Mobile Verification Toolkit & Android Quick Forensic Secure Design Assessment

0xCHE

MAY 2026

Table of contents

Executive Summary

The Mobile Verification Toolkit (MVT) and Android Quick Forensics (AQF) are the go-to open source tools of the consensual digital forensics community. Both tools are FLOSS, and that openness, rare in a forensic field dominated by non-consensual opaque tools, is what makes their methods auditable and a study like this one possible.

This report assesses the two tools as part of a single consensual forensic workflow through three lenses:

- `Secure design assessment`: a threat model of the two-stage workflow (acquisition and analysis), followed by an architectural review against four core questions: untrusted input handling, forensic soundness, case isolation and network boundaries, and analytical clarity and actionability.
- `Security analysis`: a code-level review of AQF and MVT identifying vulnerable code paths, with concrete remediations.
- `Heuristic analysis`: a usable-security evaluation of both CLIs against a ten-heuristic framework organized into three themes (fostering trust and transparency, optimizing critical cognition, and crafting the interaction framework) anchored in real extraction and analysis sessions.

Appendices include the complete findings index (A), the consensual forensic community survey results (B), a secure-by-design CLI proposal for AQF (C), and operational security guidelines for practitioners (D).

This work is offered as a contribution to the community that builds and sustains MVT and AQF, and as a model for how security and usability analysis can reinforce each other in tools that people depend on when the stakes are highest.

These tools and its workflows have proven genuinely valuable in countering spyware abuses and have a pivotal resource for a growing community. Like any living project, they have room to grow and some limitations worth acknowledging, but none of this diminishes the serious effort they represent to make consensual forensic work openly available. Assessing them carefully is worth doing because it can support their maturity, lead to useful debate and inform recommendations as needs change and the landscape shifts.

Acknowledgements

We are deeply grateful to the consensual forensics community for their time in completing the survey, to the technologists who participated in our interviews, and, in particular, to Amnesty International's Security Lab, Etienne, Filip, Fembloc Forensic Laboratory, Gurshabad Grover, Janik Besendorf, r and SocialTic for their time and support. This work was funded by the Security Lab at Open Technology Fund.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funders.

About the authors

0xche is a horizontal, worker-led collective of technologists committed to strengthening the information security of activist organizations and civil society in Latin America. Our identity combines a commitment to social transformation with solid technical expertise.

Licence

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International CC BY-NC-SA 4.0

Suggested citation

0xche, (2026), Mobile Verification Toolkit & Android Quick Forensics Secure Design Assessment

Introduction

Consensual Digital Forensics is a field of knowledge created by human rights defenders and technologists working in the civil society sphere, born out of the need to respond to the growing threat of sophisticated spyware deployed against activists, journalists and dissidents. The scale and severity of these advanced surveillance attacks led to the emergence of a new discipline, with its own methodologies, tools and approaches for acquiring and analyzing digital evidence in defense of human rights, giving visibility to abuses and supporting accountability efforts.

Within this field, the Mobile Verification Toolkit (MVT) and Android Quick Forensics (AQF), originally developed by Amnesty International's Security Lab and sustained by a broader community of developers and practitioners, have become core to current consensual forensic workflows. Both are open-source projects designed specifically for this consensual approach, where acquisition and analysis are conducted with the informed consent of the device owner.

It is important to stress that both MVT and AQF are open-source software (FLOSS). In a forensic field where most acquisition and analysis tools operate as proprietary black boxes, the transparency of these tools is a distinctive approach and a core characteristic of the consensual forensics field. Their inner workings, including extraction logic, parsing routines and analytical methods, can be independently audited and verified. This openness strengthens not only the technical integrity of the workflow but also the credibility of its outputs, allowing any party to inspect exactly how evidence was collected and processed.

Rather than analyzing each tool in isolation, we decided to take the real-world workflow as our scope of study, in which data is acquired from a device and then forensically analyzed. This included mapping the extent of their use, identifying key users and organizations within the community, and documenting emerging operational practices.

This research analyzes security risks in the mobile consensual forensic workflow built around:

- `Androidqf` (device acquisition). V1.8.1
- `Mobile Verification Toolkit` (forensic analysis). V2.7.0

Finally, this report was grounded in a Consensual Digital Forensic Community Survey conducted as part of this project with practitioners working on acquisition and forensic analysis in defense of human rights. The survey results (see Appendix B) provided insights into their experience building, maintaining and using these tools in the field. This work reflects on those shared experiences and aims to contribute to the active, ongoing efforts of advancing the field of consensual forensics led by this community.

These sources informed this report which is presented in four main sections: a Secure Design Assessment which provides design recommendations to mitigate threats, followed by a Security Analysis that identifies vulnerable code paths and suggests remediations. The third is a Heuristic Analysis, which reviews usability pain points and enhancements for frontliners and analysts, and the final is an appendixes section, in which we summarized findings, shared the results of the community survey and gave recommendations to the practitioners community, to help keep their operations safe.

Secure Design Assessment

The section is organized in three parts. First, the threat model establishes actors involved, what is at risk, and five threat scenarios that frame the adversary's capabilities. Second, an overview of how encryption, key management and bundle integrity are implemented in the current codebase. Third, the architectural review examines the workflow against the following four core questions.

Topic	Core question
`Untrusted input handling`	Can adversary controlled data from the device or the sample reach sensitive operations unsanitized?
`Forensic soundness`	Does the workflow produce evidence whose completeness and integrity can be verified?
`Case isolation and network boundaries`	Can one case contaminate another on a shared workstation, or can the forensic process leak information outward?
`Analytical clarity and actionability`	Can the analyst correctly interpret detections and assess their severity?

Each topic identifies an architectural gap, links it to specific findings from the Security Analysis and Heuristic Analysis, and describes the design implication for the workflow. The Security Analysis and Heuristic Analysis sections that follow examine specific vulnerable code paths and usability for the frontliner and analyst, respectively.

Threat model

Threat modeling was developed based on the following assumptions:

- The HRD, the acquisition frontliner and the forensic analyst are not malicious.
- Trust between these parties is established through prior vetting processes and trusted partnerships.
- The acquisition process is conducted with informed consent.
- The adversary is external and may possess advanced technical capabilities.
- The device may be currently compromised or may contain traces of previous targeted attacks.
- The forensic workflow performed may be conducted as specific technical engagements or form part of broader holistic protection support.

We distinguish two operational stages of the workflow analyzed:

1. `Acquisition Stage`: the device is physically connected to a workstation for data collection (via USB/ADB).
2. `Forensic Analysis Stage`: collected artifacts are processed and analyzed, often asynchronously and potentially by a different organization.

In practice, acquisition and analysis may be conducted:

- By the same organization (such as, grassroots technologist performing both stages, even using an unique workstation) or
- By separate entities (such as, a frontline organization performs acquisition while a threat lab performs the forensic analysis).

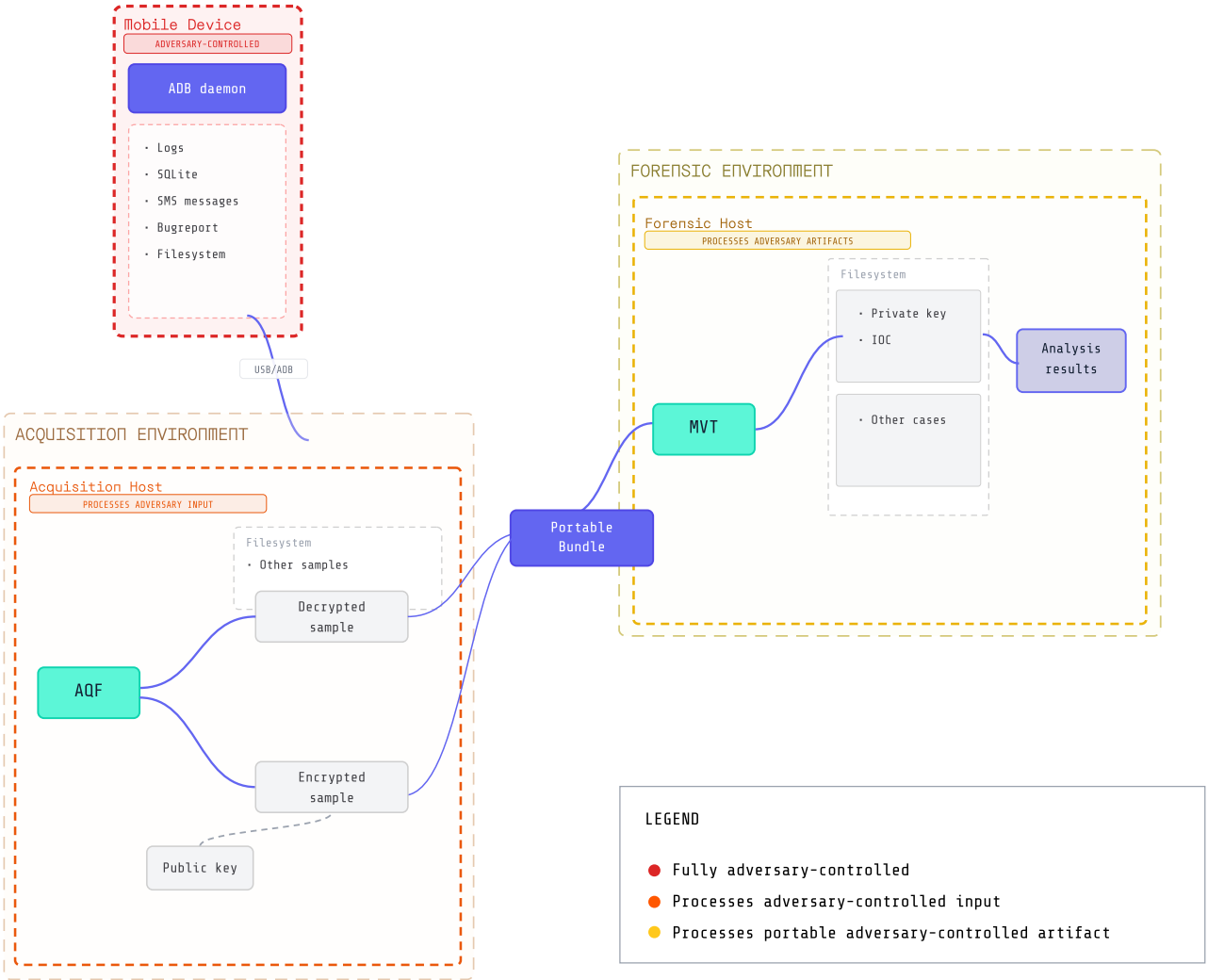
Each stage presents distinct attack surfaces and therefore requires separate threat modeling.

Main actors, Assets and Trust zones

ACTORS		
Acquisition Frontliner	*Characteristics*	• May or may not be highly technical • Operates in constrained environments (poor internet, urgency) • Connects multiple potentially infected devices to acquisition workstation • Workstation may be shared across cases • Does not necessarily need plaintext access to evidence • Needs assurance acquisition was complete • Must securely store and transfer data post-acquisition for analysis • Responsible for secure data retention/deletion policies
	Security Implication	High exposure to adversary controlled input via USB/ADB.
Forensic Analyst	*Characteristics*	• Technically skilled • Performs asynchronous analysis • Requires plaintext access to acquired logs • Does not necessarily interact directly with infected device • May reuse workstation for multiple cases • Must securely communicate findings back to the frontliner/HRD owner of the phone. • Responsible for secure data retention/deletion policies
	Security Implication	Primary risk is malicious artifact processing (parsing risks, archive handling).
Adversary	*Capabilities*	• Full control of HRD device (potentially connected via USB to frontliner workstation) • Ability to include malicious payloads in the sample, such as: malicious archives, path traversal entries, malicious JSON / SQLite, decompression bombs. • Possible physical seizure of devices/workstations
	Goals	• Compromise acquisition workstation • Compromise analysis workstation • Exfiltrate sensitive HRD investigation data • Disrupt acquisition (availability attack) • Damage forensic credibility

ASSETS
Acquisition workstation integrity
Analysis workstation integrity
Confidential HRD data
Non-public Indicators of compromise (IOCs)
Chain of custody preservation
Frontliner/Analyst credentials (such as encryption keys)
Plain text / encrypted evidence bundles (samples)

TRUSTZONES
Device: Fully adversary controlled
Acquisition Host: Connected via ADB to Device & Processes adversary controlled input
Analysis Host: Processes portable adversary controlled artifact



Threat Scenarios

The following table summarizes the identified threat scenarios, based on the TRAIL methodology developed by Trail of Bits.

SCENARIO	ACTOR(s)	COMPONENT(s)	ATTACK VECTORS	IMPACT
Infected device targeting acquisition workstation. A compromised HRD device is connected via USB/ADB and delivers adversary controlled data to the acquisition environment.	External attacker controlling device	• Mobile device • USB/ADB stack • Acquisition workstation • AQF	• Path traversal during file copy • Symlink abuse • Zip-slip extraction • Command injection • USB stack exploitation	• Workstation compromise • Confidential data exposure • Cross-case contamination • Loss of acquisition integrity
Infected device preventing acquisition or analysis (availability attack). The adversary attempts to block evidence collection rather than compromise the host directly.	External attacker controlling device	• Mobile device • AQF • Acquisition workstation	• Deep directory nesting • Nested archives / zip bombs • Excessive recursion • System resource exhaustion	• Workstation unresponsive • Failed or partial acquisition • Evidence loss • Operational disruption
Malicious acquisition bundle targeting forensic analyst workstation. Adversarial payloads embedded in the collected sample trigger vulnerabilities during forensic analysis.	External attacker controlling the data bundled in the sample. Adversary distributing malicious bundle	• Portable acquisition bundle • MVT • Analysis workstation • Parser components	• Zip-slip archive extraction • Malicious JSON structures • Crafted SQLite corruption payloads • Terminal escape injection • Command execution via parsing • Decompression bombs	• Analysis workstation compromise • Cross-case contamination • Loss of forensic credibility
Forensic clinic operational risk. Multiple acquisitions occur sequentially on a shared workstation under time pressure, increasing contamination risk even without a sophisticated attacker.	Operational risk. Adversarial device affecting other cases	• Shared acquisition workstation • Case storage directories • Temporary filesystem artifacts	• Residual artifacts between cases • Shared temporary folders • Naming collisions • Persistent malicious files across sessions	• Mixing evidence between HRDs • Incomplete acquisition • Chain of custody degradation
Physical seizure of devices or workstations. Adversaries obtain physical access to acquisition or forensic workstations.	Law enforcement or state level adversary. Physical attacker	• Acquisition workstation • Analysis workstation • Encryption keys • Evidence bundles • Investigation metadata	• Full disk imaging • Logical forensic extraction (such as Cellebrite, GrayKey) • Recovery of deleted artifacts • Key extraction attempts • Access to stored bundles and reports	• Confidentiality breach • HRD identification risk • Exposure of investigative methods • Legal and operational consequences • Compromise of investigations

Overview of Security components

Encryption implementation

AQF supports two acquisition modes. In streaming mode, data flows from ADB through memory into a zip writer and then directly into age encryption before reaching disk. No plaintext archive is written locally. In traditional mode (when no encryption key is provided), the tool writes a plaintext directory to disk with no compression or encryption.

Key management

AQF provides an opt-in encryption option using age encryption scheme for the acquisitions output to be encrypted under a `.zip.age` bundle. The encryption public key is loaded by AQF from a fixed `key.txt` file in the executable directory. The tool validates that the key is a valid age recipient format. The same key is used for all subsequent acquisitions, unless the `key.txt` file is changed. There is no support for per-case keys, alternate key paths or runtime key selection. On the decryption side, the analyst needs the corresponding age private key to open the bundle. AQF does not generate keypairs, distribute private keys or provide guidance for key handoff. This is entirely an operational responsibility outside the tool.

Sample integrity model

In traditional mode, AQF generates a `hashes.csv` file containing SHA256 hashes of all files in the bundle. In streaming mode, hash generation is skipped. Both modes produce an `acquisition.json` file recording: acquisition UUID, AQF version, storage path, start and completion timestamps, collector information (binary path, architecture), device temp directory and SD card paths and whether streaming mode was used. It does not include a module execution status or a record of operator scoping decisions.

On the analysis side, MVT writes an `info.json` in the output directory recording: MVT version, analysis timestamp, target path, IOC files loaded, and optionally newly computed file hashes of the input (disabled by default, enabled if flag `-H` is given). There are no cross validation of bundle integrity between acquisition and analysis.

Both tools generate a `command.log` file where AQF logs the acquisition session (using a verbose `DEBUG` level) and MVT the analysis session. Both tools log under these files the errors and partial failures if any.

Architectural review and recommendations

Untrusted input handling

This section examines how the tools handle attacker controlled input coming from mobile devices and bundled within forensic artifacts. The focus is on validation and sanitization controls applied before untrusted data reaches file system operations, archive handling or parser logic.

Architectural finding

- No consistent sanitization model for untrusted artifact and sample ingestion.
- No cross tool validation contract between AQF as producer and MVT as consumer.

Related findings in later sections

This architectural gap is reflected later in Secure Analysis findings for AQF: AQF-SA-L1 (APK download Path Traversal in device APK paths) and AQF-SA-L2 (Zip entry name Injection in APK bundle). And for MVT: MVT-SA-M1 (Arbitrary file write via Path Traversal in iOS check-backup), MVT-SA-L1 (Arbitrary file read via Path Traversal in iOS check-backup) and MVT-SA-L2 (Arbitrary file read via Path Traversal).

Those findings show the specific code paths where untrusted values reach file writes, file reads, archive names or parser selected filenames.

Design recommendations

`Centralized sanitization control`. Both tools should enforce a centralized validation control for untrusted names, paths and archive entries at their respective ingestion points -AQF when pulling from the device, MVT when opening a bundle- so security does not depend on individual module authors remembering to sanitize.

`Path confinement at the sink`. Every file write and file read that derives a path from device-controlled or bundle-controlled input should verify the resolved path stays within the intended output directory before proceeding (such as, `filepath.IsLocal` in Go, `Path.is_relative_to()` in Python).

`Operational security guidance`. If untrusted input handling cannot be fully guaranteed at the code level, both tools should clearly document their threat model assumptions and provide explicit operational guidance for safe acquisition and analysis, including recommendations for compartmentalization (such as, using a dedicated single purpose device for acquisition, avoiding shared workstations with other sensitive data), so that operators can layer operational controls around the tools known trust boundaries. See Appendix D.

Forensic soundness

The tools examined here are unique, and we can assess them at all precisely because they are open source. That openness is what lets us look closely, identify gaps, and make recommendations. Even with their limitations, they remain the best tools available for consensual forensic practice, and assessing them carefully is worth doing because it can contribute to their maturity and keep them aligned with changing needs and a shifting landscape.

This part examines whether the workflow produces evidence that is sufficiently complete and reliable for later verification. In this context, forensic soundness turns on two related but distinct questions: whether the maximum reasonably available relevant data was collected, and whether the resulting evidence can later be independently verified for completeness and integrity.

These questions are especially important in consensual forensics, where collection often occurs under operational constraints and handoff may span multiple people or organizations. In this setting, chain of custody is often less formalized than in law enforcement practice. Accordingly, this review takes a more functional approach, asking whether the workflow preserves, as much as possible within these constraints, the technical procedures and safeguards that would support later evidence verification.

A. Acquisition trust model

This subsection focuses on acquisition completeness, partial failures, visibility into what was and was not collected, and whether both the frontliner and later the analyst can understand the known limits of the sample.

Architectural finding

- No explicit completeness model for the acquired sample
- No reliable way to tell whether expected evidence was not present on the device, not collected or simply not processed later

Related findings in later sections

This architectural gap is reflected later in Heuristic's findings MVT-UX-H3 (Timeline data quality), MVT-UX-M1 (False positive handling), MVT-UX-M2 (Error message confusion), AQF-UX-C1 (Interactive prompt fail) and AQF-UX-H1 (Silent partial completion).

Design recommendations

`Acquisition manifest.` Each acquisition should produce a record of what was attempted, what succeeded, what failed, and what was unavailable. This completeness accounting mechanism should remain attached to the sample across the full workflow, so the analyst can distinguish "not found on device" from "not collected" from "collection failed."

`Partial acquisition flagging.` If acquisition is interrupted or a module fails, the workflow should explicitly mark the sample as incomplete and surface a warning, rather than letting it appear finalized with silently missing data.

`Scoping decision capture.` Operator choices that affect acquisition scope (such as, skip APKs, skip backup) should be recorded in the bundle metadata so the analyst knows what was intentionally excluded versus what failed or was absent.

B. Cryptographic and integrity architecture

This subsection examines whether the workflow preserves the integrity of the collected material after acquisition. It focuses on mechanisms such as integrity verification that help demonstrate that the material analyzed later corresponds to what was originally acquired and has not been altered.

Architectural finding

The current integrity model has two fundamental gaps. First, AQF lacks a clear finalization stage for the acquired sample. The tool mixes writing, hashing and permission hardening in a single pass, leaving some artifacts outside the protection boundary. Second, there is no cross-validation of bundle integrity between acquisition and analysis. MVT never reads the integrity data that AQF produces, so a tampered bundle is processed without warning.

Three structural problems underlie this:

- `Writing, hashing and locking are mixed in a single pass.` `HashFiles()` walks the case directory and, for each file, chmods it to read-only *and* computes its SHA256 in the same loop. Anything written after the walk (notably `acquisition.json`) escapes both the integrity record and the lockdown. The result is that the workflow has no single auditable moment where the sample becomes complete, immutable and ready for verification.
- **`hashes.csv`** `is self-referential and unreliable.` The integrity record is a plain CSV stored inside the same directory it is supposed to protect, and it includes an entry for itself. Because `hashes.csv` is still being written when its own hash is computed, the recorded value depends on buffer timing and may reflect an empty or partial file rather than the final content. Beyond this self-referential flaw, the current design provides no detached integrity record that binds the finished sample, its metadata and any handoff information together. As a result, the file cannot serve as a reliable integrity anchor for the acquisition bundle.
- `No cross-validation of integrity.` MVT never reads `hashes.csv` from AQF. A tampered bundle is parsed without warning as long as the files remain structurally valid. MVT does offer an optional `--hashes` flag that computes fresh SHA256 hashes of the input files and records them in `info.json`. However, this is independent hash generation for the analyst's own reference, but it never compares against AQF's `hashes.csv` to verify that the bundle matches what was originally acquired. The integrity chain is broken at the AQF-to-MVT handoff boundary.

Additionally, `streaming mode skips integrity entirely.` In streaming mode, no integrity record of any kind is produced for the encrypted bundle.

Related findings in later sections

This architectural gap is reflected later in Heuristic's findings AQF-UX-L1 (Comprehensive artifact collection), MVT-UX-M3 (Evidence integrity and user control) and Security findings for AQF-SA-I1 (Acquisition output permissions allow unnecessary read access and modification). Together, those findings show that the workflow does not turn the sample into a clearly defined immutable state once acquisition is complete.

Design recommendations

The integrity model should be structured as sequential phases, each completing fully before the next begins:

`Phase 1: Write.` All artifacts are written with restrictive permissions from the start (see AQF-SA-I1 for current defaults and recommended values). Once all modules have finished, no further writes occur inside the case directory.

`Phase 2: Hash.` Compute SHA256 hashes for every file in the case directory. Write them into a detached manifest (`manifest.json`) outside the case directory as a sibling. The manifest contains hashes, file inventory, module execution

status and operator scoping decisions. It is not itself part of the hash set, but rather an integrity anchor. This replaces the current circular model where `hashes.csv` sits inside the directory it supposedly protects.

`Phase 3: Seal`. Recursive permission lockdown on the case directory and the manifest (files to `0o400`, directories to `0o500`). After this, not even the owner can accidentally modify the sample.

`Consumer-side verification`. MVT should read the manifest's recorded hashes and verify every file in the bundle before running analysis. A mismatch or missing manifest should produce a clear warning, not a silent pass. This cross-validation is the minimum viable integrity check at the AQF-to-MVT handoff boundary and closes the current gap where `hashes.csv` is never consumed. Note that this recommendation depends on AQF first fixing its hash computation (the self-referential condition and artifact exclusion described above); MVT cannot meaningfully validate against hashes that were incorrectly computed at the source.

`Optional enhancement: manifest signing`. Without a signature, an unsigned manifest only guards against accidental corruption. Anyone with access to the bundle can tamper with the data, recompute hashes and rewrite the manifest to match, making it ineffective against a motivated adversary. Signing the manifest with the frontliner's private key closes this gap by adding authentication of origin, because the analyst can verify both that the bundle was not altered and that a specific person produced and vouched for it. This adds operational complexity. Signing requires a different key type than the existing age encryption key (such as `Ed25519` via `minisign`), and the frontliner's public signing key must reach the analyst through a separate trusted channel, since it cannot travel inside the bundle it is meant to authenticate. If implemented, the step order becomes Write, Hash, Sign, Seal and MVT verifies the signature before checking hashes.

Case isolation and network boundaries

This section examines whether the workflow prevents one case from affecting another on a shared workstation, and whether the forensic process can leak information outward.

In a forensic clinic, the same operator acquires multiple HRD devices sequentially on the same workstation under the same user account. Case isolation therefore cannot rely on filesystem user permissions alone as all cases are owned by the same user. Isolation must come from structural boundaries within the tools: per-case directories, unique temp file paths and clean session handoff.

Architectural finding

- `No structural case isolation`. AQF creates per-case UUID directories, but nothing prevents one case's process from reading or writing another's artifacts. The same Linux user owns all cases, so filesystem permissions alone cannot enforce boundaries.
- `No output directory reuse protection`. If the frontliner selects the same output folder for two different cases, the workflow allows reuse and may overwrite existing files.
- `Shared global state`. Temporary files (`bugreport.zip`, `backup.ab`) are written to the current directory -where AQF is executing from- with fixed names, risking collision if two instances share a working directory. The executable directory is also used as an implicit state (such as, for key files), creating cross-case interference points.
- `No effective network boundary` during analysis as the only mechanism intended to control outbound network access is a configuration flag that is never checked.

Related findings in later sections

This architectural gap surfaces in multiple Security findings: AQF-SA-I1 (Acquisition output permissions allow unnecessary read access and modification), AQF-SA-I2 (Serial Flag not passed to ADB), AQF-SA-I3 (Temp files written to CWD with fixed names) and MVT-SA-I2 (Dead config flag. NETWORK_ACCESS_ALLOWED never enforced) and MVT-SA-I3 (Timeline append mode generates cross run contamination).

Design recommendations

`Per-case directory isolation`. Each acquisition must write all artifacts -including temp files, bugreports and backups- exclusively within the case's UUID directory. No files should be written to the current working directory or any shared location.

`Output directory reuse prevention`. If the target output directory already exists and contains prior acquisition or analysis results, the tool should refuse to proceed rather than silently reusing it.

`Device serial enforcement`. Every ADB command -including `adb backup` and `adb bugreport`- must include the `-s <serial>` flag when a serial is set for correctly acquiring those artifacts.

`Network boundary enforcement`. The `NETWORK_ACCESS_ALLOWED` flag should be checked before every outbound request, giving analysts a reliable way to run fully offline analysis. All outbound connections should be logged with their destination URL.

Analytical clarity and actionability

This section examines whether the workflow helps analysts interpret findings accurately and assign an appropriate level of confidence to them. The focus is on result clarity, confidence calibration, deduplication, summaries and other properties that shape whether analytical conclusions are correct and actionable.

Architectural finding

- No structured synthesis or prioritization of analysis results
- No deduplication of detections across modules
- Significant gap between raw forensic output and actionable understanding

Related findings in later sections

This architectural gap is reflected later in MVT-UX-C1 (Detection deduplication failure), MVT-UX-C2 (No analysis summary), MVT-UX-H1 (Mental model mismatch), MVT-UX-M2 (Error message confusion), MVT-UX-L1 (Transparent system state), MVT-UX-L2 (Automatic IOC management), AQF-UX-H2 (Generic error messages), AQF-UX-M1 (Blocking completion prompt), AQF-UX-M2 (No progress indicator), AQF-UX-M3 (Unexplained log pull features). Together, these findings show that the workflow produces detections but does not help the analyst interpret them correctly, assess their severity or communicate them to others.

Design recommendations

`Structured analysis summary`. After all modules complete, produce a summary that groups detections by severity and type, so critical findings are not buried in hundreds of routine log lines.

`Cross-module deduplication`. Detections that appear in multiple modules for the same underlying artifact should be consolidated into a single finding, with references to each contributing module, rather than repeated independently.

`Confidence levels`. Each detection should carry a confidence indicator.

`Accessible result presentation`. Even though forensic analysis is intended to be performed by highly technical practitioners, results should be structured in a way that makes it easier for the analyst to communicate high-level conclusions to non-forensic stakeholders (such as digital security trainers or HRD support coordinators), reducing the expertise barrier when acting on findings.

Security Analysis

AndroidQF security analysis

Findings by severity

Severity	Count	Identifier	Issues
Low	2	AQF-SA-L1	APK download Path Traversal in device APK paths
		AQF-SA-L2	Zip entry name Injection in APK bundle
Informational	3	AQF-SA-I1	Acquisition output permissions allow unnecessary read access and modification
		AQF-SA-I2	Serial Flag not passed to adb backup / adb bugreport
		AQF-SA-I3	Temp files written to CWD with fixed names

[Low] APK download Path Traversal in device APK paths

Summary

`modules/packages.go:getPathToLocalCopy()` constructs a local filesystem path for saving a downloaded APK by concatenating the package name with a filename extracted from the device's reported APK path. The filename extraction (`extractFileName()`) splits on the literal string `==/` and takes everything to the right. If the device returns an APK path containing `../` after `==/`, Go's `filepath.Join` resolves the traversal sequences, allowing the APK to be written outside the intended `apks/` directory on the acquisition host.

Affected Code

- Taint Source `/adb/packages.go: getPackageFiles()` stores the output of pm directly:

```
out, err := a.Shell("pm", "path", packageName)
// ...
packagePath := strings.TrimPrefix(strings.TrimSpace(line), "package:")
packageFile := PackageFile{
  Path: packagePath,  // Untrusted, straight from device}
```

- Processing (no sanitization) `/modules/packages.go: extractFileName()` splits on `\==/` but never rejects `../`:

```
func (p *Packages) extractFileName(filePath string) string {
  if strings.Contains(filePath, "==/") {
    parts := strings.Split(filePath, "==/")
    if len(parts) > 1 {
      return fmt.Sprintf("_%s", strings.Replace(parts[1], ".apk", "", 1))
      }
```

```
    } return ""
}
```

- Path construction `/modules/packages.go: getPathToLocalCopy()` joins unsanitized filename:

```
func (p *Packages) getPathToLocalCopy(packageName, filePath string) string {
  fileName := p.extractFileName(filePath)
  localPath := filepath.Join(p.ApksPath,
     fmt.Sprintf("%s%s.apk", packageName, fileName))  // no traversal check
  ...
  return localPath
}
```

- Sink (file write) `/modules/packages.go:` the actual path traversal write:

```
localPath := p.getPathToLocalCopy(packages[ip].Name, packageFile.Path)
out, err := adb.Client.Pull(packageFile.Path, localPath)  // writes to traversed path
```

Impact

Practical exploitability is limited by Android filesystem constraints. Android enforces strict package path formats under `/data/app/` and the OS does not allow apps to register package paths containing `../`. The code is technically vulnerable (no path validation in `getPathToLocalCopy()`), but the exploitation prerequisites downgrades the severity. The finding is considered as low for defense-in-depth, since the underlying code pattern (unsanitized `filepath.Join` with attacker input) should be fixed regardless.

Recommended Fix

It is recommended that whenever accepting and operating on potentially attacker controlled paths to always use `filepath.IsLocal` or `filepath.Localize` to validate or sanitize those paths.

[Low] Zip entry Name Injection in APK bundle (Zip Slip for zip consumers)

Summary

`modules/packages.go:generateZipPath()` constructs zip entry names for APKs using device controlled content. The resulting string is used as a zip entry name inside the AQF sample (encrypted output). When any third party tool such as MVT extracts this zip without zip slip protection, and as `fileName` can contain "`../`" files are written to attacker chosen paths.

This finding shares the same root cause as AQF-SA-L1 (APK download Path Traversal) the `extractFileName()` function in `modules/packages.go` passes unsanitized device controlled content through to callers without rejecting path traversal sequences. In AQF-SA-L1 the tainted output flows into a filesystem path for adb pull; here, the same tainted output flows into `generateZipPath()`, which constructs zip entry names for the encrypted AQF bundle. Since `extractFileName()` does not reject `../` sequences, the resulting zip archive can contain entry names with path traversal payloads.

Affected Code

```
modules/packages.go:
func (p *Packages) generateZipPath(packageName, filePath string) string {
    fileName := p.extractFileName(filePath)
    return fmt.Sprintf("apks/%s%s.apk", packageName, fileName)
}
```

Impact

AQF itself uses `age` encryption and writes the bundle as an opaque stream, it does not extract the zip it creates. However, when a third party (or a future MVT feature) extracts the bundle:

- Any `archive/zip` usage without entry path validation will write to the traversed path
- Python's `zipfile.extractall()` is vulnerable to zip slip
- The forensic analyst's tooling may extract the bundle before analysis

Recommended Fix

The root cause is shared with AQF-SA-L1, `extractFileName()` allows `../` sequences in its return value. Applying `filepath.IsLocal()` validation in `extractFileName()` (as recommended in 2.1.1) eliminates the tainted input for both findings simultaneously as any non-local path component is rejected before it reaches either `getPathToLocalCopy()` (Finding 2.1.1) or `generateZipPath()` (this finding).

As a defense-in-depth layer, it is also recommended to apply `filepath.IsLocal()` validation to `generateZipPath()` and `EncryptedZipWriter.CreateFile()` (`encrypted_stream.go`).

[Informational] Acquisition output permissions allow unnecessary read access and modification

Summary

In traditional unencrypted acquisition mode, AQF creates some acquisition artifacts with permissions broader than required. In particular, the case directory and subdirectories remain accessible beyond the owner, and `acquisition.json` remains world-readable after the lockdown step because it is written after the permission hardening pass. As a result, case metadata and parts of the acquisition output remain more exposed and mutable than necessary.

In the encrypted (streaming) mode, the final `.zip.age` bundle is correctly created with `0o600` permissions and intermediate plaintext is never written to disk, making this finding specific to the unencrypted acquisition flow.

Affected Code

- `acquisition/acquisition.go` creates the storage directory with `0o755`

```
acquisition/acquisition.go:
err := os.Mkdir(acq.StoragePath, 0o755)
```

- module subdirectories such as `logs/`, `apks/`, and `tmp/` are also created with `0o755`

```
modules/logs.go:
err := os.Mkdir(l.LogsPath, 0o755)
```

```
modules/packages.go:
err := os.Mkdir(p.ApksPath, 0o755)
```

```
modules/temp.go:
err := os.Mkdir(t.TempPath, 0o755)
```

- Post-acquisition lockdown: `chmods` files to `0o400` but skips directories.

```
acquisition/acquisition.go:
_ = filepath.Walk(a.StoragePath, func(filePath string, fileInfo os.FileInfo, err error)
      error {
  if err != nil {
      return err
  }
  if fileInfo.IsDir() {
      return nil    // directories never locked down
  }
  // Makes files read only
  os.Chmod(filePath, 0o400)

  sha256, err := hashes.FileSHA256(filePath)
  // ...
})
```

- `acquisition.json` is written `after` the lockdown walk, never locked down and remains at `0o644`

```
main.go:
err = acq.HashFiles()    // chmod walk runs here
// ...
acq.StoreInfo()          // acquisition.json written here, after walk

acquisition/acquisition.go:
err = os.WriteFile(infoPath, info, 0o644)    // permanently world-readable
```

Impact

This weakens confidentiality and integrity of the acquisition output on shared systems. It also leaves room for accidental modification during normal handling by the legitimate operator. `acquisition.json` never locked down: `StoreInfo()` writes `acquisition.json` at `0o644` after `HashFiles()` has already completed its chmod walk, so this file (containing device IMEI, serial, model, and timestamps) is permanently world-readable.

Recommended Fix

Create directories and files with owner only permissions from the start, and apply the final lockdown step to the complete output set, including directories and late written metadata (See more details in Architectural review and recommendations, Design Recommendations B. Cryptographic and integrity architecture). All filesystem objects should be created with restrictive permissions from the start, following the principle of least privilege (restrictive directory permissions (`0o700`) and restrictive file creation (`0o600`). The encrypted streaming mode already does this correctly (`0o600` for the `.zip.age` file) and should serve as the reference.

[Informational] Serial flag not passed to `adb backup` / `adb bugreport`

Summary

`Backup()` and `Bugreport()` in `adb/adb.go` invoke `exec.Command` directly without prepending `-s <serial>`, unlike all other ADB operations that route through `Exec()`. When multiple devices are connected via USB, these two commands return an error, so neither a Backup nor a Bugreport can be acquired.

Affected Code

`adb/adb.go` direct `exec.Command` without serial:

```
func (a *ADB) Backup(arg string) error {
    cmd := exec.Command(a.ExePath, "backup", "-nocompress", arg)
```

```
    output, err := cmd.CombinedOutput()
    // ...
}

func (a *ADB) Bugreport() error {
    cmd := exec.Command(a.ExePath, "bugreport", "bugreport.zip")
    err := cmd.Run()
    return err
}
```

Compare with `Exec()` at `adb/adb.go`, which correctly prepends `-s`:

```
func (a *ADB) Exec(args ...string) ([]byte, error) {
    if a.Serial == "" {
        return exec.Command(a.ExePath, args...).Output()
    } else {
        var params []string
        params = append(params, "-s", a.Serial)
        params = append(params, args...)
        return exec.Command(a.ExePath, params...).Output()
    }
}
```

Streaming mode handles this correctly in `streaming_buffer.go`.

Impact

When a serial is set and multiple devices are connected, `dumpsys`, `logcat`, and file pulls target the correct device (via `Exec()`), but `backup` and `bugreport` fail, producing a reduced sample that does not contain all available artifacts.

Recommended Fix

Route `Backup()` and `Bugreport()` through the same serial aware logic as `Exec()`.

[Informational] Temp files written to CWD with fixed names

Summary

In traditional mode, `adb bugreport` and `adb backup` write their output to the current working directory with hardcoded filenames (`bugreport.zip`, `backup.ab`). AQF then renames these files into the per-case UUID directory. This creates a window where temp files exist outside the case directory, and a collision point if two acquisitions run from the same CWD.

Affected Code

- `modules/bugreport.go:` bugreport written to CWD, then renamed:

```
err := adb.Client.Bugreport()
// ...
cwd, err := os.Getwd()
if err != nil {
    log.Debugf("Impossible to get current directory: %w", err)
    return err
}

origBugreportPath := filepath.Join(cwd, "bugreport.zip")
```

```
bugreportPath := filepath.Join(b.StoragePath, "bugreport.zip")
err = os.Rename(origBugreportPath, bugreportPath)
```

- `modules/backup.go:` backup written to CWD, then renamed:

```
err = adb.Client.Backup(arg)
// ...
cwd, err := os.Getwd()
// ...
origBackupPath := filepath.Join(cwd, "backup.ab")
backupPath := filepath.Join(b.StoragePath, "backup.ab")
err = os.Rename(origBackupPath, backupPath)
```

Streaming mode avoids this entirely as `StreamBugreportToZip()` and `StreamBackupToZip()` write directly to the encrypted archive without temp files.

Impact

`Sequential collision`. If a previous acquisition crashes or is interrupted, a stale `bugreport.zip` or `backup.ab` may remain in CWD. The next acquisition's `os.Rename` would silently move the stale file into the new case directory, contaminating it with artifacts from a different device.

`Temp file outside case boundary`. During the window between `adb bugreport` completing and `os.Rename` executing, the artifact exists in CWD outside the per-case UUID directory bypassing any case-level isolation.

Recommended Fix

Pass the per-case storage path directly to the ADB command so the Bugreport and Backup files (with -f flag) are written inside the case directory from the start, eliminating both the collision window and the temp file exposure.

MVT security analysis

Findings by severity

Severity	Count	Identifier	Issues
Medium	1	MVT-SA-M1	Arbitrary File write via Path Traversal in iOS decrypt-backup
Low	2	MVT-SA-L1	Arbitrary File Read via Path Traversal in iOS check-backup
		MVT-SA-L2	Arbitrary file read via Path Traversal in Bugreport main_entry.txt
Informational	2	MVT-SA-I1	Dead config flag. NETWORK_ACCESS_ALLOWED never enforced
		MVT-SA-I2	Timeline append mode generates cross run contamination

[Medium] Arbitrary file write via Path Traversal in iOS decrypt-backup

Summary

`mvt/src/mvt/ios/decrypt.py` processes encrypted iOS backups. The `file_id` field from `Manifest.db` (a SQLite database inside the backup) is used directly in path construction for both the read source and the write destination, with no validation. When `file_id` contains path traversal sequences, the decrypted file content is written to an attacker's chosen location on the analyst's filesystem.

This is an arbitrary file write on the analyst workstation during backup decryption, as the attacker controls both the destination path and the written content.

Affected Code

```
mvt/src/mvt/ios/decrypt.py:
for item in self._backup.getBackupFilesList():
   file_id = item["backupFile"]       // Taint source: Manifest.db

   try:
       source_file_path = os.path.join(
           self.backup_path, file_id[0:2], file_id   // file_id can contain ".."
       )
       if not os.path.exists(source_file_path):
           // ...
           continue

       item_folder = os.path.join(self.dest_path, file_id[0:2])  // traversal
       if not os.path.exists(item_folder):
           os.makedirs(item_folder)                   // creates traversed dir

       pool.apply_async(
           self._process_file,
           args=(..., file_id, item_folder)           // passed to writer
       )

decrypt.py: (the writer):
```

```
def _process_file(self, relative_path, domain, item, file_id, item_folder):
   self._backup.getFileDecryptedCopy(
       manifestEntry=item,
       targetName=file_id,        // traversal path as output filename
       targetFolder=item_folder   // traversed directory
   )
```

Attack Scenario

Adversary delivers a crafted encrypted iOS backup bundle to the analyst, with a Manifest.db containing for example:

```
fileID = "../../.bashrc"
```

The analyst runs `mvt-ios decrypt-backup --password ... --destination ...` which triggers the write to an attacker controlled path outside the destination directory. The attacker's encrypted payload (for example, a backdoored `.bashrc` with a reverse shell) is decrypted and written to the analyst's home directory.

Impact

This is an arbitrary file write with attacker controlled content to any path writable by the analyst process. That could be leveraged to SSH key injection (`~/.ssh/authorized_keys`), shell profile backdoor (`~/.bashrc`, `~/.zshrc`), among others.

Severity is assessed as Medium because exploitation requires a specifically crafted malicious bundle to be parsed by the analyst. There are trust mitigations between stakeholders involved in the handoff that reduce the likelihood of this scenario. Nevertheless, the impact remains Medium given the consequences if the attack succeeds.

Recommended Fix

Implement path validations to `_get_backup_file_from_id()` in `base.py` (shared sink for all iOS modules) and in `decrypt.py` source/destination path construction, using `pathlib.Path.resolve()` and `.is_relative_to()` as the native Python fix.

[Low] Arbitrary file read via Path Traversal in iOS `check-backup` (decrypted Backup)

Summary

`_get_backup_file_from_id()` in `mvt/src/mvt/ios/modules/base.py` constructs a file path from Manifest.db's `fileID` column without validation. When processing a decrypted backup via `mvt-ios check-backup`, the modules that query Manifest.db for files pass the attacker-controlled `fileID` through this function. The resolved path can point to any file on the analyst's host. The module then opens and parses the file (as SQLite or plist), and its contents flow into MVT's JSON results and CSV timeline.

This is a read traversal as the attacker controls which host file MVT opens and parses. The parsed contents are included in forensic output.

Affected Code

```
mvt/src/mvt/ios/modules/base.py:
def _get_backup_file_from_id(self, file_id: str) -> Union[str, None]:
   file_path = os.path.join(self.target_path, file_id[0:2], file_id)  // no validation
   if os.path.exists(file_path):
       return file_path   // returned to calling module, opened as SQLite or plist
   return None
```

Attack scenario

An adversary delivers a crafted iOS backup to the analyst with a Manifest.db containing a malicious `fileID`. The target module (such as `SMS`, `Calls`, `Safari`) calls `_get_backup_file_from_id()` which resolves the traversal path. The module then opens the resolved file as SQLite via `sqlite3.connect()`.

Impact

The criticality of the finding is reduced because practical exploitation requires the attacker to know or guess the analyst's directory layout for cross-case targeting, and the traversed file must be a valid SQLite database or plist whose schema matches what the specific MVT module expects.

Recommended Fix

Implement path validations to `_get_backup_file_from_id()` in `base.py` (shared sink for all iOS modules), using `pathlib.Path.resolve()` and `.is_relative_to()` as the native Python fix.

[Low] Arbitrary file read via Path Traversal in Bugreport `main_entry.txt`

Summary

`_get_dumpstate_file()` in `mvt/src/mvt/android/modules/bugreport/base.py` reads `main_entry.txt` from within the bugreport and uses its content as a filename to open a second file, with no path validation. The content of `main_entry.txt` originates from the device's `bugreport.zip` and is fully attacker controlled. An attacker who controls the device (or delivers a crafted AQF bundle) can set `main_entry.txt` to a path traversal string such as `../../../../etc/passwd`. MVT will open and read that file on the analyst's workstation without validation.

Affected Code

```
mvt/src/mvt/android/modules/bugreport/base.py:
def _get_file_content(self, file_path: str) -> bytes:
    if self.zip_archive:
        handle = self.zip_archive.open(file_path)
    else:
        handle = open(os.path.join(self.extract_path, file_path), "rb")  // no validation
    data = handle.read()
    handle.close()
    return data

def _get_dumpstate_file(self) -> bytes:
    main = self._get_files_by_pattern("main_entry.txt")
    if main:
        main_content = self._get_file_content(main[0])
        try:
            return self._get_file_content(main_content.decode().strip())  // sink
        except KeyError:
            return None
```

Impact

Severity is Informational because exploitation is constrained by several factors:

1. `Content is read but not directly included in results.` The `_get_dumpstate_file()` return value is passed to bugreport modules that parse it as a dumpstate text format. An arbitrary file (such as `/`

`etc/passwd`) does not match dumpstate syntax, so its contents are silently discarded by the parsing logic, they will not appear in MVT's JSON or CSV output.
2. `Cross-sample inclusion is theoretical but impractical.` To read another case's bugreport, the attacker would need to know the exact filesystem path of that case on the analyst workstation. Case directory names typically include UUIDs or timestamps that are unpredictable.
3. `No data exfiltration path.` The data stays on the analyst's machine in local output files.
4. `Zip mode mitigates the filesystem traversal.` When processing the bugreport as a `.zip` (the common case), `self.zip_archive.open()` raises `KeyError` for paths not in the archive. Traversal only works in extracted directory mode.

The finding is considered low because the code pattern is unsafe and a future code change could widen the impact (for example, adding raw file content to results).

Recommended Fix

Implement path validations to `_get_file_content()` in `base.py`, using `pathlib.Path.resolve()` and `.is_relative_to()` as the native Python fix.

[Informational] Dead config flag. NETWORK_ACCESS_ALLOWED never enforced

Summary

The `NETWORK_ACCESS_ALLOWED` is a dead flag. The setting is defined at `config.py` but is never referenced anywhere else in the MVT codebase. The network calls bypass this flag entirely:

1. `url.py`: `requests.head(self.url)` in `unshorten()` (triggered by device SMS/message URLs)
2. `virustotal.py: requests.get()` for VirusTotal hash lookups

None of these check `settings.NETWORK_ACCESS_ALLOWED` before connecting. The `NETWORK_TIMEOUT` setting (also in `config.py`) is similarly unused, as `unshorten()` calls `requests.head()` without a `timeout` parameter, relying on the system TCP timeout (which can hang indefinitely).

On functions when MVT intentionally resolves URLs as part of IOC analysis, there is a risk of the analyst's public IP being sent to the operator's shortener service.

Affected Code

```
/mvt/common/config.py

NETWORK_ACCESS_ALLOWED: bool = True
NETWORK_TIMEOUT: int = 15
```

Impact

So the expectation for an analyst who wants offline analysis would be to set the environment variable `NETWORK_ACCESS_ALLOWED = False`, and expect MVT to respect it and make zero outbound connections.

Currently that expectation is broken as MVT makes HTTP requests regardless. However, the severity is further reduced because the flag is not documented in MVT's user facing documentation, meaning analysts are unlikely to rely on it as a security control in current workflows.

Recommended Fix

Enforce `NETWORK_ACCESS_ALLOWED` at a single point. Create a wrapper function that all outbound requests must use and that checks the `NETWORK_ACCESS_ALLOWED` variable before conducting a network request. And replace all direct `requests.*` calls with a call to that wrapper, such as in `url.py` and `virustotal.py`. And log all outbound connections, for them to emit a `log.info()` with the destination URL so the analyst has visibility into what MVT is contacting during analysis.

Also, add a timeout enforcement for the `NETWORK_TIMEOUT` setting to be passed to every request to prevent indefinite hangs.

[Informational] Timeline append mode generates cross run contamination

Summary

`save_timeline()` opens the timeline CSV in append mode (`"a+"`). The append mode is not needed for aggregating modules within a single run: all module timelines are first collected in memory (`command.py:253` extends `self.timeline` per module), then written in a single call to `save_timeline()` at `command.py:265`. The `"a+"` only has an effect when the output file already exists from a previous run, in which case the second run's events silently append to the first, with a duplicate header row mid-file and no warning.

Affected Code

```
common/module.py: the sink:
def save_timeline(timeline: list, timeline_path: str, is_utc: bool = True) -> None:
    with open(timeline_path, "a+", encoding="utf-8") as handle:
        csvoutput = csv.writer(
            handle, delimiter=",", quotechar='"', quoting=csv.QUOTE_ALL, escapechar="\\"
        )
        // ...
        csvoutput.writerow([timestamp_header, "Plugin", "Event", "Description"])
        for event in sorted(timeline, ...):
            csvoutput.writerow([...])


common/command.py: aggregation happens in memory, "a+" is unnecessary:
# All modules aggregated in memory first:
self.timeline.extend(m.timeline)           // per module
self.timeline_detected.extend(m.timeline_detected)

# Written once at the end:
self._store_timeline()
```

Impact

- `Cross-case contamination`. If an analyst reuses an output directory for a different case, events from two devices are silently merged. An analyst may attribute device A's events to device B.
- `Re-run inflation`. Re-running the same case (such as, after adding IOCs) doubles all timeline entries with no way to distinguish originals from duplicates.
- `Silent failure`. No warning when appending to an existing file.

Recommended Fix

Replace `"a+"` with `"w"`. The timeline is fully aggregated in memory before writing, so append mode serves no purpose within a single run:

```
with open(timeline_path, "w", encoding="utf-8") as handle:
```

This alone prevents silent accumulation across re-runs. However, the broader issue, that neither AQF nor MVT prevent reuse of an existing output directory, should be addressed at the architecture level (see Architectural review and recommendations, Case isolation and network boundaries above).

Heuristic analysis

The heuristic framework

A heuristic evaluation is a method for identifying design problems in a user interface. Heuristics such as Nielsen's principles can be highly useful for helping the development team solve specific issues and develop an understanding of concepts that, in turn, empower them to improve their UX independently.

We organized the 10 heuristics into three thematic areas: foster trust and transparency, optimize user critical cognition, and craft the interaction framework.

Theme 1: Foster trust and transparency

These heuristics address how the system communicates its state, capabilities, and limitations to build appropriate user trust.

H1: Visibility of System State, Capabilities, and Confidence — The system must communicate what it can do, what it's doing, and how sure it is. Whenever working with forensic tools, status may not be simply "complete"—it's a spectrum of uncertainty. Users need to understand the tool's scope, potential for error, and confidence in any given output to calibrate their trust appropriately.
H5: Graceful Degradation and Failure Mitigation — The system must prevent common failures and handle uncertainty helpfully, not with abrupt errors. This requires proactive "cognitive guardrails" and helpful guidance when things go wrong. When the system is uncertain, it should engage the user in dialogue rather than failing silently.

Theme 2: Optimize user critical cognition

These heuristics address how well the system aligns with human cognition and augments human capabilities.

H2: Match Between System and the Shared Cognitive World — The system should align with the user's mental model and be transparent about its own limitations. A tool that constantly requires context switching or assumes expertise the user lacks adds massive cognitive load.
H6: Augment Recognition Over Recall — The system should act as an extension of the user's memory, making information and past interactions visible and accessible. This frees the user to focus on higher-level understanding rather than remembering commands or configurations.
H7: Flexibility and Efficiency of Collaboration — The system should accelerate human-tool collaboration by augmenting strengths, not just automating tasks. An effective tool offloads repetitive or cognitively taxing work, freeing the user to focus on interpretation and decision-making.

Theme 3: Craft the interaction framework (CLI)

These heuristics address the structural information architecture and aesthetic qualities of the interface itself.

H3: Calibrated Control and Autonomy — Users must feel in control by having the ability to guide, correct, and set the tool's level of independence. The key is providing the right level of control for a given context.
H4: Consistency of Interaction Patterns — While outputs may vary, the way users interact with the system should be stable and predictable. Core interaction patterns must be consistent to reduce cognitive load.
H8: Aesthetic and High Signal-to-Noise Design — Every element should serve a purpose; the interface must prioritize clarity and eliminate distracting noise. In a forensic tool, the findings are the product—the interface's job is to present them with maximum clarity.
H10: Design for Continuous Evolution and Learning — The system must be designed to learn from interaction and improve over time. The interface should facilitate feedback and make it easy for users to contribute to the tool's improvement.

Each of the 10 heuristics was evaluated for infringements and adherence, with findings documented with specific evidence from the analysis session.

Severity rating scale

- *Critical:* Prevents task completion or causes significant user frustration; recommended to be fixed immediately.
- *High:* Major usability problem that severely impacts user experience; should be prioritized.
- *Medium:* Moderate usability issue that causes confusion or inefficiency; should be addressed.
- *Low:* Minor usability concern that has minimal impact on user experience; nice to fix.
- *Informational:* Known issues.

User impact categories

- *Efficiency*: Impact on how quickly users can complete tasks.
- *Trust*: Impact on user confidence in the system.
- *Cognitive load:* Impact on mental effort required to use the system.
- *Control*: Impact on user's sense of agency and autonomy.
- *Learnability*: Impact on how easily users can understand and use the system.
- *Error prevention:* Impact on reducing mistakes and handling failures.

In the next section, we ran an analysis of MVT and AQF using this evaluation structure, while also understanding the user's case in context, which emerged from the previous threat analysis, the community survey with consensual forensic analysis practitioners, and the deep interviews with experts.

MVT heuristic analysis for usable security

In this section, we analyze the MVT command-line interface, highlight the strengths and weaknesses of its user flows, and suggest usability improvements. We rely on these findings in our experts' advice, a trained critical agent (Opus 4.5) and deep interviews with consensual forensic analysis experts.

`Task definition.` The process and methodology for consensual analysis are far from established, although some patterns emerged from our conversations with experts and the broader community's responses in the community survey (See Appendix B, Workflows). Several tasks are regularly performed by consensual forensic analysis practitioners. These tasks are mainly documented in MVT public documentation and also exposed as product features.

1. Installation and setup workflow
2. IOC (Indicator of Compromise) download process
3. Analysis execution via `mvt-android check-androidqf`
4. Terminal output during analysis
5. Generated output files (JSON, CSV timeline)
6. Command-line interface options and behavior

Findings by severity

Severity	Count	Identifier	Issues
`Critical`	2	MVT-UX-C1	Detection deduplication failure
		MVT-UX-C2	No analysis summary
`High`	2	MVT-UX-H1	Mental model mismatch
		MVT-UX-H3	Timeline data quality
`Medium`	3	MVT-UX-M1	False positives handling
		MVT-UX-M2	Error message confusion
		MVT-UX-M3	Evidence integrity and user control
`Low`	3	MVT-UX-L1	Transparent system state
		MVT-UX-L2	Automatic IOC management
		MVT-UX-L3	No user-feedback mechanism
`Informational`	1	MVT-UX-I1	CLI conventions

[Critical] Detection deduplication failure

`Heuristic Infringed`: H8 — Aesthetic and High Signal-to-Noise Design

`Summary`: A single stalkerware package (`com.systemservice` / TheTruthSpy) generates 37 identical WARNING messages across 8 different modules. The same message repeats verbatim:

```
WARNING [mvt.android.modules.bugreport.dumpsys_dbinfo] Found a known suspicious app with ID
        "com.systemservice" matching indicators from "TheTruthSpy"
```

Distribution across modules:

- 20 occurrences in DumpsysDBInfo
- 6 occurrences in DumpsysBatteryHistory
- 3 occurrences in AQFPackages (different match types)
- 1 occurrence each in 8 other modules

The 37 detections may represent `1 unique threat` found in multiple forensic artifacts. Users cannot determine from the output whether it corresponds to 1 threat or 37 different threats.

`User Impact`: *Cognitive Load, Trust, Efficiency.* This pattern creates alert fatigue through identical messages, obscures whether multiple distinct threats exist, and prevents quick severity assessment. and hides critical details (when was malware installed? what permissions does it have?).

`Recommendation`: Implement detection aggregation that shows each unique threat once with comprehensive context:

```
DETECTIONS SUMMARY
==================
[CRITICAL] TheTruthSpy stalkerware detected

  Package: com.systemservice
  Installed: 2025-03-30 14:57:02 via com.google.android.apps.nbu.files
  Certificate: CN=TheTruth (SHA1: 5e3c376b52c672c81439358de6348f25f96eaaa4)
  Found in: 8 modules (AQFPackages, DumpsysDBInfo, DumpsysBatteryHistory, ...)

  Capabilities detected:
    - Accessibility service abuse (UIAccessibilityService)
    - Device admin privileges (UIDeviceAdmin)
    - Wake lock abuse (continuous background execution)
```

[Critical] No analysis summary

`Heuristic Infringed`: H8 — Aesthetic and High Signal-to-Noise Design

`Summary`: Analysis ends with a single line that provides no actionable information:

```
WARNING [mvt] The analysis of the AQF acquisition produced 37 detections!
```

This final message fails to answer the questions users actually have:

- What were the 37 detections? (1 unique threat or 37 different threats?)
- What severity level?
- Which are the next steps? What actions are recommended?
- What is the relevant information of the case?

A 37-detection analysis generates approximately 400 lines of output where critical findings are buried among routine informational messages. The equal visual weight given to all messages makes triage difficult. A user could easily miss a critical detection among routine log entries.

`User Impact`: *Cognitive Load, Efficiency, Trust.* Users must scroll through hundreds of lines, mentally parse technical log entries, and construct their own understanding of what was found. The burden of interpretation falls entirely on the user with no assistance from the tool.

`Recommendation`: End every analysis with a structured summary block:

```
==================================================
ANALYSIS COMPLETE - THREAT DETECTED
==================================================
Device: Motorola moto g(20) (ZY32DBQN3T)
Security Patch: 2023-03-01 (OUTDATED - 34 months old)
Sample UID: 42f2629b-a0b8-4f3f-9418-b5f8718e34c7

CONFIRMED THREATS (1):
  [CRITICAL] TheTruthSpy stalkerware
    - Installed: 2025-03-30 14:57:02
    - Active: Yes (last activity: 2025-11-23 18:31:29)
    - Capabilities: Accessibility abuse, Device admin, Persistence

SUSPICIOUS FINDINGS (3):
  [MEDIUM] com.android.shell RECORD_AUDIO access (5 occurrences)
  [LOW] SipIncomingCallReceiver matches TheOneSpy pattern (likely false positive)
  [INFO] “Send security reports” Setting t disabled

NEXT STEPS:
    - Review /Users/.../test-output/
        - timeline_detected.csv (26 events)
        - aqf_packages_detected.json (threat details)
    - The sample needs an in-depth review
==================================================
```

[High] Mental model mismatch

`Heuristic Infringed`: H2 — Match Between System and the Shared Cognitive World

`Summary`: MVT assumes significant forensic expertise. The mental model required to use MVT effectively includes understanding:

- AQF data formats and acquisition methods
- STIX2 indicator file structure and provenance
- Module architecture and what each module examines
- How to interpret technical evidence like `package_verifier_user_consent = -1`
- JSON file structure for post-analysis investigation

The workflow (`download-iocs` → `check-androidqf` → interpret JSON output) is logical for forensic analysts but opaque for the emerging technologist wanting to push the field forward. The gap between the technical finding and the external sharing is substantial.

`User Impact`: *Learnability, Cognitive Load, Trust.* Users must translate forensic data into actionable understanding with no assistance. The tool provides comprehensive data and requires specific expertise to interpret it meaningfully.

Recommendation:

1. Add a `--report` flag that generates a human-readable summary (HTML/PDF) with:
    1. Plain-language explanation of what was found
    2. Risk assessment in non-technical terms
    3. Recommended actions with clear steps
    4. Links to resources for further help
2. Create an `mvt-quick` wrapper command that handles common workflows end-to-end with guided to-dos.
3. Include contextual explanations in output by STIX (e.g., "TheTruthSpy is consumer stalkerware commonly used in domestic abuse situations. It can monitor calls, messages, location, and keystrokes.").

[High] Timeline data quality issues

Heuristic Infringed: H8 — Aesthetic and High Signal-to-Noise Design

Summary: The timeline.csv contains 173,305 rows with significant data quality problems that undermine its usefulness:

Problem 1: Unix Epoch Dates

Approximately 50,000 entries display `1969-12-31 21:00:00` (UTC-3 epoch) for system files where timestamps aren't available:

```
"1969-12-31 21:00:00.000000","AQFFiles","MAC-","/sys/kernel/tracing/..."
```

These invalid timestamps create noise and could confuse users unfamiliar with Unix epoch conventions.

Problem 2: Undocumented event codes

The `Event` column uses cryptic codes (`MAC-`, `MA--`, `M---`) without documentation explaining what each code means.

User Impact: *Efficiency, Cognitive Load*. The timeline is intended to provide chronological analysis of device activity, but the data quality issues make it difficult to use without extensive filtering and external documentation.

Recommendation:

1. Handle invalid timestamps gracefully: show "N/A" or "Unknown" instead of 1969 dates.
2. Flag old dates timestamps as a key compilation warning.
3. Document event type codes in output header, separate reference file or official documentation.

[Medium] False positives handling

Heuristic Infringed: H1 — Visibility of System State, Capabilities, and Confidence

Summary: Assuming legitimate Android system components are flagged as spyware with no confidence differentiation:

```
WARNING [mvt.android.modules.bugreport.dumpsys_receivers] Found a known
 suspicious receiver with name
"com.android.phone/com.android.services.telephony.sip.SipIncomingCallReceiver"
 matching indicators from "TheOneSpy"
```

`Analysis`: `SipIncomingCallReceiver` is a stock Android telephony component present on all devices with SIP calling support. The TheOneSpy indicator file likely matches the class name pattern too broadly.

Issues with current handling:

- No confidence score accompanies the detection
- No way to distinguish high-confidence matches from pattern overlaps
- No explanation of why this matched (was it the class name? a hash? a certificate?) beyond listing the module name
- No mechanism to mark known false positives in output

`User Impact`: *Trust, Cognitive Load.* Users who understand false positives exist still cannot differentiate them from true detections. This undermines confidence in all findings.

`Recommendation`: Add confidence scoring to detections based on match type, to illustrate:

- Certificate hash match: HIGH confidence
- Package name exact match: HIGH confidence
- Receiver/service name pattern match: LOW confidence
- Annotate low-confidence matches:

```
WARNING Found potential match [LOW CONFIDENCE]:
    Component: com.android.phone/...SipIncomingCallReceiver
    Matched: receiver_name pattern from "TheOneSpy"
    Module: mvt.android.modules.bugreport.dumpsys_receivers
```

- Provide a mechanism to suppress known false positives via a local configuration file like `.config.`

[Medium] Error message confusion

`Heuristic Infringed`: H5 — Graceful Degradation and Failure Mitigation

`Summary`: Error messages conflate “data not found” with “invalid input,” causing unnecessary user concern:

```
ERROR [mvt.android.modules.bugreport.tombstones] Unable to find any
      tombstone files. Did you provide a valid bugreport archive?
```

This error appears when analyzing valid AQF data that simply doesn’t include tombstone files. The message:

1. Questions the validity of correct input
2. Doesn’t acknowledge that AQF may not include tombstones
3. Uses ERROR level for expected missing data

`User Impact`: *Trust, Cognitive Load*. Users receiving this message may believe they’ve done something wrong when the analysis is proceeding correctly. The questioning tone undermines confidence in the tool. It is relevant for the user to know that tombstones are not available in the acquisition.

`Recommendation`: Tailor messages to acquisition type and use appropriate severity levels:

```
INFO [tombstones] No tombstone files found in acquisition
```

For truly missing expected data:

```
WARNING [module] Expected data file not found: [path]
This may indicate an incomplete acquisition.
```

[Medium] Evidence integrity and user control

Heuristic: H3 — Calibrated Control and Autonomy

Summary: Users have explicit control over what gets analyzed: they provide the data path, choose output directory and select specific modules. The tool aims to never modify source data—original source data remains read-only during analysis.

User Impact: *Control, Trust.* The explicit-invocation, read-only model is appropriate for forensic tools where evidence integrity is paramount. Users maintain full agency over the analysis process.

Minor Improvement: Document that the sample is analyzed in read-only mode (perhaps in `--help` output) to build trust with users concerned about evidence integrity.

[Low] Transparent system state

Heuristic: H1 — Visibility of System State, Capabilities, and Confidence

Summary: MVT provides excellent real-time visibility into its analysis process. Each module announces when it starts running (`Running module AQFPackages...`), reports quantitative results (`Found 306 packages`), and clearly labels detections with WARNING prefixes. The tool explicitly states its limitations ("Only expert review can confirm if the detected indicators are signs of an attack") and provides resource links for further help.

User Impact: *Trust, Learnability.* Users see exactly what's happening during analysis. The explicit capability boundaries build appropriate trust calibration.

Minor Improvement: Implement severity-based output filtering (e.g., `--quiet` for warnings/errors only). The current volume is appropriate for forensic completeness but could benefit from filtering options.

[Low] Automatic IOC management

Heuristic: H6 — Augment Recognition Over Recall

Summary: MVT automatically downloads and applies 10,933 indicators from 12 different spyware campaigns without requiring users to manually source or configure IOCs. The `download-iocs` command provides persistent, updated threat intelligence. Users don't need to recall which indicators to use or where to find them.

User Impact: *Efficiency, Cognitive Load.* This significantly reduces the knowledge burden on users. The automatic management means even users unfamiliar with STIX2 or threat intelligence can benefit from current indicators.

Minor Improvement: Add `--explain-iocs` flag to describe what each indicator set covers, its source, and when it was last updated or link to the STIX2 documentation to prevent noise creation during the output.

[Low] No user-feedback mechanism

`Heuristic Infringed`: H10 — Design for Continuous Evolution and Learning

`Summary`: MVT includes automatic update checking for both tool version and indicators, demonstrating awareness of evolution. However, there's no mechanism for users to:

- Submit potential new indicators they've discovered
- Provide feedback on detection accuracy
- Contribute to indicator improvement

  In the spyware detection domain, false positives could cause unnecessary alarm; false negatives could miss genuine threats. User feedback is essential for maintaining detection quality.

`User Impact`: *Trust, Error Prevention.* Users benefit from updated indicators but cannot participate in improving detection accuracy. The feedback loop is one-directional. The tool can rely on the community for improving detention accuracy and IOCs sharing only if PII is sanitized.

`Recommendation`:

1. Add `mvt-android report-finding` command to submit potential indicators or false positives after sanitizing PII
2. Display indicator source and last-modified date in detection output so users can assess currency

[Informational] Consistent CLI conventions

`Heuristic`: H4 — Consistency of Interaction Patterns

`Summary`: MVT uses consistent CLI conventions across commands: `-o` for output, `-v` for verbose, `-h` for help. Both `mvt-ios` and `mvt-android` share similar subcommand structures (`check-*`, `download-*`). Output formatting is consistent across modules (timestamp + level + module name + message).

`User Impact`: *Learnability.* Once users learn one command, knowledge transfers to other commands. The predictable structure reduces the learning curve for the CLI interface.

`Minor Improvements`:

1. Add shell completion scripts (`mvt-android --generate-completion bash/zsh/fish`) to enhance discoverability
2. Standardize output file naming across all modules (currently inconsistent: `aqf_files.json` vs `dbinfo.json` vs `bugreport_timestamps.json`)

AndroidQF heuristic analysis for usable security

In this report, we analyze the AQF command-line interface using an Android device (Motorola Moto E5 Play) connected to a computer via USB and document the usability issues encountered during an actual extraction. We apply the same heuristic framework used for MVT evaluation, focusing on the acquisition workflow from setup through data collection.

`Task definition`: This evaluation was conducted through direct interaction with `androidqf v1.8.1`, performing an actual extraction from a physical Android device. The evaluation examined:

1. Binary download and setup process
2. ADB installation and device connection
3. Device authorization workflow
4. Extraction execution with all default modules
5. Interactive prompt handling
6. Output file generation and format
7. Integration with MVT `check-androidqf` command

Findings by severity

Severity	Count	Identifier	Issues
`Critical`	1	AQF-UX-C1	Interactive prompts fail
`High`	2	AQF-UX-H1	Silent partial completion
		AQF-UX-H2	Generic error messages
`Medium`	4	AQF-UX-M1	Blocking completion prompt
		AQF-UX-M2	No progress indicators
		AQF-UX-M3	Unexplained log pull failures
		AQF-UX-M4	Improve encryption support
`Low`	2	AQF-UX-L1	Comprehensive artifact collection
		AQF-UX-L2	Ambiguous retry behavior

[Critical] Interactive prompts fail in initial actions

`Heuristic Infringed`: H7 — Flexibility and Efficiency of Collaboration

`Summary`: AQF requires interactive arrow-key menu selection for critical options during the first action to run an extraction: 1. Backup type (SMS only / Everything / No backup) 2. APK download (All / Non-system only / None)

While this shows ability to customize the extraction, with an execution in background or the user missing the intent, these prompts fail with EOF errors:

```
Would you like to take a backup of the device?
? Backup:
  ‣ Only SMS
    Everything
    No backup
ERROR: failed to run module backup: failed to make selection for backup option: ^D

Would you like to download copies of all apps or only non-system ones?
? Download:
  ‣ All
    Only non-system packages
    Do not download any
ERROR: failed to run module packages: failed to make selection for download option: ^D
```

`Consequences of this failure:`

- Backup module skipped entirely (0 SMS data collected)
- Packages module partially failed (168 packages catalogued, 0 APKs downloaded)
- No CLI flags exist to pre-select these options
- No `--non-interactive` mode with defaults

`User Impact:` *Control, Efficiency, Security.* This failure prevents: correct understanding of what is being extracted, scripted/automated forensic workflows, integration with orchestration tools, use in CI/CD pipelines for testing.

`Recommendation:`

1. See H1, be explicit on partial completion.
2. Add `.config` or CLI flags for all interactive options:

Examples:

```
androidqf [OPTIONS]
Extraction Options:
  --backup=<sms|full|none>          Backup type (default: none in non-interactive)
  --download=<all|nonSystem|none>   APK download scope (default: nonSystem)
  --non-interactive                 Use defaults, never prompt
```

`Scripted extraction with SMS backup, no APKs`

```
androidqf --backup=sms --download=none -o ./output
```

`Full non-interactive extraction. Default to .config`

```
androidqf --non-interactive -o ./output
```

[High] Silent partial completion

`Heuristic Infringed:` H1 — Visibility of System State, Capabilities, and Confidence

`Summary:` When modules fail (as with the interactive prompts in AQF-UX-H1), extraction continues silently. The final output provides no indication of partial failure:

```
Acquisition completed.
```

Users must manually verify output contents to discover:

- `apks/` directory is empty (0 of 168 packages downloaded)
- `backup.ab` file exists empty

No `extraction_status.json` or error summary documents what succeeded vs. failed.

`Evidence from test extraction`

Expected	Actual	User prompted?
168 APKs	0 APKs	Yes, but missed user intent
SMS backup	Empty	Yes, but missed user intent
Processes	Collected	Yes
Files list	Collected	Yes

`User Impact`: *Trust, Cognitive Load.* Users cannot trust "Acquisition completed" means "Complete acquisition". Manual verification is required for every extraction.

`Recommendation`: Generate `extraction_status.json`:

```
{
  "status": "partial",
  "modules": {
    "backup": {"status": "failed", "error": "Interactive prompt received EOF"},
    "packages": {"status": "partial", "packages_found": 168, "apks_downloaded": 0},
    "processes": {"status": "success", "count": 247},
    "files": {"status": "success", "count": 89432}
  },
  "warnings": [
    "Backup module requires interactive mode or --backup flag",
    "APK download requires interactive mode or --download flag"
  ]
```

And display summary at completion:

```
Acquisition completed with warnings.

  ✓ 14 modules succeeded
  ⚠ 2 modules had issues:
    - backup: Interactive prompt on device failed (use --backup=sms)
    - packages: APK download skipped (use --download=all)

See extraction_status.json for details.
```

[High] Generic error messages

`Heuristic Infringed`: H5 — Graceful Degradation and Failure Mitigation

`Summary`: Error messages don't diagnose specific problems:

```
Unable to get device state. Please make sure it is connected and authorized. Trying again in 5
        seconds...
```

This message appeared in multiple distinct situations during testing:

1. ADB not installed (actual cause: missing dependency)
2. Device not connected (actual cause: no USB connection)
3. Device connected but unauthorized (actual cause: needs approval on phone)
4. Device in wrong USB mode (actual cause: not in file transfer mode)

The same message for all situations prevents targeted troubleshooting.

`User Impact`: *Cognitive Load, Error Prevention.* Users cannot determine which of 4+ potential causes applies. Trial-and-error troubleshooting is required.

`Recommendation`: Implement specific diagnostics:

```
Checking device connection...

✗ ADB command failed: "adb not found"
  → ADB is not installed. See installation instructions above.

✗ No devices detected
  → Connect Android device via USB cable
  → Enable USB debugging in Settings > Developer Options

✗ Device unauthorized (ZY323K5KZ9)
  → Check device screen for "Allow USB debugging?" prompt
  → Tap "Allow" (optionally check "Always allow")

✗ Device in charging-only mode
  → Pull down notification shade on device
  → Tap USB notification
  → Select "File transfer" or "MTP" mode
```

[Medium] Improve encryption support

`Heuristic infringed`: H3 — Calibrated Control and Autonomy

`Summary`: AQF supports automatic encryption of extraction output using age public-key encryption. Users place a `key.txt` file containing their age public key alongside the binary, and extractions are automatically encrypted. If a file with that exact name is not present in that exact location, it does not encrypt. If the key is malformed, AQF continues as though encryption were enabled. The output does not indicate a problem until the process finishes and an error is logged. The only early hint appears when running with `-v`, which shows the debug message: `Encrypting streaming no available, using normal mode` at startup.

Encryption error at the end of the acquisition:

```
You provided an age public key, storing the acquisition securely.
…
ERROR: Something failed while encrypting the acquisitions: failed to parse public key
        “age...”: malformed recipient “age...” mixed case
WARNING: The secure storage of the acquisition folder failed! The data is unencrypted!
```

`User Impact`: Security, Control. This addresses operational security concerns when sharing forensic data. Encrypted extractions protect sources even if the extraction media is lost or seized.

`Improvement`:

- Document encryption setup more prominently.
- Make explicit that encrypting is or not available before the acquisition starts.
- Offer using specific keys through a custom path or start a local `.config` setting.

```
STEP x of x: Sample Encryption
──────────────────────────────

Choose how to encrypt:

? Encryption method:

 ⚠ Local encrypted file (non available, review key.txt)
   Save to local disk with strong encryption

  → Local custom encrypted file
   Choose your key path and proceed

   Local unencrypted (not recommended)
   Standard local file output

[↑↓ to select, ENTER to confirm]
```

[Medium] Blocking completion prompt

`Heuristic Infringed`: H7 — Flexibility and Efficiency of Collaboration

`Finding`: After extraction completes, AQF displays:

```
Acquisition completed. Press Enter to finish ...
```

This blocking prompt prevents automated workflows from completing, requires manual intervention even for successful extractions, has no timeout or auto-dismiss option and cannot be bypassed with any flag.

`User Impact`: *Efficiency, Control*. Combined with C2 (interactive prompt failures), this makes AQF unusable in any automated context. Even successful extractions hang indefinitely waiting for the Enter key.

`Recommendation`:

1. Remove blocking prompt in `--non-interactive` mode
2. Add `--no-wait` flag to skip completion prompt
3. Or replace with auto-dismissing message:

```
Acquisition completed. Output saved to ./output/
(Auto-closing in 5 seconds, press Enter to close now)
```

[Medium] No progress indicators

`Heuristic Infringed`: H1 — Visibility of System State, Capabilities, and Confidence

`Finding`: Long-running operations display only static messages:

```
Collecting information on installed apps. This might take a while...
Generating a bugreport for the device...
Collecting list of files... This might take a while...
```

No indication of how long "a while" means, what percentage is complete, which step of how many total nor whether the process is still active or hung. The bugreport generation took several minutes with no progress feedback.

`User Impact`: *Cognitive Load, Trust.* Users cannot distinguish "working" from "hung." Uncertainty leads to premature cancellation or unnecessary waiting.

`Recommendation`: add progress indicators

```
Collecting files...     [████████          ]  52% (46,234 / ~89,400 files)
Generating bugreport... [spinning] elapsed  2:34 (typically 3-5 min)
```

Or at minimum, step counters:

```
[Module 5/16] Collecting installed packages...
[Module 6/16] Generating bugreport (this typically takes 3-5 minutes)...
```

[Medium] Unexplained log pull failures

`Heuristic Infringed`: H5 — Graceful Degradation and Failure Mitigation

`Finding`: Multiple "Failed to pull log file" errors appear without explanation:

```
Failed to pull log file /data/system/uiderrors.txt:
Failed to pull log file /proc/kmsg:
Failed to pull log file /proc/last_kmsg:
Failed to pull log file /sys/fs/pstore/console-ramoops:
Failed to pull log file /data/anr/:
```

These failures are expected but users cannot distinguish expected failures (permissions, non-rooted device), problematic failures (actual extraction errors) or impact on analysis (what forensic value is lost?).

`User Impact`: *Trust, Cognitive Load.* Error-like messages for expected conditions create uncertainty.

`Recommendation`: categorize and explain failures:

```
Log collection summary:
  ✓ logcat.txt (383 KB)
  ✓ /data/local/tmp/ (12 files)
  ○ /proc/kmsg - missing on device (vendor alike)
  ○ /data/data/ - requires root (expected on non-rooted device)

Note: Core forensic data was collected successfully. Some system logs require root access.
```

[Low] Comprehensive artifact collection

`Heuristic`: H6 — Augment Recognition Over Recall

`Finding`: AQF automatically collects 16 categories of forensic artifacts without requiring users to specify what to gather. The module list (`-l` flag) shows:

- backup, packages, getprop, dumpsys, processes, services
- bugreport, files, settings, selinux, environment info
- root_binaries, mounts, logcat, logs, temp folder

Users don't need to recall which artifacts are forensically valuable or construct complex ADB command sequences.

`User Impact`: *Efficiency, Cognitive Load*. The automated collection reduces expertise requirements and ensures consistent, comprehensive extractions.

`Minor Improvement`: Include in official documentation or add `--help-modules` to explain what each module collects and its forensic value:

```
$ androidqf --help-modules

packages    Enumerate installed applications with metadata
            Forensic value: Detect stalkerware, suspicious
            sources of installation

processes   List running processes with context
            Forensic value: Detect active malware, suspicious
            background services

dumpsys     System diagnostic information
            Forensic value: Battery drain analysis, accessibility
            service abuse, network stats, etc
```

[Low] Ambiguous retry behavior

`Heuristic Infringed`: H1 — Visibility of System State, Capabilities, and Confidence

`Finding`: When device connection fails, the tool retries silently:

```
Unable to get device state. Trying again in 5 seconds... Unable to get device state. Trying
        again in 5 seconds...
```

Users don't know how many retries will occur, whether to wait or intervene, if anything changed between retries nor when to give up. In our test, the message repeated twice before succeeding after device authorization, but the retry that succeeded looked identical to the ones that failed.

`User Impact`: *Cognitive Load*. Users must guess whether to wait or act.

`Recommendation`:

```
Waiting for device... (attempt 1/10)
  Device ZY323K5KZ9 detected but not authorized
  → Check device screen for authorization prompt
  Retrying in 5 seconds...

Waiting for device... (attempt 2/10)
  ✓ Device authorized!
```

Final thoughts

MVT and AndroidQF are the predominant open-source tools for the consensual forensic analysis community. For most practitioners, they are the default starting point of any consensual forensic mobile investigation. This report set out to examine both tools in their real-world context of use. The analysis is grounded in a threat model derived from how human rights defenders and technologists actually operate. Naturally, as open source tools, the recommendations in this report should be read as building on that foundation.

The security analysis surfaced a bounded set of concrete issues tractable and well-scoped. The heuristic analysis surfaced a recurring tension between forensic completeness and the analyst's actionable output. AQF reliably collects a comprehensive set of artifacts when its preconditions are met. However, its interactive-only flow, opaque error messages, and lack of a configuration surface create challenges. These challenges make it difficult for human rights defenders on the ground to extract, script, recover, or explain what happened. MVT produces excellent, structured data and preserves the sample it analyzes with a read-only model that protects evidentiary integrity. Yet, the CLI surface presents that data with no summary, prioritization, or guidance on next steps.

The engines underneath both tools are sound. As practitioners emerged, the need was not a different detection model or a different acquisition pipeline. It was an iteration and a refined surface on top of the existing one. For MVT, that means progressive disclosure: surfacing critical findings in plain language, providing structured summaries that help an analyst prioritize, preserving full forensic detail for expert or legal review, and restoring power-user controls for advanced workflows. For AQF, it means closing the automation gap. Consider replacing interactive-only prompts with flag-driven equivalents. Make errors explain themselves. Give defenders a configuration surface they can version and share.

Several directions for future work emerge from this report. The bundle-format contract between AQF and MVT deserves a versioned, manifest-driven specification. It should not rely on the implicit convention it currently uses. MVT could become a platform, extensible and customized via modules. This path can explore how to make the community part of the tool maintenance and development. The heuristic framework used here could be extended to other tools in the consensual forensics ecosystem. The operational guidelines in Appendix D could grow into a living community document. The consensual forensic community survey that grounded this work could also be repeated regularly. Usability regressions and new threats both move faster than any single audit can keep up with.

This report is offered as a contribution to the community that builds and sustains MVT and AQF, and as a model for how security and usability analysis can reinforce each other in tools that people depend on at the frontline.

Appendix A: Findings

Security Analysis

Identifier	Issues
AQF-SA-L1	APK download Path Traversal in device APK paths
AQF-SA-L2	Zip entry name Injection in APK bundle
AQF-SA-I1	Acquisition output permissions allow unnecessary read access and modification
AQF-SA-I2	Serial Flag not passed to adb backup / adb bugreport
AQF-SA-I3	Temp files written to CWD with fixed names
MVT-SA-M1	Arbitrary File write via Path Traversal in iOS decrypt-backup
MVT-SA-L1	Arbitrary File Read via Path Traversal in iOS check-backup
MVT-SA-L2	Arbitrary file read via Path Traversal in Bugreport main_entry.txt
MVT-SA-I1	Dead config flag. NETWORK_ACCESS_ALLOWED never enforced
MVT-SA-I2	Timeline append mode generates cross run contamination

Heuristic analysis

Identifier	Issues	Identifier	Issues
MVT-UX-C1	Detection deduplication failure	AQF-UX-C1	Interactive prompts fail
MVT-UX-C2	No analysis summary	AQF-UX-H1	Silent partial completion
MVT-UX-H1	Mental model mismatch	AQF-UX-H2	Generic error messages
MVT-UX-H3	Timeline data quality	AQF-UX-M1	Blocking completion prompt
MVT-UX-M1	False positives handling	AQF-UX-M2	No progress indicators
MVT-UX-M2	Error message confusion	AQF-UX-M3	Unexplained log pull failures
MVT-UX-M3	Evidence integrity and user control	AQF-UX-M4	Improve encryption support
MVT-UX-L1	Transparent system state	AQF-UX-L1	Comprehensive artifact collection
MVT-UX-L2	Automatic IOC management	AQF-UX-L2	Ambiguous retry behavior
MVT-UX-L3	No user-feedback mechanism		
MVT-UX-I1	CLI conventions		

Appendix B: Community survey

Consensual Digital Forensic Community Survey 2026

Understanding the practitioners who support at-risk communities through consensual device analysis and threat detection all over the globe.

We have conducted a community survey to learn about the current usage of consensual forensic analysis tools, specifically Mobile Verification Toolkit (MVT) and Android Quick Forensics (AndroidQF) by civil society organizations. The report presents findings from a survey of consensual digital forensics practitioners conducted in December 2025 - February 2026. The survey collected responses from 55 participants across multiple countries, focusing on their roles, organizational contexts, tooling preferences, and workflow processes. Responses were voluntary and anonymized. For added privacy protection, responses are shown reordered to prevent reconstruction of the full response set. Not all participants completed all questions, so sample sizes vary by section.

https://forensic-survey.0xche.org

** To be published **

Appendix C: Secure designed CLI for AQF

Based on our previous analysis, the requests from the community in the consensual forensic practitioners survey and the deep interviews with professionals, in this section we propose a redesigned CLI experience that addresses threat models where:

1. The extraction environment may be unsafe, hostile or surveilled
2. The extraction could be interrupted at any moment
3. Local disk storage could be seized or compromised
4. The user needs plausible deniability or quick cleanup options

Furthermore, we decided to prioritize the use case of non-technical operators in the territory who may need to perform an extraction without technical support nearby.

Threat model

Threat	Mitigation
Device seizure mid-extraction	Stream directly to remote, no local artifacts
Local disk forensics	RAM-only mode, secure deletion
Network surveillance	End-to-end encryption, onion routing option
Coerced disclosure	Encrypted remote retrieval (nothing local to reveal)
Time pressure	Quickmode with essential artifacts only
Interrupted extraction	Incremental upload, resumable sessions

Design rationale

Design Decision	Rationale
Step-by-step wizard	Reduces cognitive load in stressful situations
Security context selection	Adapts UI complexity to threat level
Remote extraction	Secure organization owned server
Emergency abort with cleanup	Quick escape option that preserves uploaded data
Progress with module granularity	User knows what's been saved if interrupted
Quickmode	For time-critical extractions with sensible secure defaults

Proposed CLI workflow

`Secure setup`. A pre-flight check, a clear authorization prompt in the CLI.

```
┌──────────────────────────────────────────────────────────┐
│                  androidqf - Quick Forensics             │
│                       Secure Setup                       │
└──────────────────────────────────────────────────────────┘

STEP 1 of 5: Environment Check
──────────────────────────────

✓ ADB found at /opt/homebrew/bin/adb
✓ ADB version 34.0.5 (minimum: 28.0)
✓ Platform: macOS arm64
⚠ USB debugging authorization required on device
→ Please check your device screen and tap "Allow"

[Waiting for authorization...] ████████░░

✓ Device authorized: Motorola moto_e5_play (ZY323K5KZ9)
✓ Android version: 8.0.0 (API 26) Ctrl+C to abort...
```

`Security context`. An option to define the context and customize the outcome.

```
STEP 2 of 5: Security Context
─────────────────────────────

? What is your extraction context?

→ Standard extraction
  Safe environment, local storage OK

  Sensitive extraction
  Prior encrypted output, minimize local traces

  High-risk extraction
  Hostile environment, no local storage, remote delivery only

[↑↓ to select, ENTER to confirm]
```

`Output destination`: Prioritize secure samples. Offer an interface to extract remotely.

```
STEP 3 of 5: Output Destination
───────────────────────────────
Based on your security context, choose where to send the extraction:

? Output destination:

→ Secure remote upload (recommended)
  Encrypted upload to secure server

  Local encrypted file
  Save to local disk with strong encryption

  Local unencrypted (not recommended)
  Standard local file output

[↑↓ to select, ENTER to confirm]
```

`Remote configuration`: If remote upload is selected, offer to customize remote server address and public key encryption.

```
STEP 4 of 5: Remote Upload Configuration
──────────────────────────────────────────

? Enter the server URL > secure_server.organization.onion
? Paste the server public key > age1ql3z7hjy54pw3hyww5ayyfg7zqgvc7w3j2elw8zmrj2kg5sfn9aqmcac8p

Extraction data will be:

✓ Encrypted with AGE before upload
✓ Retrievable only with organization owned private key

┌──────────────────────────────────────────────────────────────────┐
│   Sample to be sent to: secure_server.organization.onion         │
└──────────────────────────────────────────────────────────────────┘

Press ENTER to continue...
```

`Extraction options`: Explicit data collection opt-in, available time selection and settings check list before start.

```
STEP 5 of 5: Extraction Options
─────────────────────────────────

? What data should we collect?
[x] System information (getprop, settings, mounts)
[x] Running processes and services
[x] Installed packages list
[ ] Download APK files (adds ~500MB-2GB)
[x] File system listing
[x] System logs (logcat, dumpsys)
[x] Bugreport
[ ] Full device backup with SMS/MMS (limited on Android 12+)

[SPACE to toggle, ENTER when ready]

? Speed vs completeness:

  Quick scan (~3-5 min)
  Essential artifacts for IOC detection

→ Standard scan (~10-15 min)
  Comprehensive collection, recommended

  Deep scan (~30+ min)
  Everything including APKs and full backup

Estimated extraction size: ~180 MB
Estimated time: ~12 minutes
──────────────────────────────────────────────────────────────────
Ready to begin extraction with these settings:

Device:      Motorola moto_e5_play (ZY323K5KZ9)
Output:      Secure remote upload (encrypted)
Mode:        Standard scan
Local trace: None (streaming mode)

? Start extraction?
```

```
→ Yes, begin now
  No, go back and change settings
  Abort and exit [ENTER to start]
```

`Progress display (during extraction)`. Explicit progress feedback. Control options.

```
androidqf - Extracting (Streaming Mode)
═══════════════════════════════════════

Device: moto_e5_play
Elapsed: 4:32
Status: Uploading encrypted
Destination: secure_server.organization.onion Module

Progress                          Upload
──────────────────────────────────────────────────────────────

✓ getprop.txt                      23 KB   ██████████ sent
✓ settings_*.txt                   11 KB   ██████████ sent
✓ mounts.json                       6 KB   ██████████ sent
● processes.json                  187 KB   ███████▒▒▒ 67%
○ dumpsys.txt                        —     waiting
○ files.json                         —     waiting
○ logcat.txt                         —     waiting
○ bugreport.zip                      —     waiting

Total uploaded: 1.2 MB / ~180 MB estimated
Upload speed: ~2.4 MB/s (encrypted)

[Press Q to abort and cleanup | Press P to pause]
```

`Completion screen`. Summary of the extraction. If available, public link to retrieval.

```
androidqf - Extraction Complete
═══════════════════════════════

✓ All modules completed successfully
✓ Data encrypted and uploaded
✓ No local files created

──────────────────────────────────────────────────────────────

RETRIEVAL INFORMATION (save this!)
To retrieve: secure_server.organization.onion/hash
Or run:      mvt-android fetch hash

──────────────────────────────────────────────────────────────

Summary:
• 14 modules extracted
• 173 MB uploaded (encrypted)
• 168 packages catalogued
• 0 local artifacts remaining

? What now?
→ Exit (safe, no cleanup needed)
  View extraction summary
  Start another extraction
```

`Local Cleanup`. If needed, offer a local clean up tool.

```
┌──────────────────────────────────────────────────────────┐
│           ⚠ EXTRACTION COMPLETED                         │
└──────────────────────────────────────────────────────────┘
? Cleanup action:

→ Secure cleanup (recommended)
  Delete all local traces, uploaded data preserved

  Full cleanup (cannot retrieve data after this)
  Delete local AND request remote deletion

  Keep partial data
  Preserve what was collected locally

  Cleaning up...

✓ Cleared ADB cache
✓ Removed temporary files
✓ Cleared command history entry
✓ Disconnected device

Data uploaded: secure_server.organization.onion
Hash: hashcode
Size: 45 MB (modules: getprop, settings, mounts, processes)

Press ENTER to exit...
```

`Quickmode (for minimal interaction)`. For time-critical situations, support a single-command mode:

Quick extraction with remote upload, no prompts

```
androidqf --quick --remote --server=secure_server.organization.onion --
        pubkey=age1ql3z7hjy54pw3hyww5ayyfg7zqgvc7w3j2elw8zmrj2kg5sfn9aqmcac8p
```

Output:

```
androidqf Quick Mode

✓ Device: moto_e5_play
✓ Uploading encrypted stream...
✓ Complete in 3:42

Retrieval: secure_server.organization.onion/hash
```

`Suggested flags or local .config option`. A persistent local file for configurations will improve the workflow for orchestrated setups or recurrent power users. The same nomenclature can be relied on for CLI flags.

```
androidqf [OPTIONS]

Security Options:
  --context=<standard|sensitive|high-risk>
    Set security context (default: standard)

  --remote
    Stream directly to secure remote endpoint

  --remote-server=DOMAIN
    Send encrypted extraction to secure server
```

```
  --encrypt=<age|pgp|zip>
    Encryption method (default: age)

  --pubkey=FILE
    Path to recipient public key

  --pubkey-inline=KEY
    Public key string for encryption

  --no-local
    Never write to local disk (requires --remote)

  --cleanup
    Securely delete all traces after completion

  --quick
    Minimal collection, fastest extraction
```

Examples:

High-risk extraction, remote delivery, no local traces

```
androidqf --context=high-risk --server=end.eff.onion --no-local
```

Standard extraction with age encryption to local file

```
androidqf --encrypt=age --pubkey=recipient.pub -o ./output
```

Appendix D: Community operational security guidelines for Practitioners

This section consolidates operational security guidance for acquisition frontliners and forensic analysts using AQF and MVT.

Workstation preparation

`Dedicated acquisition device`. Use a single purpose device for acquisition (such as a dedicated laptop, Raspberry Pi or similar). Never use a personal or work machine that holds other sensitive data or credentials to organizational infrastructure.

`Full disk encryption`. Mandatory on both acquisition and analysis workstations. This is the primary defense against a Physical Seizure scenario. Use LUKS on Linux, FileVault on macOS or BitLocker on Windows. Ensure the machine is powered off (not just suspended) when not in use.

Per-case isolation

`Fresh working directory per case`. Never reuse output directories across cases. Verify the output directory is empty before starting acquisition or analysis.

`Manual bundle verification`. After acquisition, verify the bundle contains expected artifacts (`packages.json`, `getprop.txt`, `bugreport.zip`, `files.json`, `dumpsys.txt`, etc.). A missing or empty file could mean the collection failed silently.
Note that if the acquisition output was encrypted using a third party's public key, it is not possible to decrypt the bundle to perform this verification.

Encryption and key management

`Always encrypt during acquisition`. Place a valid age public key in `key.txt` alongside the AQF binary. Never transport plaintext bundles between acquisition and analysis stages. If encryption is unavailable, treat the plaintext bundle as highly sensitive and transfer it to the analyst via an encrypted channel, then securely delete the local copy.

`Verify encryption is active`. Use AQF with the `-v` verbose flag to verify that encryption is enabled and working correctly.

`Key separation`. Store the private key (the counterpart to the public key used as `key.txt` during acquisition) on a different physical medium than the bundle (different USB stick, different device). Never store the private key on the acquisition workstation. If the workstation is seized, the encrypted bundle is useless without the key.

If no access to plaintext logs is needed at the acquisition stage, consider using the analyst's public key for encryption to avoid holding unnecessary decryption capability. Note that this prevents the frontliner from verifying bundle contents, so collection errors may go undetected until analysis.

Acquisition hygiene

`Disable Developer Options after acquisition`. Immediately disable Developer Options on the HRD device after acquisition is complete. AQF already displays this reminder, but it is easy to overlook under time pressure.

`ADB authorization key hygiene`. When the device prompts "Allow USB debugging?", do not check "Always allow from this computer." Opting in to persistent trust stores the acquisition workstation's ADB RSA public key on the HRD device. The tradeoff is that the operator may need to be attentive to additional authorization popups on the phone during acquisition.

Sample transfer and handoff

`Encrypted transfer channel`. Transfer samples only over encrypted channels. For example, an age encrypted file sent via Signal, OnionShare or trusted third-party platforms with end-to-end encryption.

Sample integrity

`Use the -H flag`. Run MVT with `--hashes` (`-H`) to compute SHA256 hashes of the input bundle and record them in `info.json`. This creates an independent record of exactly what was analyzed.

Post-analysis data handling

`Securely delete plaintext material`. After findings are communicated, securely delete plaintext bundles, decrypted samples, and analysis output from the analysis workstation. Use `shred` on Linux or equivalent secure deletion tools. On SSDs, be sure to enable full disk encryption as secure deletion is unreliable at the file level.

`Retain only encrypted copies`. If long-term storage is needed, retain only the encrypted bundle. The analyst may not need to keep the raw bundle after producing findings.

`Document a data retention policy`. Per engagement, define: what is kept, for how long, where it is stored, and who has access. Communicate this policy to the HRD whose device was acquired for consent.

Physical seizure preparedness

`Assume seizure is possible`. The question is not "will it happen" but "what will they find." Design your operational setup so that a seized workstation yields nothing usable.

`Full disk encryption`. Mandatory on both acquisition and analysis workstations. This is the primary defense against a Physical Seizure scenario. Use LUKS on Linux, FileVault on macOS or BitLocker on Windows. Ensure the machine is powered off (not just suspended) when not in use.

`Leverage age encryption of samples`. If the bundle is age-encrypted and the private key is stored elsewhere (different USB stick, different device, with a different person), a seized workstation contains only opaque encrypted data.

Additional operational practices

`Verify AQF binary integrity before use`. AQF is distributed as a standalone binary from GitHub releases. The frontliner should verify the binary's SHA256 checksum against the published release hashes before use.

`Compartmentalize acquisition and analysis workstations`. If possible, use separate physical machines for acquisition and analysis. The acquisition workstation connects to potentially compromised devices via USB/ADB, while the analysis workstation operates only on bundle files.