Post-Quantum TLS and Bot Detection

Do browser-impersonation tools reproduce the post-quantum ClientHello?

Pallavi Srinivasappa | M.S. Computer Information Systems (Cybersecurity), Boston University | pallavii@bu.edu

Code and data: `https://github.com/Pallavii-dev/pq-bot-fingerprint` | 3 November 2026, version 2.0

Summary

I measured whether the libraries that bots use to impersonate browsers reproduce the post-quantum ClientHello that Chrome, Firefox and Safari now send. Between 28 and 30 October 2026 I captured 288 browser disguises from eight impersonation tools on a local TLS server, and compared each one with the real browser version it claims to be: Chrome 124–156, Edge 156, Firefox 128–159 and Safari 26–27. Twenty of the 166 Chrome disguises (12%) got the post-quantum part wrong for the version they claim. A repeat run on 2 November 2026 found 42 of 214. The strongest signal was a single bit: whether the X25519 key inside the hybrid key share is the same key as the separate X25519 share. Real Chrome and Safari never reuse it, real Firefox 132 and later always does, and seven of the eight tools apply one behaviour to every browser they imitate. A rule built on that bit flagged 46 disguises and none of 78 real-browser profiles in October, and 72 disguises and none of 75 real-browser profiles in November.

1 The problem

Bot management relies heavily on TLS fingerprinting. The first message of a TLS 1.3 handshake, the ClientHello, differs in small but stable ways between browser implementations. If a client says it is Chrome in its User-Agent but its ClientHello does not look like Chrome's, that is suspicious. Impersonation libraries such as curl_cffi, tls-client and uTLS exist to close exactly that gap: they copy a browser's ClientHello byte for byte so that a script looks like a browser.

Post-quantum key agreement made that copying harder. The `X25519MLKEM768` key share alone is 1,216 bytes: a 1,184-byte ML-KEM-768 encapsulation key (FIPS 203) followed by a 32-byte X25519 key. Adopting it added more than a kilobyte to the ClientHello and brought new values into existing extensions: a new group in `supported_groups` and `key_share`, ML-DSA codepoints (FIPS 204) in `signature_algorithms`, and, in recent Chrome, a GREASE value at the front of that list. Every one of these is a new place where an imitation can get something wrong.

The common fingerprints do not see most of it. JA3 includes the `supported_groups` list but not the contents of the key share. JA4 encodes neither. So an impersonator can carry a post-quantum mistake inside the key share and still match a browser's JA3 or JA4 exactly. I could not find a published measurement of whether impersonation tools reproduce the post-quantum ClientHello correctly, or of whether the differences can catch bots without also flagging real browsers. The project answers three questions:

1. Is the post-quantum key share now one of the strongest bot signals?
2. Which tools reproduce it correctly, and how quickly do they follow browser changes?
3. Can a ClientHello that matches Chrome's fingerprint still give itself away through post-quantum details?

2 Why I did this project

Most public post-quantum measurement looks at servers: which sites and CDNs negotiate ML-KEM, and how fast that share is growing. I wanted to look at the other side of the handshake, at what clients send, and ask whether post-quantum adoption had accidentally created a new way to tell real browsers from bots. It seemed likely, because a browser gets these details for free from its TLS library, while an impersonation tool has to copy each one by hand and keep up with every release. I deliberately did not build my own ClientHello forger. The offensive side of the study uses only existing, publicly available tools, so the results describe what bot operators can already download.

3 How I did it

- **Capture.** A local TLS server reads each client's raw ClientHello from the socket before completing the handshake, then finishes TLS and records the HTTP/2 layer: SETTINGS, WINDOW_UPDATE, PRIORITY and pseudo-header order (the Akamai HTTP/2 fingerprint format), plus header order and the User-Agent.
- **Validation.** For every hybrid key share I check that the ML-KEM part is a structurally valid ML-KEM-768 encapsulation key: each 12-bit coefficient must be below the modulus $q = 3329$, the FIPS 203 input check. A tool that padded the slot with random bytes would fail it.
- **Features.** From each ClientHello I extract the post-quantum key share (group, order, length), whether the X25519 key is reused inside the hybrid share, ML-DSA signature algorithms, GREASE placement, JA4 and the HTTP/2 fingerprint. JA4 is computed as in FoxIO's reference implementation, which drops GREASE values from every list, including `signature_algorithms`.
- **References.** I measured real browsers on the platforms where they would run reliably (Table 1). Older Chrome came from official Chrome for Testing builds on GitHub Actions, because endpoint-security software on my Windows machine blocked the unsigned builds.
- **Scoring.** Each disguise is judged against the measured behaviour of the exact browser version it claims (from its User-Agent, or its profile name when it sends none), not against the newest browser. Comparing a Chrome 120 profile with Chrome 156 mostly measures how old the profile is.
- **Repetition.** Every client was captured three times over fresh connections, and browsers started with a fresh profile each time. A monthly automated run repeats the whole measurement with the newest tool releases and the newest stable browsers.

Two counts need explaining. Of the 288 disguises, 227 could be scored: all 166 Chrome disguises, 44 of 52 Firefox disguises and 17 of 70 Safari disguises. The other 61 claim a version I had no real browser for: 8 claim Firefox 102–123 (I measured from 128) and 53 claim Safari 14–18 (I measured 26 and 27). The 78 real-browser profiles are 33 Chrome for Testing

Table 1: Data sources for the October run (28–30 October 2026).

Set	Contents	Platform
Disguises	288 profiles that claim Chrome, Edge or Opera (166), Firefox (52) or Safari (70), from curl_cffi 0.17.0, tls-client 2.0.0, primp 2.0.1, impit 0.14.7, CycleTLS 2.0.5, got-scraping 4.2.2, rnet 2.4.2 and 3.0.0rc22, and uTLS built from commit `ff1b50f`	Windows, loopback
Chrome, Edge	Installed Chrome 156 and Edge 156, fresh profiles (8 and 14 hellos); headless and Playwright runs	Windows
Chrome history	Chrome for Testing 124–156, one build per milestone, 2–6 hellos each	Linux
Firefox	Official release builds 128–159, 2–3 hellos each	Linux
Safari	Safari 26 on macOS 14, 15 and 26; Safari 27 on macOS 15	macOS

milestones, 7 Windows Chromium runs (installed Chrome and Edge, headless Chrome and Edge, and three Playwright runs), 32 Firefox releases and 6 Safari captures. The November run had no Windows machine, so its 75 are 34 Chrome for Testing captures (Chrome 156 in two builds), 32 Firefox releases and 9 Safari captures.

4 What I found

4.1 Most post-quantum errors are stale profiles, but not all

Compared naively with Chrome 156, nearly every Chrome disguise differs in some post-quantum detail. That comparison is misleading: 150 of the 166 claim a Chrome older than 150. Judged against the version each one claims, 20 of 166 (12%) are post-quantum-incorrect, and 15 of the 81 that claim Chrome 131 or newer (19%). The November run, which captured every profile each tool offers rather than a fixed list, found 42 of 214. In October the newest named Chrome profile of every tool sent a valid `X25519MLKEM768` share (CycleTLS has no named profiles; it is configured with a fingerprint string), and every hybrid share passed the ML-KEM-768 key check. Measuring Chrome milestone by milestone confirmed the public timeline (the Kyber draft from 124, ML-KEM from 131, ML-DSA signature algorithms from 150) and showed one change I found no public note of: from Chrome 152, a GREASE value sits at the front of `signature_algorithms`.

4.2 X25519 reuse is the sharpest signal

A client may put the same X25519 public key inside the hybrid share and in its separate x25519 share, or generate two keys. The hybrid-design draft allows both, so this is a habit of the TLS library, not a protocol requirement. The habit is consistent within each browser family and wrong in most tools (Table 2). Safari's post-quantum support depends on macOS rather than on the Safari version, and its User-Agent freezes the macOS version at `10_15_7`, so a Safari 26 client with no post-quantum share is normal and is not flagged.

Table 2: Measured X25519 reuse inside the hybrid key share (October run).

Client	What was measured	X25519 reused
Chrome, Edge	Chrome 124–156 (Linux); Chrome and Edge 156 (Windows)	never
Firefox	132–159; 128–131 send no post-quantum share	always
Safari	26 on macOS 26; 26 on macOS 14 and 15, and 27 on macOS 15, send no post-quantum share	never
curl_cffi, tls-client, got-scraping, rnet	every profile with a hybrid share	never
primp, impit	every profile with a hybrid share	always
uTLS	Chrome, Firefox and Safari profiles	as the real browser
CycleTLS	sends no hybrid share	not applicable

4.3 A JA4 match is not enough

CycleTLS, configured with real Chrome 156's JA4_r string, produced a JA4 identical to real Chrome's. It still sent no post-quantum key share, and listed x25519 twice in `supported_groups`, apparently in place of the ML-KEM group it does not implement. A JA4 allowlist accepts this client; its HTTP/2 fingerprint gives it away. The X25519 habit of primp and impit is likewise invisible to JA3 and JA4. Smaller leaks: primp sends a 3-byte ALPS value no real browser sends, impit sends the literal string `impit` as its ALPS protocol, and 50 of the 166 Chrome disguises keep the same extension order on every connection, while real Chrome shuffles it each time.

4.4 The tools lag behind Chrome

From the release histories, curl_cffi published its first Chrome 131 (ML-KEM) profile 17 days after Chrome 131 reached Stable, and tls-client after 7 days. The lag grew afterwards: 39 and 64 days for Chrome 150, and 57 and 8 days for Chrome 152. On 3 November 2026, the day Chrome 157 reached Stable, the newest releases of both tools (curl_cffi 0.17.0, and tls-client's Go core 1.16.0) still stop at Chrome 152, five milestones behind; Chrome 153 has been stable for 56 days. I have not yet measured Chrome 157 itself.

5 Detection rules and the single-connection detector

Each rule checks whether the TLS and HTTP behaviour is consistent with the browser and version named in the User-Agent. Table 3 lists the rules that matter for this report. R4 to R6 are left out: R4 (an ML-DSA signature algorithm expected for the claimed version but missing) had no hits once judged by version, and R5 (an ALPS value never seen from a real browser; 18 hits in October, 47 in November) and R6 (a Chromium-claiming client whose extension order is identical across connections, while real Chrome shuffles it every time; 50 and 94 hits) do not involve post-quantum fields.

R6 also needs at least two connections from the same client. R10 is the X25519 rule; its expected behaviour for each family is learned from the measured browsers rather than typed in.

Table 3: Rule hits on Chrome disguises: 166 in the October run, 214 in the November run.

Rule	Fires when the client	Oct	Nov
R1	claims Chrome 131 or newer but sends no `X25519MLKEM768` share	4	3
R2	claims Chromium and sends no `X25519MLKEM768` share (catches stale profiles; not version-aware)	88	103
R3	sends a hybrid share whose ML-KEM part is not a valid ML-KEM-768 key	0	0
R7	matches real Chrome or Edge on JA4 but not on the HTTP/2 fingerprint	4	0
R8	claims Chrome 152 or newer but has no GREASE value in `signature_algorithms`	5	7
R9	claims Chromium and reuses the X25519 key inside the hybrid share	12	36
R10	reuses, or does not reuse, X25519 against the habit of the family it claims	12	36

Across all three families R10 flagged 46 disguises in October (12 Chrome, 31 Firefox, 3 Safari) and none of the 78 real-browser profiles; in November it flagged 72 (36, 33 and 3) and none of 75. All rules together flagged 110 of the 166 Chrome disguises and 161 of the 227 scored disguises in October, and 155 of 214 and 210 of 283 in November (all 214 Chrome, 52 of 67 Firefox and 17 of 82 Safari disguises could be scored). No rule fired on the Windows Chromium runs (installed Chrome and Edge 156, headless Chrome and Edge, Playwright); the November run had no Windows machine, so that check rests on October. The rules flag impersonation libraries, not real or automated browsers. The disguises that passed every TLS check were rnet's Chrome 131–145 profiles, curl_cffi chrome131–151 and uTLS chrome_131 and chrome_133. uTLS's Chrome 155 and Safari 26.3 hellos were also TLS-perfect and were caught only at the HTTP/2 layer, because my test client pairs uTLS with Go's standard HTTP/2 code.

The same rules run as a single-connection detector that returns one of four verdicts: looks real, likely disguise, outdated browser or old disguise (only R2 fired), or unknown (no browser in the User-Agent, or no measured reference for that version). Replaying one connection per client for the first six tools, every real and automated Chromium run came out as *looks real*, and the 103 disguises split into 39 likely disguise, 14 outdated, 10 looks real and 40 unknown. Most of the unknowns sent no User-Agent naming a browser, so there was nothing to compare against.

6 Limitations

- Disguises were captured on one Windows machine over loopback. Chrome history and Firefox were measured only on Linux, Safari only as versions 26 and 27, and iOS profiles were compared with macOS Safari.
- The Windows Chrome and Edge reference is small (22 hellos). Zero false positives there, or on 78 real-browser profiles, is not a false-positive rate; enterprise policies, extensions, proxies and middleboxes were not tested.
- Chrome for Testing runs without staged feature rollouts, so installed Chrome 152–155 may differ.
- Percentages count the profiles a tool offers, each weighted equally. They say nothing about which profiles bots use in practice.

7 Conclusion and next steps

The post-quantum key share is a strong bot signal only against stale profiles: the newest profiles of every tool send a valid hybrid share. The more durable signals sit in details that JA3 and JA4 do not encode, above all X25519 reuse, whose correct value depends on the browser family the client claims. Next I want to check installed Chrome 152–155 against the Chrome for Testing results, measure Chrome 157, older Safari and real iOS devices, and measure the catch-up lag again when curl_cffi and tls-client ship Chrome 153+ profiles.