Mobile Device Triage Toolkit: Deterministic, Read-only Forensic Pre-Assessment

Luis F. Jusino Alamo
Master in Computer Science
Advisor: Nelliud Torres Batista, DBA
Polytechnic University of Puerto Rico
Graduate Project EXPO, February 2026

Abstract *– Mobile investigations face evidence backlogs and limited time to decide whether a device merits full imaging. We present the Mobile Device Triage Toolkit, a read-only, deterministic workflow that inspects disk images (RAW and E01 via pytsk3/pyewf or E01 export) or backups/logical folders and summarizes high-value artifacts. MDTK provides a filesystem summary, app inventory , SQLite table counts, endpoint-pattern hits, and a mini-timeline, exporting both JSON and a uniform PDF. For forensic defensibility, MDTK records an append-only JSONL audit log, refuses writable mounts, pins the runtime environment, and fixes timestamps to UTC seconds; Ed25519 signing is supported. We evaluate MDTK on a manifest of sample images and report runtime, artifact coverage, and reproducibility by comparing JSON/PDF hashes across repeated runs. Results show byte-identical outputs, median execution under 8.33s and fast visibility into artifacts such as messaging and browser histories. MDTK targets triage escalation, not full analysis, and runs on Windows via WSL.*

Keywords *– Digital Forensics, Evidence Integrity, SQLite Artifacts, Triage.*

INTRODUCTION

Digital Forensics teams increasingly face “too many devices and too little time”. When dozens of mobile images arrive, the first question that arrives is not what happened, but which ones deserve full imaging now. Yet triage tools often trade speed for rigor, risking write operations, nondeterministic outputs, or ad-hoc procedures that weaken chain of custody. This paper introduces the Mobile Device Triage Toolkit, a read-only deterministic workflow for inspecting mobile device images in RAW and E01 formats using the EWF ecosystem [1] [2]. The toolkit summarizes the evidence through a filesystem summary, app inventory, SQLite table counts, endpoints hits, and a mini-timeline, all exported to JSON and PDF with append-only audit logs [3] [4].

Scope and Objective

This work targets mobile disk-image rather than full forensic analysis. The objective is to provide a fast, defensible snapshot that helps prioritize which devices should be fully imaged. We assume access to images in RAW or E01, operate in read-only mode, and avoid deep content parsing that could expand the evidentiary surface or introduce nondeterminism. The toolkit emphasizes structural signals (filesystem type and layout, application presence, SQLite table-counts, endpoint-pattern hits, and recent file activity) that are informative for early decision making under chain-of-custody constraints.

Contributions

This paper has four contributions:

- Adapter layer for RAW/E01 with write refusal checks, enabling inspection without mounting images read-write.
- Rule-driven extractors that produce a filesystem summary, app inventory, SQLite table counts, endpoint indicators, and a mini timeline suitable for triage [4].
- We implement a defense-in-depth zero-write layer: read-only handles (pytsk3/pyewf), host mount checks that fail fast on any writable signal, bounded read-only extractors, and an isolated workspace of the evidence [1] [2] [5]. Refusals include a machine-readable code and a plain-English reason.
- We enforce determinism + auditability: fixed-order directory walks with locale-independent collation, canonical JSON (stable

keys/separators), ordered merges for any concurrency, UTC-ms timestamps, and an append-only JSONL audit log (run ID, adapter, paths, bytes, warnings, start/stop).

Together, these contributions make MDTK a practical and defensible bridge between "I have an image" and "I know what's worth deeper analysis". By combining write-refusal guardrails with rule-driven artifact extraction, the toolkit delivers rapid, consistent triage outputs while preserving evidence integrity. Deterministic reporting and an audit log provide a transparent chain from input to output, supporting repeatable results across runs and environments and enabling investigators to justify prioritization decisions without performing full acquisition or invasive analysis.

BACKGROUND

Mobile investigations often face tight time constraints and large evidence backlogs, creating a need for rapid, defensible decisions about what data merits deeper analysis. This section distinguishes triage from full parsing suites and explains why mobile platforms add constraints beyond desktop workflows. These factors motivate MDTK's goals: minimal read-only access, lightweight presence/summary extraction, and outputs that support prioritization without replacing full analysis.

Triage vs. Full Parsing Suites; Mobile vs. Desktop Paradigms

Triage aims for defensible "first look": quickly answer scope and prioritization questions (is there anything here worth full image? Which artifacts exist? Roughly when were they active?) without heavy parsing. Full suites chase depth (complete decoding of app databases, cross-artifact correlation, analytics) at the cost time, dependencies, and potential side effects. Mobile adds extra friction versus desktop: sandboxed app data, multiple user/work profiles, encrypted stores, vendor partition layouts, and rapid OS churn. Desktop workflows often assume broader, more uniform file systems and fewer app silos. Our toolkit sits squarely in the mobile triage niche: minimal reads, presence/summary extractors, and strict read-only posture to inform whether deeper mobile parsers should be deployed.

Determinism/Reproducibility in Forensics

Forensic outputs should be reproducible: the same inputs and configuration should yield the same outputs, byte for byte. In practice, many tools are not deterministic due to unsorted directory walks, locale and timestamp formatting differences, concurrency effects, and library defaults. While this variability may be acceptable during exploratory investigation, it weakens defensibility in incident response, litigation, and repeatable lab workflows where results must be independently re-run and verified. Here, determinism means a fixed traversal order, consistent transformations, and stable emitted artifacts and logs for a given image and configuration. This is especially important in mobile triage, where examinations may be repeated months later under legal scrutiny. Because many tools prioritize artifact breadth and user-facing features over constrained execution, determinism remains uncommon as an explicit design goal, motivating triage frameworks that treat reproducibility as a first-class engineering requirement.

Existing Tools & Libraries: Strengths and Gaps for Zero-write, Auditable Triage

Mature libraries already support low-level forensic access to disk images and mobile data, including The Sleuth Kit (TSK) with pytsk3 for filesystem analysis [5] [6] and the EWF ecosystem (libewf/pyewf) for common container formats. In parallel, many mobile-specific parsers can extract artifacts from iOS/Android backups, application databases, message stores, and system logs.

However, these tools are typically building blocks or monolithic applications, not tightly controlled end-to-end triage pipelines with explicit guarantees about write behavior and audit structure. Many allow read-only operation but do not enforce

framework-level zero-write guardrails against side effects (e.g., writable mounts or temporary writes), and their logging is often ad hoc rather than a structured, machine-verifiable audit trail suitable for replay and independent verification. This motivates a triage toolkit that builds on proven libraries while adding strict zero-write enforcement and deterministic, structured auditability as first-class requirements.

THREAT MODEL & SAFETY REQUIREMENTS

This paper presents the Mobile Device Triage Toolkit (MDTK), designed for rapid, defensible triage when investigators face evidence backlogs and limited time to decide whether a device merits full imaging and deep analysis. MDTK assumes an honest-but-fallible analyst and a potentially misleading execution environment. The primary adversary in this model is not malicious examiner, but accidental evidence modification and irreproducible outcomes caused by operating system behaviors (e.g., auto-mounting media, background indexing, antivirus scanning, thumbnail generation, and "repair" prompts), dependency drift, or unstable parsers. MDTK targets triage decisions rather than full forensic reconstruction; therefore, safety controls prioritize evidence integrity, bounded execution, and transparent auditability over exhaustive decoding.

Assets

MDTK protects the evidence source, including source, including disk images (RAW, E01 via pyewf/pytsk3 or E01 export) and backups/logical folder collections, which must remain unmodified; the forensic workstation environment (including Windows with WSL support), since uncontrolled parsing and writes can contaminate results or leak sensitive data; and the outputs as evidentiary artifacts, including JSON summaries, the uniform PDF report, and the append-only JSONL audit log, which must be consistent, attributable to a specific run, and verifiable across repeats.

Risks

The primary risk is accidental read-write access, such as evidence presented via a writable mount (OS/WSL) or opened through write-capable interfaces. A second risk is nondeterminism: differences in traversal order, timestamp/locale handling, concurrency, or dependency versions producing different outputs from the same input. A third risk is parser instability on malformed or unexpected data (e.g., corrupted files or SQLite databases), which can cause crashes or partial extraction and lead to misleading triage summaries if not explicitly logged.

Safety Requirements

MDTK enforces the following requirements to mitigate these risks:

1) Read-only access by construction - Evidence is accessed through read-only paths; mount state (if used) is treated as a safety precondition.
2) Refusal on writable conditions (fail closed) - If writable access is detected or cannot be ruled out (e.g., rw flags or ambiguous state), MDTK aborts and logs a clear diagnostic.
3) Deterministic traversal and emission - Evidence is traversed in a fixed order and reports are emitted deterministically; repeated runs on identical inputs/config produce byte-identical JSON/PDF hashes.
4) Time normalization and environment pinning - Timestamps are normalized to UTC seconds, and the runtime environment (tool/dependency versions and configuration) is recorded to prevent drift.
5) Bounded reads and controlled failure - Resource limits are enforced; failures or limits result in explicit logged skips/warnings rather than crashes or silent omissions.
6) Exhaustive append-only auditability – MDTK emits a JSONL audit log capturing evidence identifiers, configuration, key decisions, extraction attempts, errors/skips, and output hashes.

7) Optional cryptographic provenance – Outputs and/or audit records can be Ed25519-signed to detect post-run modification.

These requirements operationalize triage defensibility: MDTK is designed to produce rapid summaries while preserving evidence integrity through read-only access, deterministic execution, and an auditable record of what was examined, what was skipped, and why.

SYSTEM DESIGN

MDTK is designed as a small, deterministic pipeline that separates evidence access, artifact extraction, and reporting/signing. This separation is intentional: it reduces the chance of accidental writes, enables strict data contracts between stages, and makes reproducibility testable by hashing well-defined outputs. The system processes either disk images (RAW or E01) or backup/logical folder sources and produces uniform JSON and PDF outputs accompanied by an append-only JSONL audit log.

Architecture Overview

At a high level, MDTK transforms a single evidence source into a fixed set of structured summaries through a staged pipeline:

- Evidence > Adapter > Extractor Pipeline > Reporters > Optional Signature

MDTK architecture: read-only adapters → deterministic extractors → JSON/PDF outputs, with append-only audit logging and optional Ed25519 signing.

To keep modules composable and auditable, MDTK defines explicit data contracts between stages using Pydantic schemas. Each extractor receives a typed "context" object (including run configuration, evidence identifiers, and adapter metadata) and returns a typed result object. This also provides a natural place to enforce "no surprises" constraints, such as maximum sizes, normalized timestamps (UTC seconds), and stable ordering.

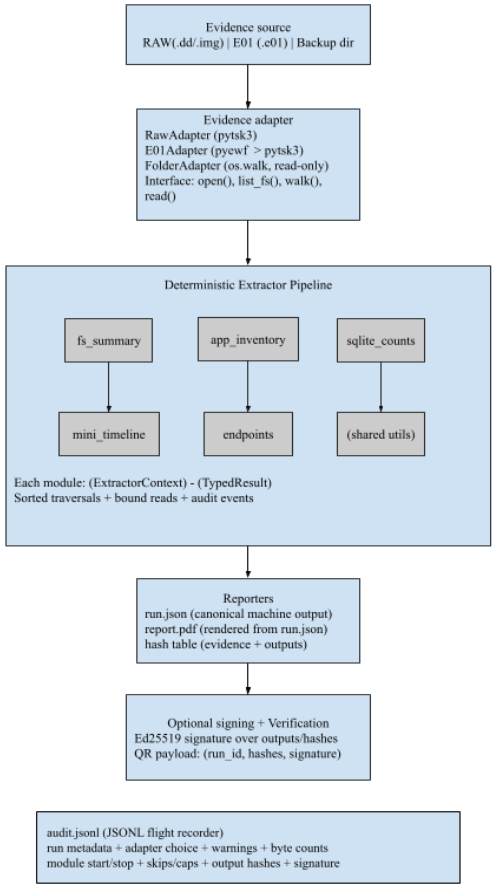

Figure 1
MDTK End-To-End Pipeline

Evidence Adapters

Evidence access is the most safety-critical layer, so MDTK isolates all I/O behind adapters that expose a strictly read-only interface. Adapters open the evidence source, enumerate filesystem structures (when applicable), and return bytes for bounded reads; they never mount evidence into the OS namespace and provide no write operations.

Disk images support - For RAW images, MDTK uses pytsk3 to parse partition tables and traverse supported file systems via The Sleuth Kit primitives. For E01, MDTK uses pyewf to stream EWF segments and present a read-only byte addressable source into pytsk3 (or an exported RAW-compatible stream). This design avoids relying on OS-level mounting while still enabling filesystem traversal at triage speed.

Backup/logical folder support - For logical sources, MDTK uses a directory-based adapter that provides the same walk/read semantics as image-backed adapters. This keeps extractors identical regardless of acquisition type; they operate on an abstract "tree of paths and bytes" rather than branching on "image vs folder" in every module.

Adapter interface – The adapter exposes minimal set of fundamental functions, for example:

- open () to initialize and validate the evidence source (and record adapter metadata)
- list_fs() to enumerate partitions/filesystems (for images)
- walk(root) to deterministically yield paths and metadata
- read(path, offset, length) to return bytes under bounded limits.

As shown in Figure 1, adapters form the pipeline's first stage and expose only read-only methods that return bytes/iterables and are defined to be free of side effects. If the adapter detects a condition that suggests writing risk (e.g., evidence appears mounted writable or the source is a live path rather than image), it signals a fail-closed refusal before any extraction proceeds. Adapter selection and evidence metadata are always recorded in the audit log to support later review.

Extractors

MDTK's extractors are intentionally small and opinionated: each produces a bounded summary rather than attempting comprehensive application decoding. This matches MDTK's purpose, supporting triage escalation decisions rather than full mobile analysis.

- fs_summary – Provides quick context about the evidence layout. For images, it records detected partitions, filesystem types (via TSK), basic volume metadata, and top-level counts under a fixed traversal strategy. For folder sources, it reports basic tree statistics (file/byte totals, depth measures) and presence of known backup markers. The summary is produced early and informs safe caps without introducing nondeterministic branching.
- app_inventory – Performs deterministic presence checks for installed applications using known directory conventions and manifest indicators. Outputs a normalized list of inferred app identifiers (when possible) with evidence paths and lightweight metadata; unknown apps are still represented as directory entries to avoid silent omission.
- sqlite_counts – Treats SQLite triage as safe enumeration rather than analysis. It validates headers, lists tables, and records row counts using read-only queries while avoiding operations that can trigger writes (e.g., VACUUM or write-enabled pragmas). It notes WAL/journal artifacts when present and degrades gracefully by logging skipped/errored databases rather than failing the run. SQLite structure and header expectations follow the official SQLite database file format documentation [4] [7].
- mini_timeline – Produces a bounded chronology from allow-listed timestamps normalized to UTC seconds. It applies caps and throttling to avoid scanning every file and emits a stable ordering (timestamp, then path) for reproducibility. Where compatible timeline conventions are referenced, MDTK aligns with mactime(1)-style timelines [8] [9].
- Endpoints – Reports deterministic presence hits for high-value artifact families (e.g., messaging, browser, logs) as structured records (pattern ID, matched path, minimal metadata). Endpoints are treated as indicators for escalation, not final conclusions.

Determinism and Logging

Determinism is enforced at both traversal time and output time.

Deterministic traversal – All enumeration operations are ordered using stable sorting rules. Directory entries are emitted in a consistent order, and extractor lists (apps, databases, endpoint hits, events) are sorted using fixed keys. MDTK avoids

concurrency in extraction stages where scheduling could reorder results, and it normalizes ambiguous elements (e.g., timestamps) to UTC seconds.

Deterministic serialization – JSON outputs are emitted with fixed key ordering and stable formatting rules so that identical result yield byte-identical run.json. Where the PDF is derived from JSON, the PDF generation is likewise constrained to avoid nondeterministic fields (e.g., random object IDs, variable ordering of tables). When a field must vary (such as a "run started at" time), MDTK either isolates it into the audit log rather than the main report or explicitly documents it as a non-hash-stable field and excludes it from reproducibility claims.

Audit logging – MDTK emits an append-only JSONL audit log that functions as a structure flight recorder. Each entry includes:

- Run identifiers, tool version, and dependency versions (environment pinning)
- Adapter selection and evidence metadata (including hashes where applicable)
- Extractor start/stop records, warning, and exceptions
- Byte counts read per module, caps applied, and any skipped paths
- Output hashes (JSON/PDF), and optional signature metadata.

The audit log is designed to be sufficient to be sufficient for reconstructing "what happened" without relying on screenshots or manual notes.

Outputs and Signing

MDTK produces two primary outputs per run:

- run.json (machine readable): A single structured document containing evidence identifiers, module, results, and summary statistics. This serves as the canonical output for reproducibility testing and for downstream tooling.
- Uniform PDF report (human-readable): A deterministic report rendered from run.json, presenting key tables such as filesystem summary, app inventory, SQLite tables counts, endpoint hits, and a mini-timeline. The PDF includes a hash-table (e.g., SHA-256) for the evidence source (when available) and the generated outputs, enabling quick integrity checks. PDF generation is implemented using ReportLab [3] [10].

Optional Ed25529 signing- MDTK optionally signs selected outputs (e.g., run.json, the PDF hash, and/or a verification bundle) using Ed25519. This supports provenance and tamper detection when reports are shared or archived. When enabled, MDTK can embed a compact verification payload (e.g., signature + hashes + run ID) as a QR code in the PDF to support quick validation in classroom or field settings.

IMPLEMENTATION

MDTK is implemented command-line application to support repeatable, scriptable triage runs in both classroom and field environments. The CLI acts as the single-entry point that binds together evidence adapters, extractor selection, output configuration, and runtime guardrails. This design reduces "operator variability" by making the run configuration explicit, serializable, and auditable [11].

Command-line Interface

The primary workflow is a single command that accepts an evidence source and a selection of extractors and profiles:

- triage run <evidence> --extract … --profile quick|standard|deep

Profiles provide predefined bundles of extractors and limits tuned for triage urgency. For example, quick emphasizes endpoint presence checks, filesystem summary, and a small mini-timeline; standard adds broader SQLite counting and a larger app inventory; deep increases caps and expands endpoint patterns while still maintaining bounded behavior. The CLI also exposes explicit overrides for key bounds (e.g., maximum files, maximum bytes per file, timeline cap) so that operators can adapt to unusually large sources

without changing code. On every run, MDTK serializes the resolved configuration (including defaults) into the canonical JSON output and the audit log so that the exact conditions of execution can be reproduced.

Deployment and Usage

MDTK is distributed as a Python package and executed via a command-line interface to ensure repeatable triage runs. Installation is performed inside a dedicated Python 3.11 virtual environment to pin dependencies and reduce host variability. After environment creation and activation, the operator installs toolkit dependencies and runs triage using the triage entrypoint.

The resolved configuration (profile selection, enabled extractors, and enforced bounds) is recorded in both the run outputs and the append-only audit log to preserve reproducibility and support later reviews.

Guardrails and Refusal Logic

To enforce "zero-write" safety, MDTK implements guardrails that detect risky mount states and fail closed. Before opening evidence, the tool checks whether the supplied path appears to be mounted in a writable mode. On Linux and WSL, this is implemented by parsing /proc/mounts (or /proc/self/mounts) and matching the evidence path to its mount entry to inspect mount options (e.g., presence of rw vs ro). Where /proc/mounts is unavailable or ambiguous, the tool can fall back to invoking mount and parsing its output.

If a mount entry is detected and indicates writable access, MDTK refuses to proceed and emits a clear diagnostic describing what was detected and how to correct it. If the state is ambiguous (e.g., the evidence path does not map cleanly to a mount entry), MDTK treats ambiguity as unsafe and aborts unless the operator supplies evidence via a non-mounted image path or a verified read-only source. This refusal logic is logged as a structured audit event, including adapter selection and the refusal reason, making safety decisions reviewable after the fact.

Error Handling and Safe Degradation

Mobile triage must tolerate imperfect inputs: corrupted files, partial backups, unknown app schemas, and large databases are common. MDTK therefore uses a "graceful skip" strategy rather than treating individual extractor failures as fatal. Extractors are wrapped in a standard module envelope that records start/stop, bytes read, and outcome status (ok, warn, skip, error). When a parser encounters an exception or an unsupported format, the extractor logs a warning (or error) event and continues, returning a partial result with explicit skip reasons. This prevents a single malformed SQLite database or unexpected directory structure from collapsing the entire run.

To avoid silent failure, MDTK distinguishes between "no findings" and "not processed." For example, an endpoints module reports explicit misses, while a skipped extractor reports a structured reason such as cap_exceeded, corrupt_header, or timeout. These statuses are propagated into both run.json and the PDF so the report communicates limitations clearly.

Time and Space Bounds

MDTK enforces bounded reads to remain triage-speed and to prevent runaway scans on large images. Bounds are implemented at multiple layers:

- Traversal caps: maximum directory entries visited and maximum recursion depth, applied deterministically to prevent "scan everything" behavior.
- Per-file limits: maximum bytes read per file and maximum file size eligible for certain extractors (e.g., excluding large media from SQLite enumeration).
- Extractor-specific caps: maximum SQLite databases examined, maximum tables counted per database, and maximum timeline events emitted.
- Throttling: early stopping rules that terminate an extractor once its goal is satisfied (e.g., endpoint checks stop after a threshold of hits per category).

All caps are treated as part of the run configuration and are logged whenever they influence behavior (e.g., "timeline capped at N events," "SQLite counting limited to first N databases by path order"). This makes performance predictable and preserves interpretability: a fast run should not hide the fact that it was fast because it applied limits.

Testing Strategy: Unit Tests and Golden Outputs

MDTK's testing strategy targets both correctness and reproducibility. Unit tests validate core primitives (path normalization, stable sorting, hashing, schema validation) and adapter behavior using small synthetic images or directory fixtures; extractors are tested with representative fixtures (e.g., SQLite samples) to ensure deterministic behavior and enforcement of configured bounds. For end-to-end stability, MDTK uses golden output tests: each sample input is executed under a fixed profile and the SHA-256 hashes of run.json and the PDF are compared against reference values. Tests fail on unexpected changes, catching nondeterminism from dependency upgrades, traversal bugs, timestamp formatting, or PDF rendering drift. When changes are intentional (e.g., schema or endpoint updates), golden references are updated in a controlled commit with a brief rationale.

EVALUATION

This section evaluates MDTK along three axes that matter for triage: speed (can it run fast enough to inform decisions), coverage (does it surface high-value artifacts reliably), and reproducibility/safety (are outputs stable and does it refuse unsafe conditions). We report results on a small manifest of mobile-like images and backup-folder sources, using fixed host conditions and repeated runs to test determinism.

Datasets

We evaluate MDTK on a small set of publicly available or lab-prepared datasets that resemble mobile evidence sources, including both disk images (RAW/E01) and logical folder structures. As shown in Table 1, for each dataset we record size, acquisition format, and SHA-256 checksums to uniquely identify the input.

Table 1
Dataset Manifest

Image ID	Format	Size	SHA-256
HTC_Desire_S (A)	RAW	2.2 GB	6d6548…
HTC_One_XL (B)	RAW	15.27 GB	b14e26…
N115015_CHIP_OFF (C)	001	7.6 GB	6e7952…
Jo-favorites-usb-2009-12-11 (D)	E01	0.211 GB	1798e0…

Metrics

We report metrics aligned with MDTK's goals:
Performance

- Runtime (wall-clock) per run and per extractor
- Files walked and total traversal depth (proxy for workload)
- Bytes read (from audit log) to characterize I/O intensity

Coverage proxies

- App directories detected (app inventory yield)
- SQLite databases detected and tables counted
- Endpoint hits (count by endpoint category)
- Mini-timeline rows (bounded event count)

Output footprint

- JSON size (MB)
- PDF pages + PDF size

Determinism/reproducibility

- run.json SHA-256 match across repeated runs
- PDF SHA-256 match across repeated runs

Safety behavior

- Refusal behavior under simulated RW mount conditions
- Behavior under corrupted SQLite (should warn/skip, not crash)

Experimental Method

All experiments were executed on a single host to minimize environmental variability. For each dataset, we ran MDTK N = 3 times under the same

profile and configuration. We report on the median runtime and the range across repeats to characterize stability. Because filesystem caching can influence performance, we note whether runs were performed under a cold-cache condition (first run after reboot or after dropping caches, where feasible) or a warm-cache condition; reproducibility results should hold under both. All runs used the same MDTK version and pinned dependency set. Outputs were compared using SHA-256 over the emitted run.json and PDF files. Audit logs were inspected to confirm identical traversal counts and extractor yields across repeats.

Results

Table 2 summarizes MDTK's performance and output characteristics across datasets A–D, MDTK completed triage in 2.83–14.93 s, with runtime generally increasing with workload (B: 15.27 GB, 14.93 s, 35 files; C: 10.73 s, 44 files; A: 5.93 s, 6 files; D: 2.83 s, 1 file). No app directories or SQLite databases were detected (Apps=0, DBs=0), while mini-timeline rows ranged from 3 (B) to 4,439 (A); run.json sizes were 9–884 KB. Reproducibility is Yes (3/3) when SHA-256 hashes match for both run.json and the PDF across three runs. Writable-mount safety tests triggered a fail-closed refusal with a logged diagnostic.

Table 2

Metrics Per Image

Image	GB	Time	Files	Apps	DBs	TL	JSON	Repro
A	2.2	5.93s	6	0	0	4439	884KB	Yes
B	15.27	14.93s	35	0	0	3	9KB	Yes
C	7.6	10.73s	44	0	0	165	40KB	Yes
D	0.21	2.83s	1	0	0	739	160KB	Yes

MDTK evaluation metrics per dataset (median of N=3 runs). Runtime is wall-clock time:yields report extractor outputs; JSON size is the emitted run.json.

Triage Decision Support and Escalation

MDTK supports escalation decisions by emitting interpretable triage indicators (app inventory, SQLite presence/counts, endpoint hits, and mini-timeline density) rather than a single "score." Profiles and configurable endpoint lists let organizations emphasize mission-relevant artifacts without changing the read-only workflow, and the selected policy (profile/version/active indicators) is recorded in run.json and the append-only audit log for replicability. High-value indicators suggest full imaging, while minimal signals may justify deferral, with all applied limits and skipped checks surfaced to prevent "low signal" from being misread as "no evidence."

Safety and Refusal Tests

To validate MDTK's guardrails, we executed controlled "unsafe" scenarios intended to trigger fail-closed behavior, as well as a positive control confirming that verified read-only mounts proceed normally. MDTK first performs a mount-state check for the evidence path (or its parent mounts) and requires a verified read-only (`ro`) state before extraction. When presented with a mount configured as read-write (`rw`), MDTK refused to run and exited before any extraction occurred, preventing accidental modification risk. When the mount state could not be verified because the provided path was not a mountpoint, MDTK also aborted (fail-closed) and instructed the operator to provide a verified read-only mountpoint. As a positive control, when the evidence was provided via a verified read-only mount, MDTK proceeded successfully and produced outputs. These behaviors demonstrate that MDTK treats both explicit `rw` and ambiguous mount state as unsafe conditions and refuses to proceed, while allowing normal execution under verified `ro` conditions.

As shown in Table 3, MDTK failed closed when evidence was writable or the mount state was unverifiable: it refused on rw mounts and aborted on non-mountpoint inputs. Under a verified read-only mount, MDTK ran successfully and produced the expected outputs, confirming guardrails that block triage under write risk or ambiguity.

These results support MDTK's fail-closed guardrails: the toolkit proceeds only under a verified read-only mount and refuses execution when evidence is writable or the mount state is unverifiable. By blocking extraction under write risk or ambiguity and emitting explicit refusal

reasons, MDTK reduces operator error and strengthens forensic defensibility.

Table 3
Safety and Refusal Test Outcomes

Test	Setup	Expected behavior	Observed result	Outcome
RW mount refusal	Evidence path under rw mount	Refuse before extraction	Refused with message indicating mount is rw	Pass
Ambiguous mount state	Path is not a mountpoint	Abort fail/closed	Aborted: "cannot verify read-only state … not a mountpoint"	Pass
RO mount positive control	Evidence path under ro mount	Proceed normally	Mount check showed ro; run completed and emitted JSON/PDF	Pass

DISCUSSION

In this section, we interpret the empirical results in practical, investigative terms and relate them back to MDTK's design goals. Rather than emphasizing individual metrics in isolation, we focus on what the observed runtime, output stability, and fail closed behavior imply for real-world triage workflows, including when MDTK is most useful and where deeper analysis tools remain necessary.

Operational Meaning of the Results

The evaluation indicates MDTK can speed triage by producing a consistent, low-latency snapshot of an evidence source and whether it likely warrants full acquisition. The main benefit is structured output: filesystem summaries, presence indicators, and bounded timelines support quick escalation or deferral decisions. Producing both run.json and a PDF also makes triage outcomes easy to document and review.

Trade-offs: Breadth vs. Depth

MDTK favors breadth-first triage over deep, app-specific parsing. Endpoint checks and lightweight SQLite enumeration can quickly surface high-value artifacts (e.g., messaging or browser stores), but they do not replace full interpretation, decryption, or semantic parsing. This is intentional: deeper parsing increases complexity, runtime variability, and failure risk. MDTK outputs should therefore be treated as escalation signals, interpreted alongside logged limits and skips.

Portability and Deployment Constraints

MDTK is easiest to deploy on Linux/macOS where filesystem tooling and read-only verification are native. On Windows, WSL enables execution but adds complexity (mount behavior, permissions, and platform-specific dependencies such as libmagic). Containerization can improve consistency by pinning dependencies, but safe evidence handling still depends on host controls and least-privilege operation. In practice, MDTK works best with explicit guidance on permissions and narrowly scoped elevation for mount checks. PDF generation is implemented using ReportLab [3], [10].

Where Determinism Provided Concrete Value

Deterministic traversal and canonical serialization make outputs easy to reproduce and compare across runs, analysts, and tool versions. This simplifies diffing and supports CI regression testing using golden run.json and PDF hashes. Determinism also improves defensibility because results can be regenerated under the same policy and configuration, with audit logs capturing bounds and skips that affect interpretation.

Limitations and Future Directions

These results reflect triage-focused datasets and bounded extraction; stronger completeness claims require broader corpora and ground-truth labeling. Mount-state refusal is only as reliable as the platform checks and must be implemented carefully to avoid false assurance across environments. Future work includes versioned endpoint/profile expansion, stronger cross-platform safety verification (especially for Windows-native use), and larger-scale evaluation while preserving determinism and zero-write guarantees.

LEGAL/ETHICAL AND CHAIN OF CUSTODY

MDTK supports chain-of-custody documentation by embedding evidence identifiers (e.g., SHA-256), execution timestamps (UTC), tool version, configuration/profile, and the exact command line used to generate outputs. These metadata are included in both machine-readable JSON and the human-readable PDF to enable later verification and reproducibility. MDTK is designed for triage and therefore minimizes unnecessary exposure of personal content by default, prioritizing summaries, counts, and presence indicators over bulk extraction. Any applied bounds, skipped paths, and refusal decisions are recorded in an append-only audit log to preserve transparency during early-stage handling.

CONCLUSION

We show that a minimal, read-only pipeline can provide actionable triage outputs while preserving forensic defensibility through deterministic execution and comprehensive audit logging. Across evaluated datasets, MDTK produced stable outputs and failed closed under unsafe mount conditions. These properties support rapid escalation decisions without sacrificing transparency or reproducibility.

REFERENCES

[1] J. Metz. (2025, Dec.). *libewf: Libewf is a library to access the Expert Witness Compression Format (EWF)* [Online]. Available: https://github.com/libyal/libewf.

[2] J. Metz. (2024, May, 5). *libewf-python 20240506* [Online]. Available: https://pypi.org/project/libewf-python/.

[3] A. Robinson, R. Becker, and the ReportLab team. (2026, Jan.). *ReportLab PDF Generation User Guide* [PDF]. Available: https://www.reportlab.com/docs/reportlab-userguide.pdf.

[4] SQLite. *Database File Format* [Online]. Available: https://www.sqlite.org/fileformat.html.

[5] M. Cohen. *pytsk3* [Online]. Available: https://pypi.org/project/pytsk3/.

[6] py4n6. pytsk: *Python bindings for The Sleuth Kit (libtsk)* [Online]. Available: https://github.com/py4n6/pytsk.

[7] SQLite. *SQLite Documentation* [Online]. Available: https://www.sqlite.org/docs.html.

[8] B. Carrier. *mactime(1) Arch Linux Manual Pages* [Online]. Available: https://man.archlinux.org/man/extra/sleuthkit/mactime.1.en.

[9] B. Carrier. (n.d.). *mactime(1) The Sleuth Kit manual page* [Online]. Available: https://www.sleuthkit.org/sleuthkit/man/mactime.html.

[10] A. Robinson, R. Becker. *ReportLab Documentation* [Online]. Available: https://docs.reportlab.com/.

[11] Pallets. *Welcome to Click* (Click Documentation 8.3.x) [Online]. Available: https://click.palletsprojects.com/en/stable/.