Artifact Evaluation Summary

Protocol Prying: Systematic Vulnerability Research in the
Apple AirDrop and Android Quick Share Proximity Transfer Protocols

WOOT'26 — 20th USENIX WOOT Conference on Offensive Technologies

---

1 Core Idea of the Paper

The paper presents the first systematic offensive analysis of two of the most widely deployed proximity file-transfer protocols: **Apple AirDrop** (closed-source, on ∼2B Apple devices) and **Android Quick Share** (closed-source, on ∼3B Android devices, including the Google and Samsung variants). Both protocols expose a TLS / Wi-Fi-Direct attack surface to any attacker within proximity, run inside privileged system daemons, and have received almost no public security scrutiny.

We attack the closed-source-binary problem with three contributions:

1. **Reverse-engineered protocol model.** A complete seven-layer specification of AirDrop and a frame-level model of Quick Share, recovered from `sharingd` on macOS and decompiled APK/PE binaries on Android/Windows.
2. **AirFuzz**, a coverage-guided protocol fuzzer that uses Frida Stalker to obtain AFL-style edge coverage on the unmodified `sharingd` binary, combined with protocol-aware multi-layer mutators (HTTP / bplist / CPIO odc / DVZip) and AFLFast power scheduling.
3. **Six new vulnerabilities** discovered with this pipeline (V1–V6), all responsibly disclosed to Apple, Google, and Samsung.

2 Focus of the Artifact

The artifact bundles every component required to (i) reproduce the six vulnerabilities cited in the paper, (ii) re-run the AirFuzz fuzzer that discovered them, and (iii) inspect the protocol reverse-engineering results. Concretely, it contains:

- **AirFuzz** (`airfuzz/`) — the coverage-guided fuzzer described in §3 of the paper.
- **Vulnerability reproducers** (`vulnerabilities/V1 - V6/`) — one directory per CVE/finding, each with a self-contained proof-of-concept script, the captured crash logs (`.ips`), and a root-cause analysis.
- **Frida instrumentation** (`frida-scripts/`) — the live-tracing scripts used to recover the protocol structure and the auto-accept daemon used for unattended fuzzing.
- **Seed corpus** (`corpus/seed-corpus/`) — 463 serialized AirDrop inputs collected across 21 fuzzing campaigns.
- **Protocol documentation** (`protocol-docs/`) — the machine-readable specifications behind the paper's protocol figures.
- **Experiment harness** (`experiments/`) — scripts used to compute the impact-assessment numbers in Table 2.

We claim the WOOT'26 **Artifacts Available**, **Artifacts Functional**, and **Results Reproduced** badges. The Available and Functional badges are achievable on any reviewer machine; full Reproduction of V1–V3 requires a macOS target and is detailed in §4.

3 Artifact Retrieval

Archive URL:
https://drive.google.com/file/d/12DG4wNHjUbJKNB3VI3ivAfMGkZIrwTIw/view?usp=sharing

The link points to the full artifact archive on Google Drive. After downloading and extracting, the reviewer should start at the two entry points inside the archive:

```
cd artifact-evaluation/
less README.md      # entry point - full reviewer walkthrough
less CLAIMS.md      # claim-to-artifact mapping (10 claims)
```

4 What the AEC Should Check

This section maps each paper claim to the artifact element that substantiates it and gives a one-line "what to look for". The authoritative version of this mapping lives in `CLAIMS.md` inside the archive.

4.1 Environment requirements

Strictly required for Available + Functional badges: a recent Linux or macOS machine, Python 3.11+, `frida`, `numpy`. **Required to fully reproduce V1–V3:** an Apple Silicon Mac running macOS 15.7+ or 26.x with SIP disabled (Frida must attach to `sharingd`); an AWDL-capable second Apple device for the target (or two Macs). **Out of scope for live reproduction:** V4–V6 require Samsung Galaxy / Google Quick Share for Windows targets — we provide static analysis and decompilation evidence rather than a button-press repro.

4.2 Bench checks (no special hardware)

- **AirFuzz import & help.**
  `cd artifact-evaluation && python3 -m airfuzz --help`
  Expected: subcommand list `fuzz` / `replay` / `minimize` / `triage` and full option help. No Python tracebacks.
- **Offline test suite.**
  `python3 -m pytest airfuzz/tests/ -v`
  Expected: 92 tests pass (no `sudo` or Frida required).
- **Crash-log inspection.** For each of V1, V2, V3 the directory `vulnerabilities/V`x`/crash-logs/` contains the original `.ips` files referenced in the paper. Reviewers should open one and confirm the `Exception Type` and backtrace match the paper's Table 2.
- **Protocol model.** `protocol-docs/AIRDROP_PROTOCOL.md` documents the seven-layer AirDrop stack used in §2 of the paper.
- **Seed corpus integrity.** `ls corpus/seed-corpus/ | wc -l` should report 463 entries; each `.pkl` file deserializes to a structured `AirDropInput`.

4.3 Live reproduction (target Mac required)

In every snippet below, `fe80::T%awdl0` stands for the AWDL link-local address of the target Mac, obtained on that Mac via:

```
ifconfig awdl0 | grep inet6 | awk '{print $2}'
```

V1 — fatal Swift assertion (§4.1).

```
cd vulnerabilities/V1-airdrop-fatal-assertion
python3 poc_sharingd_unhandled_path.py --target fe80::T%awdl0
```

Expected: `sharingd` crashes with `SIGTRAP (EXC_BAD_INSTRUCTION)`; a fresh `.ips` appears in `~/Library/Logs/DiagnosticReports/` on the target.

V2 — XMLPlistScanner unbounded recursion (§4.2).

```
cd vulnerabilities/V2-airdrop-xml-plist-recursion
python3 reproduce_crash_002.py --target fe80::T%awdl0 --depth 200
```

Expected: `sharingd` crashes with `SIGBUS (EXC_BAD_ACCESS)`; the backtrace shows `XMLPlistScanner.scanDict` recursion exhausting the 512 KB thread stack.

V2 — Foundation isolation (no network).

```
swiftc swift-standalone/foundation_crash_decoder.swift -o fc && ./fc
```

Expected: same SIGBUS at depth 200, reached via `PropertyListDecoder` — proving the bug lives in Foundation, not in `sharingd`-specific code.

V2 — Persistent DoS / launchd throttling.

```
python3 impact_assessment.py --target fe80::T%awdl0 --count 4
```

Expected: four consecutive crashes; the `consecutiveCrashCount` field in the crash logs increments $0 \to 1 \to 2 \to 3$; `launchd` imposes a $\sim$60 s `throttleTimeout` after the third crash.

V3 — HTTP header smuggling (§4.3).

```
cd vulnerabilities/V3-airdrop-http-smuggling
python3 reproduce_crash_004.py --target fe80::T%awdl0
```

Expected: `sharingd` crashes with `SIGSEGV (EXC_BAD_ACCESS, read of NULL)` inside `nw_http_transaction / _http_parser_execute`.

4.4 End-to-end fuzzing demo (§3 / Claim 6)

```
sudo python3 -m airfuzz --mode local --max-time 300 --output /tmp/airfuzz_demo
ls /tmp/airfuzz_demo/corpus/ | wc -l   # grows from ~10 seeds to 40+
ls /tmp/airfuzz_demo/crashes/ 2>/dev/null  # may be empty in 5min, that is OK
```

What to check: corpus grows over the run (coverage feedback is producing new edges), `virgin_bits.bin` is non-empty, and the console reports per-second exec rate plus AFLFast scheduling stats. A 5-minute run is sufficient to demonstrate the loop is healthy; the paper's results used 21 multi-hour campaigns whose outputs are shipped in `corpus/seed-corpus/`.

4.5 Quick Share findings (V4–V6)

V4–V6 target closed-source Android/Windows binaries:

- `vulnerabilities/V4-quickshare-auth-bypass/README.md` — decompiled dispatcher pseudocode showing pre-auth frame handling.
- `vulnerabilities/V5-quickshare-enc-bypass/README.md` — post-handshake frames sent without the SecureMessage wrapper.
- `vulnerabilities/V6-quickshare-use-after-free/README.md` — endpoint lifecycle UAF, with Google's acknowledgement on file.

4.6 Sanity checklist for the AEC

1. Archive downloads, hash matches, extracts cleanly.
2. `python3 -m airfuzz --help` runs without traceback.
3. `pytest airfuzz/tests/` reports all green.
4. Crash-log `.ips` files for V1–V3 are present and readable, exception types match the paper's Table 2.
5. (Optional, target Mac) at least one of V1/V2/V3 reproduces a live crash within $\leq$30 seconds of running its PoC.
6. (Optional, target Mac) AirFuzz fuzzing demo produces a growing corpus over a 5-minute run.