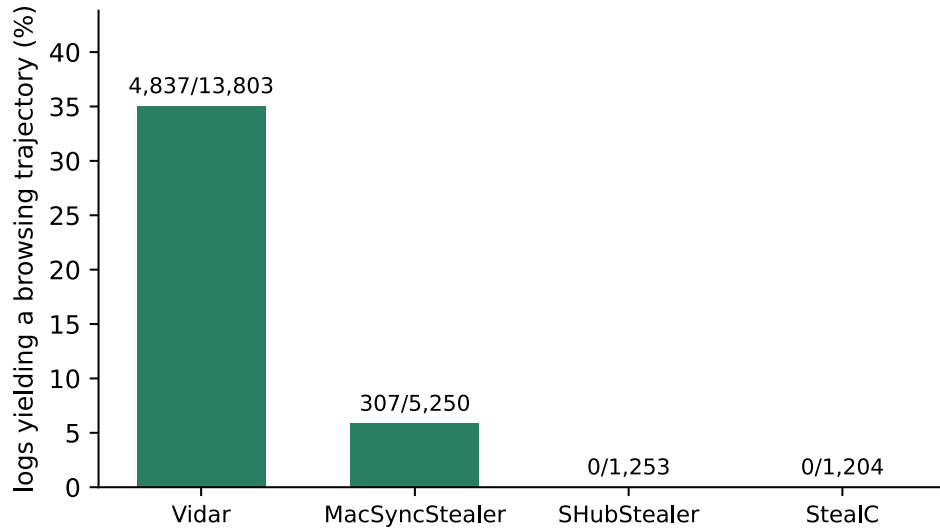


Figure 5: Share of each family's YARA-labeled logs that yielded a browsing trajectory.

5.4 Delivery-Vector and Point-of-Infection Results

The instruments of Section 3.6 are applied to the 6,940 trajectory victims in turn. The graph and its components come first, because their failure is what motivates the community's step. Before constructing the transition graph, two simpler victim-side aggregates motivated the move to community detection. The host-frequency ranking (how many victims visited each host) produced a small set of recurring delivery hosts that repeated across otherwise unrelated victims, and the most-recent ("top") URL concentrated on the same endpoints. This motivated the transition graph and community detection.

5.4.1 The Transition Graph and the Component Collapse

Counting adjacent token pairs within the stored trajectories reveals **273,535** unique directed transitions, with **106,821** exceeding the three-stream edge threshold. The thresholded graph contains **84,573** host nodes. Analysis of its weakly connected components confirms the pathology described in Section 3.6: there are 1,143 components, but the largest one encompasses **68,782** hosts, representing 81 % of the graph. Having a single large component that covers four-fifths of the hosts does not work well for campaign analysis because the shared file hosts and redirectors used across different campaigns combine separate delivery chains into a single chain. As a result, the component identifier cannot determine campaign membership for a specific host, which is exactly the issue the Leiden step aims to address.

5.4.2 Ecosystem Communities and the Resolution Sweep

Community detection is performed on the undirected projection, retaining only edges with a combined weight of at least 5, thereby preserving **34,501** hosts at each resolution. Table 5.17 presents the results of the parameter sweep. As γ increases, the number of communities increases and the size of the largest community decreases, with no abrupt formation of a giant community at any parameter setting. This result confirms that the Constant Potts Model behaves as expected and shows that the grouping is not caused by resolution limitations. At the selected operating point $\gamma = 0.05$, the largest community contains 264 hosts, representing 0.77 % of the projection, compared to the 81 % encompassed by the giant weakly connected component in the same graph. Figure 6 displays the two groupings as rank-size curves: the weakly connected components put 81 % of hosts in a single giant blob, while the Leiden communities at $\gamma = 0.05$ spread the hosts across a long tail with no dominant unit.

Table 5.17 Leiden+CPM resolution sweep over the 34,501-host projection.

γ	Communities	Largest size	Largest share	Singletons
0.01	2,779	617	1.79 %	57
0.02	3,117	452	1.31 %	97
0.05	3,824	264	0.77 %	154
0.10	4,549	131	0.38 %	173

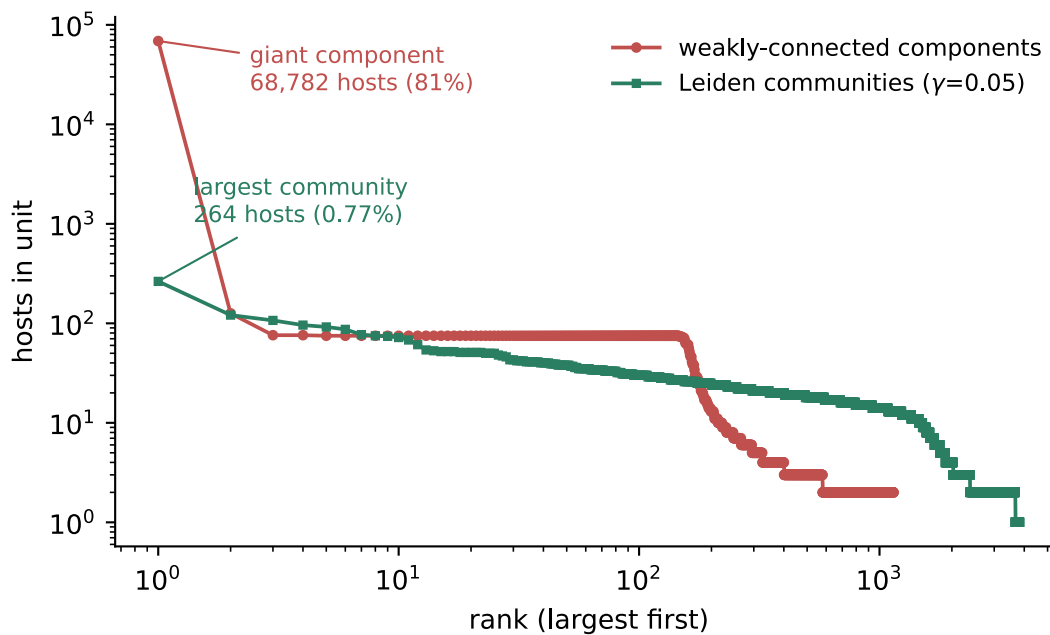


Figure 6: The same graph under two groupings, by descending unit size (log-log).

Out of the 3,824 communities identified at $\gamma = 0.05$, not all are substantial. The size distribution shows a long tail: three communities have at least 100 hosts, 25 have at least 50, 402 have at least 20, and 1,509 have at least 10. The rest are mostly pairs and 154 singletons. The 1,509 communities with 10 or more members include 27,666 of the 34,501 hosts, making up 80 % of the total, and these communities carry the meaningful signal. The host representatives, which are the nodes with the highest PageRank in each community, are easily recognizable as documented infostealer lure themes. The theme of each community is determined by manual inspection of the representative host and its co-clustered domains. Table 5.18 presents the largest communities. The dominant community, C_0 , includes the cracked unlock tool and Android firmware ecosystem. The remaining communities are categorized as firmware and FRP-bypass sites, link cloakers and traffic-distribution endpoints, movie-piracy portals, search hijackers, and a YouTube channel.

Table 5.18 Largest community per lure theme at $\gamma = 0.05$, by representative (highest-PageRank) host. Paths are truncated for width.

Community	Size	Representative host	Theme
C_0	264	<code>econventa.com/fp.engine</code>	cracked unlock-tool / firmware
C_1	121	<code>href.li</code>	redirect / cloaker
C_2	107	<code>samfw.com/blog/...frp-tool</code>	Samsung FRP / firmware
C_5	87	<code>clck.showmelinks.com</code>	traffic distribution
C_8	74	<code>moviebox.ph/web/searchResult</code>	movie piracy
C_{10}	68	<code>xfinder.pro/q</code>	search hijacker
C_{19}	51	<code>youtube.com/@unlocktoolpro</code>	tutorial-channel lure
C_{24}	50	<code>filecr.com/search</code>	cracked-software portal

5.4.3 Shared Delivery Infrastructure

Bridge analysis of the directed graph produces 3,262 hosts with edges originating in at least two communities. Of these, 1,048 connect three or more communities and are categorized as shared infrastructure. Since the trajectory token records the specific resource rather than just the host, the most important bridges become concrete infrastructure indicators across campaigns rather than generic platforms. Table 5.19 presents the most frequently reused resources. The portal `unlocktoolpro.com` is visited by 119 different communities. Meanwhile, a single FRP-tool blog post on `samfw.com` attracts 111 communities, and two redirect endpoints on `econventa.com` are accessed by 103 and 92 communities, respectively. The URL-stripped token produces more granular graph structure: a specific payload archive on Mediafire

(mediafire.com/file/.../Driver+TFT+TeaM.zip) becomes a bridge shared by 24 communities, while a host-only token would have merged it into a broad "everyone uses Mediafire" node.

Table 5.19 Top shared delivery infrastructure by the number of distinct communities that link into each host.

Host (url-stripped token)	Feeding communities
unlocktoolpro.com	119
samfw.com/blog/samfw-frp-tool-1-0-...	111
econventa.com/fp.engine	103
econventa.com/Redirect.eng	92
androidfilehost.com	74
samfw.com	56
file.unlocktool.net	52
mobiletechfrp.blogspot.com	41
xfinder.pro/q	32
rtbbhub.com/script/s2iurl.php	31
mediafire.com/file/.../Driver+TFT+TeaM.zip	24

5.4.4 Sequential Infection-Vector Patterns

Closed PrefixSpan mining, using a support threshold of 300 and a minimum pattern length of three, identified **460** unique patterns. The cross-host pattern that has the highest support is the unlock-tool delivery chain. The chain, with 1,267 support streams, is as follows: unlocktoolpro.com → unlocktoolpro.com/download.php → the specific Mediafire payload archive. Most of the longer patterns identified by the miner are not cross-host journeys; instead, they are single-service navigation sequences, predominantly involving clk.showmelinks.com short-link slugs and the new-tab.club hijack endpoint. These patterns repeat because a single traffic distribution service is used multiple times. This result shows the expected trade-off linked to the more detailed tokenization outlined in Section 3.5.7: although it improves the detection of shared-infrastructure indicators, it also leads to more within-service chains appearing in the pattern output. As a result, the shared-infrastructure table offers more reliable claims about indicators.

The family attribution is completely degenerate, clearly indicating a monoculture. All 460 patterns correspond to either Vidar or unlabeled cases, with only five patterns involving a minority family (MacSyncStealer, affecting 16 victims in total). The primary unlock-tool chain is linked to 762 Vidar victims and 285 unlabeled cases, and no other family meets the support threshold. Because of this, the family correlation cannot help distinguish between

different infection vectors in this dataset. As a result, we need to shift our analysis to the operator brand to gain clearer insights.

5.4.5 Operator-Brand Correlation

18 unique operator brands were recovered for 6,496 of 6,940 trajectory victims, representing 93.6%. The brand recovers structural distinctions that the family label cannot. Specifically, it divides the Vidar monoculture into separate operator panels, labels victims missed by the YARA scan (the `logactive` and `up_daisycloud` panels each contain hundreds of family-unlabeled logs), and isolates the non-Vidar tail, as every MacSyncStealer victim is associated with a macOS-specific brand. While the family distribution only suggested these divisions, the brand definitively resolves them.

A key question is whether the brand can distinguish between delivery vectors where the family label cannot. To evaluate this, each victim is assigned to a community at $\gamma = 0.05$ based on the plurality of its trajectory hosts, and the brand is cross-tabulated against this assignment, which covers the 6,666 victims that land in a community. On the Vidar side, the brand is nearly orthogonal to the vector. Each of the seven large Windows panels is distributed across 128 to 563 communities, with all peaking in the same community, the cracked-unlock-tool ecosystem C_0 , at concentrations between 15 and 30%. Similarly, C_0 is sourced by all four leading operators (`reg.vidars.su` 28%, `up_daisycloud` 19%, `logactive` 19%, `nexuadmin` 14%), with none exceeding 28%. No operator exclusively controls a vector, nor does any vector belong to a single operator. Therefore, the brand does not split the corpus into individual-operator vectors, reflecting the constraints of the family label. Instead, it follows an affiliate model [1, 5] where operators source installs from a common cracked-software network rather than using exclusive delivery methods.

The only instance where the brand aligns clearly with a distinct vector is in the macOS tail. All MacSyncStealer panels (`russia34.com`, `slurmcloudsupport`, `wichlovefromr`, `cloudhesoyam`) bypass C_0 entirely. Each panel is distributed across a separate set of communities, with no dominant ecosystem, peaking at only 2 to 5% on cloud drives, webmail, and various SaaS or portal hosts, rather than the Windows cracked-software sites. The macOS operators utilize a distinct delivery surface, corresponding to the same boundary identified by the family axis and representing the only operator-to-vector division supported by the corpus. The macOS operators' only point of contact with the Windows side is cracked software community C_{24} , which is the only one that merges Vidar and MacSyncStealer's populations, making `russia34.com` appear here.

By examining the assignment by operator composition within each community rather than focusing on the top community for each operator, we confirm orthogonality and identify one exception (Figure 7). The firmware, traffic-distribution, movie-piracy, search-hijacker, and tutorial communities (C_2 , C_5 , C_8 , C_{10} , C_{19}) all exhibit the same operator mix as C_0 , indicating that while the lure theme varies, the operator composition remains constant. The exception is C_1 , the `href.li` cloaker community, which is predominantly operated by two entities:

up_daisycloud accounts for 55% of its victims and valenciga for another 17%, compared to their respective 20% and 3% shares in C_0 . Additionally, 76% of the community is YARA-unlabeled. This cell functions as a clear operator-by-vector indicator, illustrating the intended difference between the two axes: a cloaked delivery channel linked to two specific operators, which cannot be identified only through the family label or the community map. Therefore, the two axes carry different information: the brand indicates the operator and labels unlabeled samples, while the community describes the ecosystem. These axes converge into a single operator-specific vector only within the C_1 cloaker channel and the C_{24} cracked software channel.

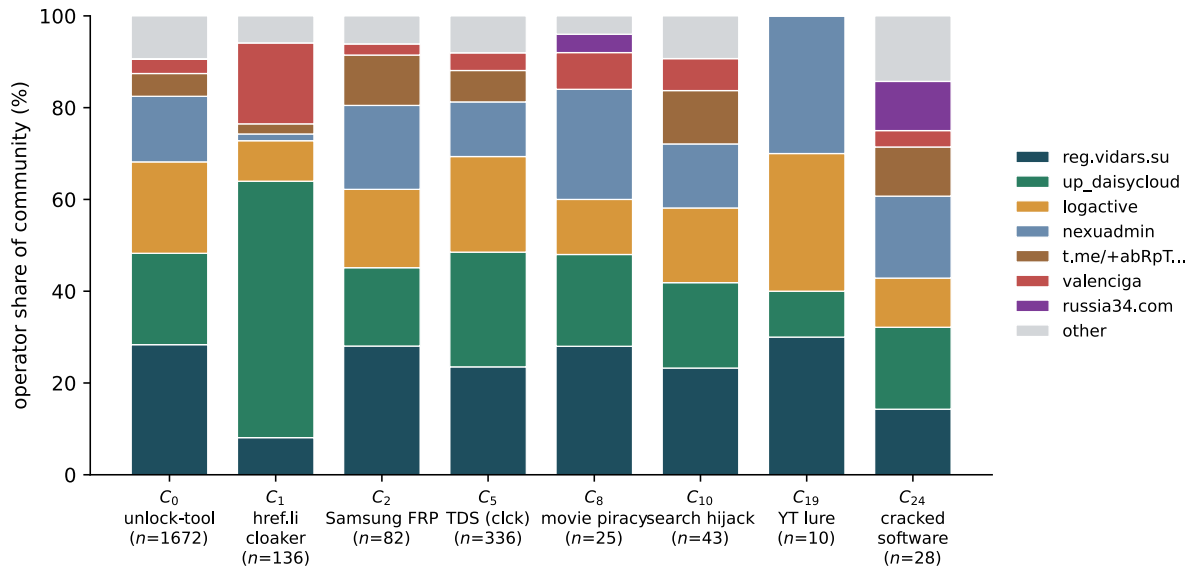


Figure 7: Operator-brand composition of each curated community.

5.4.6 Direct Lure Recovery: the Last-Non-Sink Hop

The last-non-sink instrument of Section 3.6.8 recovers the individual lures that the count-thresholded graph cannot see. This analysis produced **3,024** distinct last-non-sink domains out of 6,940 victims, showing that a few lure domains account for most victims, while most domains were used against only one. Of the 3,024 domains, **959** (32%) were the inferred lure for a single victim, while the top 10 domains cover 25% of victims, the top 50 cover 38%, and the top 250 cover 55%. Figure 8 shows the full cumulative curve, while Table 5.20 provides the overview with a role label.

The ranking reveals three key insights that could not be uncovered by the community view alone. Firstly, a single Amazon S3 bucket, `makfile.s3.us-east-1.amazonaws.com`, functions as the final home for 693 victims, serving as a significant high-value payload sink. Second, YouTube ranks second as the last hop with 880 victims, and analysis of the channels indicates that the 1,010 YouTube-terminal histories are primarily linked to specific tutorial channels (@souravjvlogs, @sunilpandaofficial, @seyhayt4692, @BeeBuYog). Lastly, the off-projection tail surfaces concrete cloaked-link indicators that fall below the community floor, including

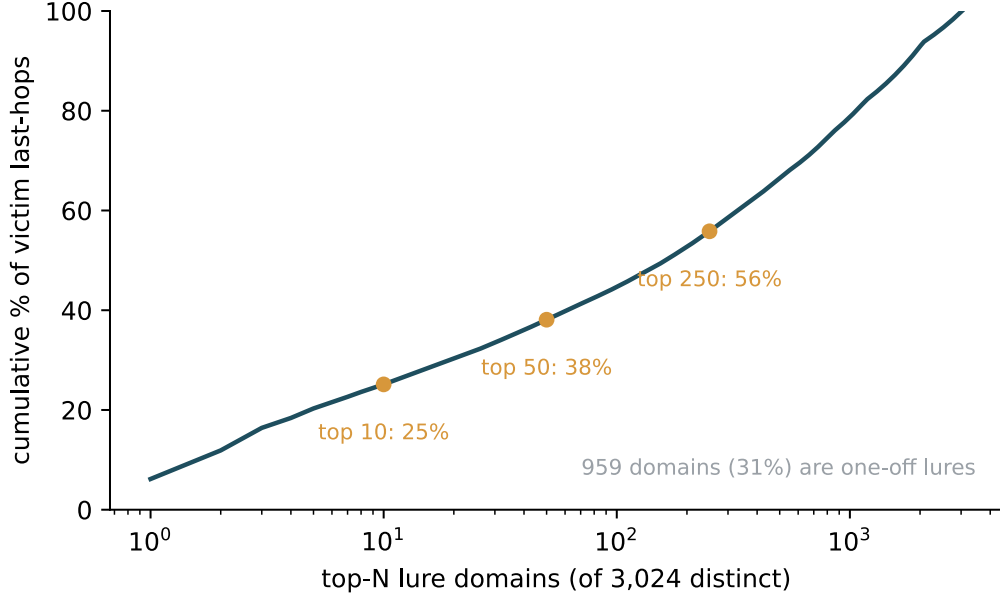


Figure 8: Cumulative share of victim last-hops covered by the top last-non-sink lure domains.

`linktr.ee/WLFRcloud` (44 victims), `usrlnk.io/Hesoyam` (28), and `usrlnk.io/SlurmProject` (28).

5.4.7 Method Convergence

The three victim-to-community assignment methods described in Section 3.6 produce convergent results. Table 5.21 reports the numbers. All three methods place about 25 to 28% of victims in the dominant cracked-software community C_0 , and under every method, the most common install name within the community, `koc0dj3.exe`, accounts for 23 to 25% of the victims in C_0 . Sink-stripping has minimal impact on these proportions because, within this corpus, both the lure brands and download sinks are located in the same community: C_0 includes both `unlocktoolpro.com` and the generic file hosts. It follows that removing a sink simply relocates the landing host within the same group. The consistent results across three different voting token definitions indicate that the dominant-vector property is a core feature of the data itself, rather than just a result of any specific assignment rule.

5.4.8 Structural Bias and Threats to Validity

The results are subject to the limitations outlined in Section 3.6.10, now supported by empirical data. The two most significant limitations are: 94% of family-labeled trajectory victims are associated with Vidar, and only a fraction (22.1%) of the ingested stealer logs contain analyzable history. For instance, in this corpus, the SHubStealer and StealC families do not collect browser history and therefore contribute zero trajectories, despite collectively labeling 2,457 logs. It follows that all reported communities, patterns, and lures are associated with the Vidar victim ecosystem, and the question of whether different families share a delivery vector cannot be addressed beyond the

Table 5.20 Head of the last-non-sink lure ranking, by distinct victims, with a manual role label.

Last-non-sink host	Victims	Role
unlocktoolpro.com	945	lure: cracked unlock-tool
youtube.com	880	lure: tutorial video
makfile.s3.us-east-1.amazonaws.com	693	sink: S3 payload bucket
samfw.com	305	lure: firmware / FRP
new-tab.club	292	search hijacker
go.cyberslinks.com	193	cloaker / TDS
find-it.pro	165	search hijacker
ingraiv.cyou	150	cloaker
mobiletechfrp.blogspot.com	121	lure: firmware
file.unlocktool.net	112	lure: cracked unlock-tool
internetdownloadmanager.com	91	lure: cracked software

Table 5.21 Three victim-assignment methods converge on the same dominant community and the same dominant build.

Assignment method	Victims	C_0 share	koc0dj3.exe $\rightarrow C_0$
Plurality across hosts	6,666	25 %	23 %
Terminal host	5,499	27 %	25 %
Last non-sink hop	5,459	28 %	25 %

limited MacSyncStealer data. The operator-brand analysis partly addresses this issue by splitting the dominant family into operator panels, yet it still lacks a balanced cross-family sample.

The remaining limitations are less impactful. The installer filename acts as a randomized build tag instead of a descriptive label, which makes per-community dropper validation less helpful and is why this path was not continued. (Section 5.3.2). Furthermore, because the trajectories are based on browsing histories, the final significant host before infection indicates a possible vector but isn't definitive evidence, as the pipeline lacks timestamps or referrer data needed to confirm causality. Additionally, community membership relies on the resolution parameter, requiring the reporting of the entire sweep instead of just one value. The sink strip set is a manual heuristic, and some generic hosts are not filtered out; however, this is advantageous in this context because their tenant paths (such as the S3 bucket, `usrlnk.io`, and `linktr.ee` slugs) serve as indicators. Finally, victims are deduplicated based on the byte-exact hash of their inner archive. This means that logs re-shared and recompressed during transit might be counted more than once, which could cause the victim totals to be inflated to an unknown degree.

6 Conclusions and Future Works

This thesis evaluates the Telegram-based commodity infostealer ecosystem from two perspectives: the operator side, through binaries published on public malware repositories, and the victim side, through stealer logs circulating on Telegram.

6.1 Summary of Contributions

On the operator side, a reproducible host-only environment extracts Telegram IoCs from 173 of the analyzed samples. The overall recovery rate of 5.7 % masks the extractor's effectiveness on its intended target. Many structural properties of the corpus and the malware landscape are adversarial against recovery. Despite its recognized Telegram exfiltration capabilities, Telegram is only one of several available channels; highly sophisticated sandbox evasion can prevent IoC-bearing code paths from ever executing; and no authoritative documentation for Bot Token formats exists, making extraction dependent on heuristics reverse-engineered from observed samples. Restricting the rate to samples that exhibit any Telegram signals raises it to 40.3 %. The 187 recovered endpoints correspond to 110 distinct bot tokens, which represent the operator infrastructure observed in the corpus. Besides the quantitative results, two additional findings from this analysis extend beyond the corpus. First, static and dynamic recovery are intertwined: no endpoint can be extracted from static scanning of the distributed binary alone, but every indicator attributed to string matching comes from a payload dropped at runtime. Second, only a small subset of endpoints appear in both detonation passes, demonstrating they are complementary rather than redundant. Furthermore, the uninstrumented pass alone, not surprisingly, provides the largest contribution by recovering infrastructure that sandbox-aware samples suppress when monitoring is detected. Beyond extraction mechanics, the documented distinction between Telegram-native and Telegram-capable families is descriptive rather than predictive: extraction yield does not vary systematically between the two classes. Nevertheless, the recovered tokens cluster into shared-infrastructure operator groups, and a read-only live query confirms that 52.7 % remain active months after collection. Overall, it demonstrates that the extraction produces actionable indicators rather than mere historical artifacts.

On the victim side, browsing trajectories can be recovered for 22.1 % of the 31,388 ingested logs. The generated host-transition graph is dominated by a single component that aggregates unrelated campaigns through shared delivery infrastructure, but community detection divides this giant component and separates the different trajectory populations into recognizable lure ecosystems. The largest delivery infrastructure community covers unlock tools and the firmware economy, while the remaining communities include link cloakers, traffic-distribution endpoints, and tutorial-channel lures. A direct last-hop measure then captures the one-off lures that fall below the community size floor. Since family attribution is dominated by a single family, Vidar, the analysis instead categorizes operators by brand. The relationship between the operator and the delivery vector reveals that large operator panels converge on the same dominant ecosystem, which recurs through an affiliate-sourcing model in which operators draw installs

from a shared delivery substrate.

6.2 Linking the Two Modalities

Beyond the questions posed in this thesis, the original objective of joining these two sides into a single chain was to link a binary sample to its operator, and that operator to the delivery vector that produced its victims. This aim was not achieved due to structural constraints. In particular, the operator-side corpus is built around families that exploit Telegram directly from the binary, while panel-based only stealers are excluded by design. In contrast, the victim population obtained through public Telegram channels is dominated by a single panel-based family. As a result, the two corpora describe separate segments of the same ecosystem that share neither a family nor a common identifier linking them.

6.3 Limitations

The main limitation of these findings lies in how the data was collected. The operator-side corpus was detonated under a recency-first selection that covers 38.7 % of all admitted samples; therefore, the per-family rates represent a minimum estimate of recall, and multiple Telegram-exfiltrating families remain underrepresented. In addition, the victim-side analysis sits in a window of only a few days with a single log provider, so the monoculture of the trajectory population may reflect that single source, and the window is too narrow to prove whether the operator-to-vector sharing remains stable across the rapid campaign changes characteristic of this economy.

6.4 Future Works

The most important extension of this work follows directly from this structural gap. The addition of panel-based stealers to the operator-side corpus would place the same families on both sides. The remaining challenge would be to find the key that would enable binding a binary-side operator to a victim-side vector. A wider log window would test whether the trajectory analysis holds across a wider family spectrum and shared delivery infrastructure, while detonating the full corpus would push the undersampled families closer to their real recall. Each of these is a small step that, joined together, collectively leads from two distinct ecosystem halves to the unified intelligence view that inspired this work.

Bibliography

- [1] Constantinos Patsakis, David Arroyo, and Fran Casino. The Malware as a Service Ecosystem. In Dimitris Gritzalis, Kim-Kwang Raymond Choo, and Constantinos Patsakis, editors, *Malware: Handbook of Prevention and Detection*, pages 371–394. Springer Nature Switzerland, Cham, 2025. ISBN 978-3-031-66245-4. doi: 10.1007/978-3-031-66245-4_16. URL https://doi.org/10.1007/978-3-031-66245-4_16.
- [2] Sayak Saha Roy, Elham Pourabbas Vafa, Kobra Khanmohammadi, and Shirin Nilizadeh. DarkGram: A Large-Scale Analysis of Cybercriminal Activity Channels on Telegram. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, 2025. doi: 10.48550/arXiv.2409.14596. URL <http://arxiv.org/abs/2409.14596>. arXiv:2409.14596 [cs].
- [3] BitSight. Exfiltration over Telegram Bots: Skidding Infostealer Logs, October 2024. URL <https://www.bitsight.com/blog/exfiltration-over-telegram-bots-skidding-infostealer-logs>.
- [4] Athanasios Avgetidis, Omar Alrawi, Kevin Valakuzhy, Charles Lever, Paul Burbage, Angelos D. Keromytis, Fabian Monroe, and Manos Antonakakis. Beyond The Gates: An Empirical Analysis of HTTP-Managed Password Stealers and Operators. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5307–5324, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/avgetidis>.
- [5] Quentin Bourgue and Livia Tibirna. Traffers: a deep dive into the information stealer ecosystem, August 2022. URL <https://blog.sekoia.io/traffers-a-deep-dive-into-the-information-stealer-ecosystem/>.
- [6] KELA Cyber Team. Understanding the Infostealer Epidemic in 2025: Highlights from KELA’s 2025 Report, May 2025. URL <https://www.kelacyber.com/blog/understanding-the-infostealer-epidemic/>.
- [7] Kahng An. Weaponizing Telegram Bots: How Threat Actors Exfiltrate Credentials, March 2026. URL <https://cofense.com/blog/weaponizing-telegram-bots-how-threat-actors-exfiltrate-credentials>.
- [8] Bhisham Bhanot, Vijay Kumar, and Sandeep Sarowa. Hybrid Analysis of Agent-Tesla: A Prominent Malware as a Service. In *2025 3rd International Conference on Device Intelligence, Computing and Communication Technologies (DICCT)*, pages 17–22, March 2025. doi: 10.1109/DICCT64131.2025.10986671. URL <https://ieeexplore.ieee.org/document/10986671/>.
- [9] Pierre Le Bourhis, Quentin Bourgue, Pierre Le Bourhis, and Quentin Bourgue. Raccoon

- Stealer v2 - Part 2: In-depth analysis, June 2022. URL <https://blog.sekoia.io/raccoon-stealer-v2-part-2-in-depth-analysis/>.
- [10] ESET Research. Lumma Stealer: A fast-growing infostealer threat, January 2025. URL <https://www.eset.com/blog/en/business-topics/threat-landscape/lumma-stealer-threat/>.
- [11] ASEC. Infostealer Logs Analysis Report, November 2024. URL <https://asec.ahnlab.com/en/84967/>.
- [12] Michele Campobasso and Luca Allodi. Impersonation-as-a-Service: Characterizing the Emerging Criminal Infrastructure for User Impersonation at Scale. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1665–1680, New York, NY, USA, November 2020. Association for Computing Machinery. ISBN 978-1-4503-7089-9. doi: 10.1145/3372297.3417892. URL <https://dl.acm.org/doi/10.1145/3372297.3417892>.
- [13] Michele Campobasso and Luca Allodi. Know Your Cybercriminal: Evaluating Attacker Preferences by Measuring Profile Sales on an Active, Leading Criminal Market for User Impersonation at Scale. pages 553–570. USENIX Association, 2023. ISBN 978-1-939133-37-3. URL <https://www.usenix.org/conference/usenixsecurity23/presentation/campobasso>.
- [14] Juha Nurmi, Mikko Niemelä, and Billy Bob Brumley. Malware Finances and Operations: a Data-Driven Study of the Value Chain for Infections and Compromised Access. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, ARES '23, pages 1–12, New York, NY, USA, August 2023. Association for Computing Machinery. ISBN 979-8-4007-0772-8. doi: 10.1145/3600160.3605047. URL <https://dl.acm.org/doi/10.1145/3600160.3605047>.
- [15] Kyle Soska and Nicolas Christin. Measuring the Longitudinal Evolution of the Online Anonymous Marketplace Ecosystem. pages 33–48. USENIX Association, 2015. ISBN 978-1-939133-11-3. URL <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/soska>.
- [16] Rolf van Wegberg, Samaneh Tajalizadehkhoob, Kyle Soska, Ugur Akyazi, Carlos Hernandez Ganan, Bram Klievink, Nicolas Christin, and Michel van Eeten. Plug and Prey? Measuring the Commoditization of Cybercrime via Online Anonymous Markets. pages 1009–1026. USENIX Association, 2018. ISBN 978-1-939133-04-5. URL <https://www.usenix.org/conference/usenixsecurity18/presentation/van-wegberg>.
- [17] Quentin Bourgue. Overview of the Russian-speaking infostealer ecosystem: the logs, May 2023. URL <https://blog.sekoia.io/overview-of-the-russian-speaking-infostealer-ecosystem-the-logs/>.

- [18] Counter Threat Unit Research Team. The Growing Threat from Infostealers, May 2023. URL <https://www.sophos.com/research/the-growing-threat-from-infostealers>.
- [19] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, Daniel Margolis, Vern Paxson, and Elie Bursztein. Data Breaches, Phishing, or Malware? Understanding the Risks of Stolen Credentials. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pages 1421–1434, New York, NY, USA, October 2017. Association for Computing Machinery. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134067. URL <https://dl.acm.org/doi/10.1145/3133956.3134067>.
- [20] Jeremiah Onaolapo, Enrico Mariconti, and Gianluca Stringhini. What Happens After You Are Pwnd: Understanding the Use of Leaked Webmail Credentials in the Wild. In *Proceedings of the 2016 Internet Measurement Conference, IMC '16*, pages 65–79, New York, NY, USA, November 2016. Association for Computing Machinery. ISBN 978-1-4503-4526-2. doi: 10.1145/2987443.2987475. URL <https://dl.acm.org/doi/10.1145/2987443.2987475>.
- [21] Dario Adriano Bermudez Villalva, Jeremiah Onaolapo, Gianluca Stringhini, and Mirco Musolesi. Under and over the surface: a comparison of the use of leaked account credentials in the Dark and Surface Web. *Crime Science*, 7(1):17, November 2018. ISSN 2193-7680. doi: 10.1186/s40163-018-0092-6. URL <https://doi.org/10.1186/s40163-018-0092-6>.
- [22] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning More about the Underground Economy: A Case-Study of Keyloggers and Dropzones. In Michael Backes and Peng Ning, editors, *Computer Security – ESORICS 2009*, pages 1–18, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-04444-1. doi: 10.1007/978-3-642-04444-1_1.
- [23] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monroe. All Your iFRAMEs Point to Us. In *Proceedings of the 17th USENIX Security Symposium (Security '08)*. USENIX Association, 2008.
- [24] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Shady paths: leveraging surfing crowds to detect malicious web pages. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, CCS '13*, pages 133–144, New York, NY, USA, November 2013. Association for Computing Machinery. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516682. URL <https://dl.acm.org/doi/10.1145/2508859.2516682>.
- [25] Teryl Taylor, Xin Hu, Ting Wang, Jiyong Jang, Marc Ph Stoecklin, Fabian Monroe, and Reiner Sailer. Detecting Malicious Exploit Kits using Tree-based Similarity Searches. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, CODASPY '16*, pages 255–266, New York, NY, USA, March 2016.

- Association for Computing Machinery. ISBN 978-1-4503-3935-3. doi: 10.1145/2857705.2857718. URL <https://dl.acm.org/doi/10.1145/2857705.2857718>.
- [26] Arindam Banerjee and Joydeep Ghosh. Clickstream Clustering using Weighted Longest Common Subsequences. In *Proceedings of the Workshop on Web Mining, 1st SIAM Conference on Data Mining*, pages 33–40, August 2001. URL https://www.researchgate.net/profile/Joydeep-Ghosh-8/publication/2365525_Clickstream_Clustering_using_Weighted_Longest_Common_Subsequences/links/004635231ccb57760d000000/Clickstream-Clustering-using-Weighted-Longest-Common-Subsequences.pdf.
- [27] Jian Pei, Jiawei Han, B. Mortazavi-Asl, H. Pinto, Qiming Chen, U. Dayal, and Mei-Chun Hsu. PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings 17th International Conference on Data Engineering*, pages 215–224, Heidelberg, Germany, 2001. IEEE Comput. Soc. ISBN 978-0-7695-1001-9. doi: 10.1109/ICDE.2001.914830. URL <http://ieeexplore.ieee.org/document/914830/>.
- [28] Hugo Bijmans, Tim Booij, Anneke Schwedersky, Aria Nedgabat, and Rolf van Wegberg. Catching Phishers By Their Bait: Investigating the Dutch Phishing Landscape through Phishing Kit Detection. pages 3757–3774. USENIX Association, 2021. ISBN 978-1-939133-24-3. URL <https://www.usenix.org/conference/usenixsecurity21/presentation/bijmans>.
- [29] Taisiia Garkava, Asier Moneva, and E. Rutger Leukfeldt. Stolen data markets on Telegram: a crime script analysis and situational crime prevention measures. *Trends in Organized Crime*, April 2024. ISSN 1936-4830. doi: 10.1007/s12117-024-09532-6. URL <https://doi.org/10.1007/s12117-024-09532-6>.
- [30] Tina Marjanov, Taro Tsuchiya, Konstantinos Ioannidis, Jack Hughes, Nicolas Christin, and Alice Hutchings. Stayin’ Alive: How Global Stolen Data Markets Thrive on Telegram. USENIX Association, January 2026. doi: 10.17863/CAM.124824. URL <https://www.repository.cam.ac.uk/handle/1810/395272>.
- [31] Luca Allodi, Roy Ricaldi, Jai Wientjes, and Adriana Radu. Where is Dmitry going? Framing ‘migratory’ decisions in the criminal underground, November 2024. URL <http://arxiv.org/abs/2411.16291>. arXiv:2411.16291 [cs].
- [32] Cisco Talos. Operation Blacksmith: Lazarus targets organizations worldwide using novel Telegram-based malware written in DLang, December 2023. URL https://blog.talosintelligence.com/lazarus_new_rats_dlang_and_telegram/.
- [33] Hive pro. Iranian MOIS Leverages Telegram-Based C2 in Espionage Campaign Targeting Dissidents | Hive Pro, March 2026. URL <https://hivepro.com/threat-advisory/iranian-mois-leverages-telegram-based-c2-in-espionage-campaign-targeting-dissidents/>.

- [34] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *2012 IEEE Symposium on Security and Privacy*, pages 65–79, May 2012. doi: 10.1109/SP.2012.14. URL <https://ieeexplore.ieee.org/abstract/document/6234405>. ISSN: 2375-1207.
- [35] Daniel Plohmann, Martin Clauß, Steffen Enders, and Elmar Padilla. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. *Journal of Computer Virology and Hacking Techniques*, 2017. doi: 10.18464/cybin.v3i1.17. URL https://www.researchgate.net/publication/354701352_Malpedia_A_Collaborative_Effort_to_Inventorize_the_Malware_Landscape.
- [36] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. A Comprehensive Measurement Study of Domain Generating Malware. pages 263–278. USENIX Association, 2016. ISBN 978-1-931971-32-4. URL <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/plohmann>.
- [37] Chaz Lever, Platon Kotzias, Davide Balzarotti, Juan Caballero, and Manos Antonakakis. A Lustrum of Malware Network Communication: Evolution and Insights. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 788–804, May 2017. doi: 10.1109/SP.2017.59. URL <https://ieeexplore.ieee.org/abstract/document/7958610>. ISSN: 2375-1207.
- [38] Palo Alto Networks. Complete Guide to Understanding Indicators of Compromise (IoCs). URL <https://www.paloaltonetworks.com/cyberpedia/indicators-of-compromise-iocs>.
- [39] Xiaojing Liao, View Profile, Kan Yuan, View Profile, XiaoFeng Wang, View Profile, Zhou Li, View Profile, Luyi Xing, View Profile, Raheem Beyah, and View Profile. Acing the IOC Game. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM Conferences, pages 755–766. October 2016. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978315. URL <https://dl.acm.org/doi/10.1145/2976749.2978315>.
- [40] Juan Caballero, Gibran Gomez, Srdjan Matic, Gustavo Sánchez, Silvia Sebastián, and Arturo Villacañas. The Rise of GoodFATR: A Novel Accuracy Comparison Methodology for Indicator Extraction Tools. *Future Generation Computer Systems*, 144:74–89, July 2023. ISSN 0167-739X. doi: 10.1016/j.future.2023.02.012. URL <https://www.sciencedirect.com/science/article/pii/S0167739X23000535>.
- [41] U.S. DoD Defence Cyber Crime Center. DC3-MWCP, April 2026. URL <https://github.com/dod-cyber-crime-center/DC3-MWCP>. original-date: 2015-05-06T15:11:27Z.
- [42] CERT Polska. Karton, April 2026. URL <https://github.com/CERT-Polska/karton>. original-date: 2020-05-26T16:29:58Z.

- [43] Kevin O'Reilly and Andriy Brukhovetsky. CAPE: Malware Configuration And Payload Extraction, June 2026. URL <https://github.com/kevoreilly/CAPEv2>. original-date: 2019-10-15T18:16:09Z.
- [44] Oleg Boyarchuk and Jason Zhang. Emotet C2 Configuration Extraction and Analysis, March 2022. URL <https://blogs.vmware.com/security/2022/03/emotet-c2-configuration-extraction-and-analysis.html>.
- [45] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Comput. Surv.*, 52(5):88:1–88:48, September 2019. ISSN 0360-0300. doi: 10.1145/3329786. URL <https://dl.acm.org/doi/10.1145/3329786>.
- [46] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security & Privacy*, 5(2):32–39, March 2007. ISSN 1558-4046. doi: 10.1109/MSP.2007.45. URL <https://ieeexplore.ieee.org/document/4140988/>.
- [47] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, May 2005. doi: 10.1109/SP.2005.20. URL <https://ieeexplore.ieee.org/document/1425057/>. ISSN: 2375-1207.
- [48] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, December 2007. doi: 10.1109/ACSAC.2007.21. URL <https://ieeexplore.ieee.org/abstract/document/4413008>. ISSN: 1063-9527.
- [49] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2):1–42, February 2012. ISSN 0360-0300, 1557-7341. doi: 10.1145/2089125.2089126. URL <https://dl.acm.org/doi/10.1145/2089125.2089126>.
- [50] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM Comput. Surv.*, 52(6):126:1–126:28, November 2019. ISSN 0360-0300. doi: 10.1145/3365001. URL <https://dl.acm.org/doi/10.1145/3365001>.
- [51] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1009–1024, May 2017. doi: 10.1109/SP.2017.42. URL <https://ieeexplore.ieee.org/abstract/document/7958622>. ISSN: 2375-1207.
- [52] Sukant Kumar. Telegram Bot API Abuse, April 2026. URL https://netlas.io/blog/abuse_of_telegram_bot_api/.

- [53] Estelle Ruellan, Eric Clay, and Nicholas Ascoli. LLM-Based Identification of Infostealer Infection Vectors from Screenshots: The Case of Aurora, July 2025. URL <http://arxiv.org/abs/2507.23611>. arXiv:2507.23611 [cs.CR].
- [54] VirusTotal, . URL <https://www.virustotal.com/gui/home/upload>.
- [55] Free Automated Malware Analysis Service - powered by Falcon Sandbox, . URL <https://hybrid-analysis.com/>.
- [56] MalwareBazaar, . URL <https://bazaar.abuse.ch/browse/>.
- [57] VirusShare, . URL <https://virusshare.com/>.
- [58] MalShare, . URL <https://www.malshare.com/>.
- [59] Sean Gallagher and Markel Picado. Agent Tesla amps up information stealing attacks, February 2021. URL <https://www.sophos.com/blog/agent-tesla-amps-up-information-stealing-attacks>.
- [60] Patrick Schläpfer. Snake Keylogger's Many Skins: Analysing Code Reuse Among Infostealers, June 2021. URL <https://threatresearch.ext.hp.com/the-many-skins-of-snake-keylogger/>.
- [61] Kevin Su. FortiSandbox 5.0 Detects Evolving Snake Keylogger Variant | FortiGuard Labs, February 2025. URL <https://www.fortinet.com/blog/threat-research/fortisandbox-detects-evolving-snake-keylogger-variant>. Section: Threat Research.
- [62] Blank-c. Blank-Grabber, April 2026. URL <https://github.com/Blank-c/Blank-Grabber>. original-date: 2022-01-26T12:04:56Z.
- [63] Omer Hofman. Turning Telegram toxic: 'ToxicEye' RAT is the latest to use Telegram for command & control, April 2021. URL <https://blog.checkpoint.com/security/turning-telegram-toxic-new-toxiceye-rat-is-the-latest-to-use-telegram-for-command-contr>
- [64] LimerBoy. StormKitty: Open source stealer written on C#, all logs will be sent to Telegram bot., April 2023. URL <https://web.archive.org/web/20230403114012/https://github.com/LimerBoy/StormKitty>.
- [65] Atinderpal Singh and Brett Stone-Gross. Prynt Stealer's Backdoor Exposed, September 2022. URL <https://www.zscaler.com/blogs/security-research/no-honor-among-thieves-prynt-stealer-s-backdoor-exposed>.
- [66] nullixx. Phemedrone-Stealer: OpenSource Stealer, January 2024. URL <https://web.archive.org/web/20240118013358/https://github.com/nullixx/Phemedrone-Stealer>.

- [67] Splunk Threat Research Team and Teoderick Contreras. Unveiling Phemedrone Stealer: Threat Analysis and Detections, February 2024. URL https://www.splunk.com/en_us/blog/security/unveiling-phemedrone-stealer-threat-analysis-and-detections.html.
- [68] Vikas Kundu. No Honour Among Thieves: Uncovering a Trojanized XWorm RAT Builder Propagated by Threat Actors and Disrupting Its Operations, January 2025. URL [https://www.cloudsek.com/blog/no-honour-among-thieves-uncovering-a-trojanized-xworm-rat-builder-propagated-by-threat-](https://www.cloudsek.com/blog/no-honour-among-thieves-uncovering-a-trojanized-xworm-rat-builder-propagated-by-threat-actors-and-disrupting-its-operations)
- [69] Efstratios Lontzetidis. The Detection & Response Chronicles: Exploring Telegram Abuse, December 2025. URL <https://blog.nviso.eu/2025/12/16/the-detection-response-chronicles-exploring-telegram-abuse/>.
- [70] Ox6rss. The Close Relationship Between Telegram Bots and Threat Actors: New Stealers, Hunters Becoming the Hunted, March 2026. URL <https://cti.monster/blog/2026/03/24/new-stealers.html>.
- [71] Blank-c. Umbral-Stealer, April 2026. URL <https://github.com/Blank-c/Umbral-Stealer>. original-date: 2023-03-24T04:22:05Z.
- [72] V. A. Traag, L. Waltman, and N. J. van Eck. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1):5233, March 2019. ISSN 2045-2322. doi: 10.1038/s41598-019-41695-z. URL <https://www.nature.com/articles/s41598-019-41695-z>.

Appendix I - YARA Rules

This appendix presents two sets of YARA rules developed for this study. The first set supports the Telegram indicator taxonomy described in Section 3.4.5. The second set extends the stealer-family identification methodology described in Section 3.5.5 by incorporating two additional families not covered by the vendored rule set. Each rule provides the relevant strings and condition matching logic. Non-functional metadata has been abbreviated for clarity, and the complete files are distributed with the code.

I.1 Telegram Indicator Taxonomy Rules

The first file holds the two rules that separate a specific Bot API reference from a generic Telegram one. A hit on `tg_bot_api_specific` with no recovered token is the API-tier signal of the diagnostic ladder; a hit on `tg_generic` alone is one of the weaker reference signals.

Listing 6.2: Telegram Bot API taxonomy rules (`telegram_bot_api.yar`).

```
rule tg_bot_api_specific
{
    meta:
        description = "Specific Telegram Bot API evidence (token regex or /bot path)"
        author      = "NickP3lle"

    strings:
        // Bot token: numeric bot ID, colon, exactly 35 [A-Za-z0-9_-] chars.
        $bot_token_re = /\b[0-9]{8,16}:[A-Za-z0-9_-]{35}\b/
        // Canonical /bot path (nocase covers obfuscated casing).
        $api_url_1    = "api.telegram.org/bot"          ascii wide nocase
        $api_url_2    = "https://api.telegram.org/bot"  ascii wide
        $api_url_3    = "http://api.telegram.org/bot"   ascii wide
        // Base64 of the API /bot path, matched mid-stream (no padding).
        $b64_bot_path = "YXBpLnRlbGVncmFtLm9yZy9ib3Q"  ascii
        $b64_https_bot = "aHR0cHM6Ly9hcGkudGVsZWdyYW0ub3JnL2JvdA" ascii

    condition:
        filesize < 50MB
        and ( $bot_token_re or any of ($api_url_*) or any of ($b64_*) )
}

rule tg_generic
{
    meta:
```

```

description = "Generic Telegram references without specific Bot API
               evidence"
author      = "NickP3lle"
strings:
    // Direct Telegram domain / link references.
    $generic_tme      = "t.me/"          ascii wide nocase
    $generic_tg_org    = "telegram.org"  ascii wide nocase
    $generic_tg_bot    = "Telegram Bot"  ascii wide nocase
    // Base64 of the bare API host.
    $generic_b64_host  = "YXBpLnRlbGVncmFtLm9yZw"          ascii
    $generic_b64_https = "aHR0cHM6Ly9hcGkudGVsZWdyYW0ub3Jn" ascii
    // Bot API method names (2+ required to gate alone).
    $method_1 = "/sendDocument"  ascii wide
    $method_2 = "/sendMessage"   ascii wide
    $method_3 = "/sendPhoto"     ascii wide
    $method_4 = "/getMe"         ascii wide
    $method_5 = "/getUpdates"    ascii wide
    $method_6 = "/sendAnimation" ascii wide
    $method_7 = "/sendAudio"     ascii wide
    // Config field-name fragments (2+ required to gate alone).
    $ident_1 = "TelegramToken"    ascii wide nocase
    $ident_2 = "TelegramBotToken" ascii wide nocase
    $ident_3 = "TelegramChatId"   ascii wide nocase
    $ident_4 = "TelegramChatID"   ascii wide nocase
    $ident_5 = "telegram_token"   ascii wide nocase
    $ident_6 = "telegram_chat"    ascii wide nocase
    $ident_7 = "tg_token"         ascii wide nocase
    $ident_8 = "tg_chat"          ascii wide nocase
    $ident_9 = "useTelegram"      ascii wide nocase
    $ident_10 = "SendToTelegram"  ascii wide nocase
    // Base64 of "telegram".
    $b64_telegram = "dGVsZWdyYW0" ascii
condition:
    filesize < 50MB
    and ( any of ($generic_*) or 2 of ($method_*)
          or 2 of ($ident_*) or $b64_telegram )
}

```

The second file contains the four detectors for weaker references in dropped payloads: supergroup identifiers, tg:// deep links, private invite links, and Pyrogram or Telethon user API credentials. A hit on any of them, with no token and no API evidence, is a reference-tier signal.

Listing 6.3: Telegram reference taxonomy rules (telegram_extras.yar).

```
rule tg_supergroup_id
{
    meta:
        description = "Telegram supergroup/channel ID (-100 + 10-13 digits)"
        author      = "NickP311e"

    strings:
        // No \b in YARA regex: require a non-digit byte on each side.
        $sg = /[~0-9]-100[0-9]{10,13}[~0-9]/

    condition:
        filesize < 50MB and $sg
}

rule tg_deep_link
{
    meta:
        description = "Telegram tg:// deep links (resolve / join / openmessage)"
        author      = "NickP311e"

    strings:
        $a = "tg://resolve"      ascii wide nocase
        $b = "tg://join"         ascii wide nocase
        $c = "tg://openmessage"  ascii wide nocase
        $d = "tg://msg"          ascii wide nocase

    condition:
        filesize < 50MB and any of them
}

rule tg_invite_link
{
    meta:
        description = "Telegram private invite links (t.me/+<hash> or  
t.me/joinchat/<hash>)"
        author      = "NickP311e"

    strings:
```

```

        $plus = /t\.me\/\+[A-Za-z0-9_-]{10,40}/          ascii nocase
        $jchat = /t\.me\/joinchat\/[A-Za-z0-9_-]{10,40}/  ascii nocase
    condition:
        filesize < 50MB and any of them
}

rule tg_user_api
{
    meta:
        description = "Pyrogram/Telethon user-API credential pair (api_id +
                        api_hash)"
        author      = "NickP311e"
    strings:
        $id = "api_id"      ascii wide nocase
        $hash = "api_hash"  ascii wide nocase
    condition:
        filesize < 50MB and all of them
}

```

I.2 Stealer-Family Identification Additions

These two rules extend the vendored *what-is-this-stealer* set with families that appeared in the corpus but were not covered upstream (Section 3.5.5). Each key is on the fixed strings that the family writes into its own system-information file, so the matching rule name is the family name.

Listing 6.4: Author-added stealer-family rules.

```

rule SHubStealer
{
    meta:
        author      = "NickP311e"
        description = "SHub Stealer system information file"
    strings:
        $x1 = "SHub Stealer"      ascii
        $x2 = "=== System Info ==="  ascii
        $x3 = "Build Tag:"        ascii
    condition:
        all of them
}

```

```
}

rule MacSyncStealer
{
    meta:
        author      = "NickP3ll3"
        description = "MacSync Stealer system information file (Information.txt)"
    strings:
        $x1 = "MacSync Stealer" ascii
        $x2 = "Build Tag:"      ascii
        $x3 = "Username:"       ascii
        $x4 = "Password:"       ascii
    condition:
        $x1 and $x2 and $x3 and $x4
}
```