Reverse Engineering the S2x Drone Control and Video Interfaces

Abhirama Sonny
The University of Texas at Austin
abhirama@utexas.edu

Andrew Wang
University of California, Berkeley
andrewcwang@berkeley.edu

14 November 2026

Abstract

We reverse engineered the S2x communication interface used by low-cost camera drones, tracing flight-control commands from Android applications to UDP transport and the flight-controller UART. We recovered a 20-byte control frame, its channel and flag assignments, and the checksum rule. We also characterized the fragmented-JPEG video protocol and compared the documented vendor-native parser with an open-source Python implementation. S29 serial observations show unchanged forwarding of tested control frames and a separate 14-byte optical-flow report. S20 measurements document video throughput and a five-byte UDP response whose changing field correlates with wireless signal strength. The analysis identifies a reassembly weakness in the Python client: during frame-ID rollover, it can treat an incomplete but contiguous fragment sequence as a complete frame. We provide byte layouts, test vectors, and source references, and distinguish directly checked software behavior from hardware observations preserved only in laboratory notes.

1 Introduction

Consumer camera drones commonly expose their flight controls and video feeds through vendor applications rather than documented interfaces. Reconstructing those interfaces requires more than locating an outgoing socket: application code establishes which bytes are sent, while measurements at the flight-controller boundary show what reaches the hardware.

We developed *Zwarm*, an autonomous drone-swarm project built on Marshall Richards's open-source *TurboDrone* codebase [6, 5, 7]. To support programmable control, we analyzed related Android applications, investigated the video receiver's native parser, and instrumented the Loiley S29's UART connection. We use the resulting research notes, code, tests, and measurements to characterize the Hiturbo S20 / S2x family [8].

Our contributions are (i) a reconstruction of the active Android-to-UDP control path and its 20-byte frame, (ii) a comparison of native and Python JPEG reassembly rules, and (iii) cross-layer observations relating UDP commands to UART traffic and distinguishing optical-flow reports from commands. We additionally report S20 video and return-channel measurements. We separate findings checked in source and offline tests from hardware observations for which the raw captures are no longer retained.

2 Related Work

Protocol reverse engineering recovers message formats and, where sufficient observations exist, state machines [1, 2]. Prior UAV studies have inferred proprietary flight-control formats [3] and examined other drone wireless interfaces [4]. Our work differs in tracing a single interface across Android packet construction, UDP transport, JPEG reassembly, and FC-bound serial traffic, then checking an open-source reimplementation against the documented native parser.

3 Experimental Basis

Our primary materials are the Zwarm research notes, RC and video adapters, ten existing unit tests, and the saved S20 measurement fixture [8, 10, 12, 13, 14, 15]. We compared the documented findings with the original TurboDrone project [5]. The Android analysis and S29 UART experiments were conducted during our development of Zwarm; this manuscript analyzes the records retained in the repository rather than reporting new physical experiments. Original APK hashes, native-binary disassembly files, and complete timestamped UART and video captures are not included in those materials. Consequently, hardware measurements reported below are attributed to the lab notes, whereas code behavior and arithmetic are checked separately.

4 System and Investigation Method

4.1 Targets and software corpus

The principal devices in the notes are the Hiturbo S20 and Loiley S29. Four related Android packages were inspected in the retained investigation. Comparing their call sites is necessary because common networking classes coexist with product-specific or inactive packet builders (Table 1).

Table 1: Android applications represented in the retained analysis. A missing version indicates that no specific version was recorded.

Application	Android package	Version
HiTurbo FPV	`com.vison.macrochip.hiturbo.fpv`	—
PL FPV	`com.vison.macrochip.pl.fpv`	1.1.5
REDRIE FLY	`com.vison.macrochip.inporsa.fly`	1.0.5
LOILEY FLY	`com.vison.macrochip.loiley.fly`	1.0.3

4.2 Source-tracing procedure

We traced flight-screen controls through Java packet builders and UDP transmission, then followed the video receiver into its JNI boundary. We compared those findings with the Python implementation and unit tests. The control egress path was identified through `BaseApplication.writeUDPCmd(...)` and its DHCP-gateway addressing logic. This leads to the gateway's UDP port 8080; the S20 fixture identifies 172.16.10.1 as the drone gateway and 172.16.10.2 as the client [9, 15].

Active call sites proved important. REDRIE contained both a 20-byte HY builder and an older eight-byte builder, while a referenced activity was absent from the decompiled material. In LOILEY, the flight activity called `SendHuiYuanThread` at an 80-ms interval, whereas co-located consumer classes had no recovered callers. The active thread also contained a flip flag omitted by an unused HY consumer. We therefore based the LOILEY interpretation on its reachable thread, not on unused builders [9].

The older message had the form `66 RR PP TT YY FF CK 99`. Its flag allocation is not interchangeable with the HY format. Likewise, other Macrochip-related applications include distinct frame families; shared video components do not imply shared control semantics.

4.3 Recovered component boundary

Figure 1 summarizes the observed division between the application, wireless gateway, camera board, flight controller, and sensor. The camera-board microcontroller is tentatively identified as an XR872AT; its firmware was not extracted.

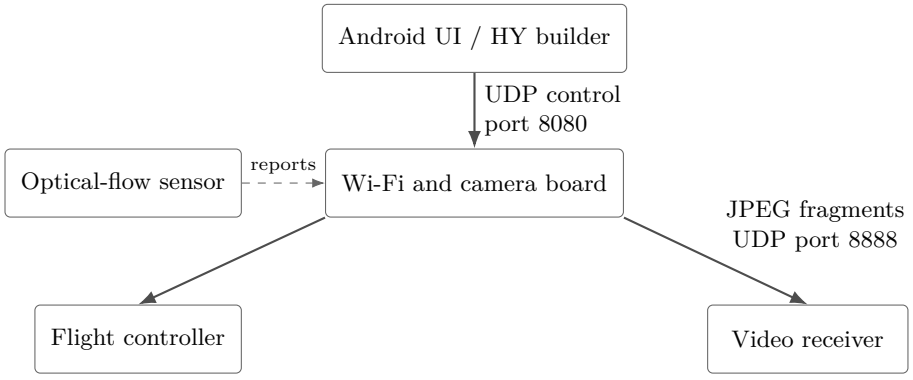


Figure 1: Logical interfaces described by the reviewed sources. The figure is not an electrical schematic. The board-to-controller link is UART at 115200 baud, 8N1. The sensor connection shown here is conceptual.

5 Control-Frame Reconstruction

5.1 Frame layout

The inspected stock HY builders emit 20-byte commands, with offsets numbered from zero. Table 2 gives the field interpretation derived from app builders and the client implementation [9, 10].

Table 2: Recovered 20-byte HY control frame.

Offset	Field	Recovered content
0	Start	Constant `0x66`
1	Framing	Constant `0x14` in inspected stock HY builders
2–5	Axis bytes	Roll, pitch, throttle, yaw, respectively
6	Command flags	Flight toggle, motor stop, calibration, variant flip
7	Mode flags	Headless, fixed app bit, recording, rocker mode
8–17	Unresolved	Zero in the inspected builders
18	Checksum	XOR of offsets 2 through 17
19	Terminator	Constant `0x99`

The axis labels originate from Java variable-to-offset tracing. The S29 notes also report independent byte sweeps at offsets 2–5 and unchanged forwarding to the UART. Those sweeps confirm position and transport preservation, but not the physical direction associated with an axis.

5.2 Model-to-wire axis conversion

The current client models each axis with center $c = 128$ and nominal range $[60, 200]$. Its conversion to an unsigned wire byte is

$$W(v) = \begin{cases} \operatorname{trunc}\left(128 + \dfrac{127(v-128)}{72}\right), & v \geq 128, \\ \operatorname{trunc}\left(\dfrac{128(v-60)}{68}\right), & v < 128. \end{cases} \tag{1}$$

Thus $W(60) = 0$, $W(128) = 128$, and $W(200) = 255$. The implementation subsequently applies an eight-bit mask; it does not saturate arbitrary out-of-range values. Equation 1 describes the *Python client*, not an inferred firmware transfer function [10, 11].

For roll and pitch, the client applies speed-tier scaling before conversion:

$$v_s = 128 + s(v - 128). \tag{2}$$

The inspected HiTurbo tiers were $s \in \{0.6, 0.8, 1.0\}$; PL FPV used $\{0.7, 0.8, 1.0\}$. Throttle and yaw bypass this scaling. Under the PL low tier, internal roll 200 and pitch 60 yield wire bytes 216 and 38, respectively, matching retained tests [9, 13]. Although the model names byte 1 as a speed field, inspected application builders leave it at `0x14`.

5.3 Command and mode bits

Table 3 summarizes the bit assignments inferred from application event handlers and masks. They describe application intent rather than independently verified flight-controller semantics.

The current adapter initializes byte 7 to `0x0A`, whereas stock builders begin with `0x02` before applying optional masks. The `0x08` flag is therefore not established as an independently decoded controller mode. Client methods `takeoff()` and `land()` also produce the same `0x01` operation: a state-dependent toggle rather than separate wire-level opcodes [9, 10, 15].

Table 3: Observed bit assignments in HY builders.

Byte	Bit	Mask	App-level meaning
6	0	`0x01`	One-key flight toggle, used for both takeoff and land
6	1	`0x02`	Emergency motor stop
6	2	`0x04`	Calibration
6	3	`0x08`	Flip/roll in inspected inline variants
7	0	`0x01`	Headless mode
7	1	`0x02`	Set by inspected stock builders; deeper meaning unresolved
7	2	`0x04`	Recording state
7	3	`0x08`	Rocker/control mode

5.4 Checksum

For a 20-byte frame P, the recovered checksum is

$$P[18] = \bigoplus_{i=2}^{17} P[i]. \tag{3}$$

A single changed byte in the covered range permits a differential check,

$$\mathrm{CK}_{\mathrm{new}} = \mathrm{CK}_{\mathrm{old}} \oplus b_{\mathrm{old}} \oplus b_{\mathrm{new}}. \tag{4}$$

The fixed markers, framing byte, checksum field, and terminator are excluded. The XOR offers error detection but provides no cryptographic authentication. Table 4 records offline examples with centered axes and the current adapter's `0x0A` mode default.

Table 4: Selected offline checksum vectors; fields not shown are zero.

Case	R	P	T	Y	Byte 6	CK
Neutral	80	80	80	80	00	0A
Flight toggle	80	80	80	80	01	0B
Stop flag	80	80	80	80	02	08
Calibration	80	80	80	80	04	0E

6 Video Transport and Native Reassembly

6.1 Session establishment

The application sends a five-byte video request consisting of `0x08` followed by the client's four IPv4 octets to gateway UDP port 8080. The client listens on UDP port 8888; the saved S20 fixture reports video datagrams originating from source port 4096. The app code repeats its request every 1000 ms, whereas the current Python adapter uses 2000 ms. The advertised address must correspond to the local interface connected to the drone network [9, 12, 15].

6.2 JNI path and fragment format

The PL FPV ARM64 split included `libvison_main.so`. Java passes received data through the JNI method `add872Stream`; the native notes associate it with an `analysis` routine that writes to an image buffer. The original binary and instruction addresses are unavailable [9].

Each image datagram has an eight-byte header, variable image payload, and two-byte trailer. The decoded layout is given in Table 5.

Table 5: Recovered UDP JPEG-fragment structure.

Offset	Width	Meaning
0–1	2 bytes	Sync marker `40 40`
2–3	2 bytes	Frame ID, unsigned little-endian
4	1 byte	Declared fragment count N
5	1 byte	Fragment index j
6–7	2 bytes	Total datagram length, little-endian
8–$L-3$	variable	Fragment payload
$L-2$–$L-1$	2 bytes	Trailer (typically `23 23`)

For datagram bytes D of length L, the decoded integer values are

$$\begin{aligned} F &= D[2] + 256D[3], \\ N &= D[4], \qquad j = D[5], \\ L_d &= D[6] + 256D[7]. \end{aligned} \tag{5}$$

The declared length must equal the received datagram length, $L_d = L$. The frame identifier is 16 bits, rather than an eight-bit counter. Its high byte is therefore necessary to avoid collisions every 256 frame identifiers.

6.3 Native and Python parser comparison

The native-parser analysis in the repository describes two frame slots, a bitmap of received indices, and a storage stride of `0x56E` bytes. That stride is an internal copy offset, not proof of a fixed transmitted payload size. The native path rejects zero counts and indices above 100, accepts out-of-order arrivals, and completes a frame once all declared indices have arrived [9].

The reviewed Python parser imposes $1 \le N \le 100$, $j < N$, and $L_d = L$. It maintains only a single current frame ID and a fragment dictionary; the first payload received for a repeated index wins. After all declared indices are present, it concatenates them in order and extracts the span between the first JPEG start-of-image marker and last end-of-image marker [12]. JPEG markers alone do not guarantee a decodable image.

Table 6: Reassembly differences established by retained notes and source.

Property	Native notes	Python client
Frame state	Two concurrent slots	One current ID
Length validation	Declared equals received	Declared equals received
Trailer treatment	Remove two bytes by length	Remove only if trailer matches
Completion	Declared-index bitmap	Declared set; rollover fallback
Reordering	Per-frame tracking	Within current frame

A frame-ID change also triggers a Python rollover fallback. That fallback accepts fragments with contiguous observed indices between their minimum and maximum before checking JPEG markers; it does not prove that all N declared fragments are present. With interleaved frames, an arrival from frame B can displace incomplete frame A, and a subsequent late fragment from A can displace B. Existing tests establish within-frame out-of-order handling, not native-equivalent two-frame interleaving [12, 14].

Constructed test case. A retained two-fragment test uses frame ID `0x1234` with indices 0 and 1. The first payload contains `FF D8` followed by `hello`; the second contains `world` followed by `FF D9`. Either arrival order produces the same marker-delimited assembled bytes. This verifies order-independent framing and concatenation, not actual JPEG decoding. Altering the first declared datagram length from 17 to 18 bytes without changing the received datagram causes rejection [14].

7 S29 UART Correlation and Sensor Reporting

7.1 Measurement arrangement

The S29 retained notes describe a receive-only observation of the Wi-Fi board's TX wire toward the flight controller. The signal was connected to an Arduino Mega2560 RX1 input (pin 19) with shared ground. Serial decoding used 115200 baud, 8N1, and records were divided by an approximately 5-ms idle gap. USB records were exported at 2,000,000 baud. Color labels recorded for the tested unit are not claimed to be a universal connector pinout [9].

7.2 Bridge observations

The notes report a distinctive, non-neutral four-axis signature with an identical 20-byte HY frame appearing on the FC-bound UART about 17 ms after its UDP transmission. Separate axis sweeps covered values from `0x00` through `0xFF` at offsets 2–5 with the other channels centered. Tested command masks `0x01`, `0x02`, and `0x04`, as well as selected values of bytes 1 and 7, were also forwarded unchanged [9].

The tested control bytes reached the FC-bound wire unchanged; this does not establish that the controller executed each command. The 17-ms observation is not a latency distribution.

When the UDP controller ceased transmitting, the observed HY frames ceased while separate `77 0E` reports continued. Thus the camera board did not continue replaying the last HY command in that test. Any controller timeout or resulting flight behavior was not measured in the available notes.

7.3 Independent optical-flow report

A separate 14-byte UART report has the structure

$$\underbrace{\texttt{77 0E}}_{\text{header}}\ \underbrace{\texttt{XL XH YL YH QQ}}_{\text{fields}}\ \underbrace{\texttt{00 00 00 00 00}}_{\text{reserved}}\ \texttt{CK 99}. \tag{6}$$

Its first two data fields are signed 16-bit little-endian integers. For the first field, let $u_x = D[2] + 256D[3]$; then

$$x = \begin{cases} u_x - 65536, & u_x \geq 32768, \\ u_x, & u_x < 32768. \end{cases} \tag{7}$$

The second field at offsets 4–5 is interpreted analogously; checksum byte 12 is the XOR of offsets 2–11. The retained example `77 0E 56 FE B0 FF FF 00 00 00 00 00 18 99` decodes to $x = -426$, $y = -80$, and byte 6 equal to `0xFF` [9].

The notes report approximately 22 messages per second, with an interval near 45 ms. Motion above a textured floor changed the two signed fields and byte 6; a stationary condition returned them to zero without altering commanded HY axes. This supports a separate motion-sensor path, but sign conventions, units, scale, and the meaning of byte 6 remain unknown.

8 Validation and Performance

The repository includes four control tests and six video/transport tests covering axis conversion, speed tiers, command-flag repetition, fragment ordering, length rejection, and loopback request handling [13, 14]. We inspected their assertions and independently checked the packet arithmetic; a complete test-run transcript is not archived. These tests validate client logic rather than aircraft behavior.

Table 7 reproduces measurements from the saved S20 fixture dated 5 November 2026 [15]. They are properties of that capture rather than universal S2x specifications.

The current backend constructs control frames at 80 Hz; inspected HiTurbo and PL FPV loops used 50-ms intervals, while the reachable LOILEY thread used 80 ms. Client-side repetition of a stop flag and one-frame clearing of other transient flags are software design decisions, not firmware guarantees [9, 10, 11].

8.1 S20 return-channel observations

The Zwarm S2x research notes also report a drone-to-client response accompanying datagrams addressed to UDP port 8080 [8]. In the retained November 8 capture description, response datagrams follow the five-byte

Table 7: Retained S20 measurement-fixture observations.

Metric	Recorded value
Network addresses	Gateway 172.16.10.1; client 172.16.10.2
Ping round-trip time	3.175 ms minimum; 11.679 ms mean; 27.152 ms maximum
Decoded JPEG resolution	1184 × 656 pixels
Advertised resolution	640 × 360 pixels
JPEG frame size	34,041–34,357 bytes; median 34,197 bytes
Frame rate	15.8–17.9 frames/s; typical 16.9 frames/s
Receive data rate	533–568 KiB/s
Video source UDP port	4096

shape `FF 53 54 08 XX`, with the final octet changing with wireless-link conditions. The notes record 6,965 responses for 6,987 transmitted control packets (a response-to-transmission ratio of 99.69% within that reported observation). This ratio is *not* an end-to-end reliability estimate: the original series, capture-loss accounting, and correspondence rule are not archived in the inspected material.

A two-minute walk observation in the notes associates the changing byte with signal strength; it reports a correlation of $r = 0.84$ with received Wi-Fi signal and an approximate affine relationship between the two quantities [8]. This suggests a link-quality field rather than battery status. The raw paired samples are unavailable, so the reported correlation cannot be independently assessed. The S20 fixture also records UDP replies and a status-reply prefix [15].

This UDP return path is distinct from the S29 optical-flow UART reports.

8.2 Parser conformance boundary

Existing tests verify length rejection and reassembly of reversed fragments within one frame [14]. They do not cover cross-frame interleaving, rollover with missing initial fragments, or JPEG decoding after marker extraction. The latter cases are important because the Python fallback checks contiguous *observed* indices rather than the full declared set. A replay corpus would establish how often these differences affect decoded video.

9 Discussion and Limitations

Source-level field assignments describe application intent, not necessarily flight-controller behavior. The S29 serial tests establish forwarding for selected inputs, but not motor response, control timeout behavior, or interpretation of unused bits. The 17-ms forwarding delay is a single recorded observation. Likewise, the optical-flow rate and S20 signal correlation are reported in the notes without raw sample series or confidence intervals. Results should not be generalized across firmware versions or related product families without testing.

The open-source client and retained notes are sufficient to reconstruct the control and video layouts and expose the Python reassembly discrepancy. Stronger evaluation would require versioned app binaries, paired timestamped UDP/UART captures, raw S20 video fragments, and the samples underlying the return-channel correlation. A fragment replay corpus could quantify failures from interleaved frames and incomplete rollover. The recovered XOR checksum is not authentication, and this work does not establish a security exploit.

10 Conclusion

We reconstructed the S2x control-frame layout and JPEG transport, correlated selected UDP commands with FC-bound UART traffic, and identified an independent optical-flow report. Comparing the documented native video parser with the Python client exposed a concrete reassembly mismatch during frame-ID rollover. The additional S20 return-channel observation suggests a link-quality field. Together these findings characterize the observed interface and identify where further captures are needed to validate controller semantics and parser behavior.

Acknowledgments and Availability

We developed Zwarm and carried out the Android analysis and S29 UART experiments described here. Zwarm builds on Marshall Richards's original TurboDrone project, whose authors and contributors retain credit for the upstream work [5, 7, 6]. The code, tests, and retained research notes are publicly available in the cited repositories; original APK binaries and complete hardware capture archives are not included.

References

[1] J. Narayan, S. K. Shukla, and T. C. Clancy, "A Survey of Automatic Protocol Reverse Engineering Tools," *ACM Computing Surveys*, vol. 48, no. 3, Art. 40, Dec. 2015. doi: 10.1145/2840724.

[2] Y. Huang, H. Shu, F. Kang, and Y. Guang, "Protocol Reverse-Engineering Methods and Tools: A Survey," *Computer Communications*, vol. 182, pp. 238–254, Jan. 2022. doi: 10.1016/j.comcom.2021.11.009.

[3] R. Ji, J. Wang, C. Tang, and R. Li, "Automatic Reverse Engineering of Private Flight Control Protocols of UAVs," *Security and Communication Networks*, vol. 2017, Art. 1308045, 2017. doi: 10.1155/2017/1308045.

[4] D. Pratama *et al.*, "Behind the Wings: The Case of Reverse Engineering and Drone Hijacking in DJI Enhanced Wi-Fi Protocol," in *Proc. IEEE PlatCon*, 2024, pp. 127–132. doi: 10.1109/PlatCon63925.2024.10830741.

[5] M. Richards and contributors, "TurboDrone," original open-source project, https://github.com/marshallrichards/turbodrone, accessed 14 Nov. 2026.

[6] A. Wang and contributors, "Zwarm," open-source code and research repository, https://github.com/andrewwbuilds/zwarm, accessed 14 Nov. 2026.

[7] Zwarm contributors, "NOTICE," https://github.com/andrewwbuilds/zwarm/blob/main/NOTICE, accessed 14 Nov. 2026.

[8] Zwarm contributors, "S2x Research Notes," https://github.com/andrewwbuilds/zwarm/blob/main/docs/research/S2x.md, accessed 14 Nov. 2026.

[9] TurboDrone and Zwarm contributors, "S2x Research Notes," https://github.com/marshallrichards/turbodrone/blob/master/docs/research/S2x.md; see also [8]. Accessed 14 Nov. 2026.

[10] Zwarm contributors, "S2x RC Protocol Adapter," https://github.com/andrewwbuilds/zwarm/blob/main/backend/protocols/s2x_rc_protocol_adapter.py, accessed 14 Nov. 2026.

[11] Zwarm contributors, "S2x RC Model," https://github.com/andrewwbuilds/zwarm/blob/main/backend/models/s2x_rc.py, accessed 14 Nov. 2026.

[12] Zwarm contributors, "S2x Video Protocol Adapter and Model," https://github.com/andrewwbuilds/zwarm/tree/main/backend, relevant paths `protocols/s2x_video_protocol.py` and `models/s2x_video_model.py`, accessed 14 Nov. 2026.

[13] Zwarm contributors, "S2x RC Protocol Tests," https://github.com/andrewwbuilds/zwarm/blob/main/backend/tests/test_s2x_protocol.py, accessed 14 Nov. 2026.

[14] Zwarm contributors, "S2x Video Protocol Tests," https://github.com/andrewwbuilds/zwarm/blob/main/backend/tests/test_s2x_video_protocol.py, accessed 14 Nov. 2026.

[15] Zwarm contributors, "S20 Measurement Fixture," https://github.com/andrewwbuilds/zwarm/blob/main/backend/tests/fixtures/s20_capture.json, measurements labeled 5 Nov. 2026; accessed 14 Nov. 2026.

[16] Zwarm contributors, "Experimental S2x Notes," https://github.com/andrewwbuilds/zwarm/blob/main/experimental/s2x/README.md, accessed 14 Nov. 2026.

A Offline Reproduction Notes

The source tree includes tests for command encoding and video reassembly. The following command runs the two offline test modules in a prepared repository environment:

```
cd backend
.venv/bin/python -m unittest tests.test_s2x_protocol \
    tests.test_s2x_video_protocol -v
```