PQ Bot Fingerprint

Do bot tools copy the post-quantum part of a browser's handshake correctly?

Plain-English notes on what I built, how each piece works, and what I found.
The short answer: mostly yes for new profiles, but one small habit, X25519 reuse, gives most of them away.

Author Pallavi Srinivasappa, M.S. Computer Information Systems (Cybersecurity), Boston University
Code and data https://github.com/Pallavii-dev/pq-bot-fingerprint
Measurements October run 28–30 Oct 2026; repeat run 2 Nov 2026

1 The idea in one paragraph

Websites try to tell real browsers apart from bots. One of the oldest tricks is to look at the very first message a client sends when it opens an HTTPS connection. Chrome, Firefox and Safari each write that message in their own slightly different way, so if a client *says* it is Chrome but its first message does not look like Chrome's, it is probably a script pretending. Bot tools know this, so they copy a real browser's first message byte for byte. Over the last two years browsers added post-quantum cryptography to that first message, which made it more than a kilobyte bigger and added new details. My question was simple: did the bot tools copy those new details correctly, and if not, can we use the mistakes to catch them without ever flagging a real browser?

Words I use. Hello: one captured ClientHello. **Disguise**: one browser profile from an impersonation tool, like curl_cffi's "chrome131". **Stale profile**: a disguise copying a browser version that is now old, which is not the same as a wrong one. **Hybrid share**: the X25519MLKEM768 key share, an ML-KEM key and an X25519 key glued together.

2 Background I need first

2.1 What the ClientHello is

Every HTTPS connection starts with a TLS handshake, and the client always speaks first. That first message is the **ClientHello**. It is basically the client saying "here is everything I can do, pick something". It is a list of lists, and the parts that matter for this project are in Table 1.

Table 1: The parts of the ClientHello I care about.

Part	What it is, in plain words
Cipher suites	Which encryption combinations the client can use.
Extensions	Optional add-ons, each with a 2-byte type and its own data. Chrome shuffles their order on every connection.
supported_groups	The key-exchange methods the client supports, in order of preference.
key_share	The client's actual public keys for the methods it guesses the server will pick, sent up front to save a round trip.
signature_algorithms	Which signature types the client accepts on the server's certificate.
GREASE	Random "fake" values browsers sprinkle into lists on purpose, so servers don't break when real new values show up later.
ALPN / ALPS	Which application protocol to speak next (like HTTP/2), plus a Chrome-specific settings extension.

2.2 How bot detection uses it

A detector compares two things the client said: the browser named in its `User-Agent` header, and the shape of its ClientHello. To compare shapes quickly, people turn the ClientHello into a short fingerprint string. **JA3** glues together the version, cipher list, extension list, supported groups and point formats in the order sent, then hashes it. Because Chrome shuffles extensions, Chrome gets a new JA3 on almost every connection. **JA4** fixes that by sorting the ciphers and extensions before hashing, keeping signature algorithms in order, and throwing away GREASE values, so it stays stable. The important gap for me: **JA3 includes the supported_groups list but not what is inside the key share, and JA4 includes neither.** Anything wrong inside the key share is invisible to both. There is also an **HTTP/2 fingerprint** (the Akamai format), built from what the client sends right after TLS: its SETTINGS values, its WINDOW_UPDATE size, PRIORITY frames and the order of the pseudo-headers like `:method` and `:path`.

2.3 What post-quantum changed

Classic key exchange uses X25519, a 32-byte public key that a future quantum computer could break. So browsers moved to a **hybrid**: `X25519MLKEM768` (code `0x11EC`). Its key share is two keys glued together: a 1,184-byte ML-KEM-768 encapsulation key (the post-quantum part, from FIPS 203) followed by a normal 32-byte X25519 key, 1,216 bytes in total. Browsers still also send a plain X25519 share as a fallback, in case the server doesn't speak post-quantum. Chrome went through stages, and I confirmed each one by measuring every milestone: the older Kyber draft from Chrome 124, ML-KEM from 131, ML-DSA codes (`0x0904`–`0x0906`, FIPS 204) in signature_algorithms from 150, and from 152 a GREASE value

placed at the very front of signature_algorithms. That last one I could not find written down anywhere public.

2.4 The X25519 reuse habit, explained simply

Look at Figure 1. The last 32 bytes of the hybrid share are an X25519 public key, and the separate fallback share is also an X25519 public key. A client can either make *one* X25519 key and put it in both places, or make *two* different keys. The standards allow both, so it is purely a habit of the TLS library. Chrome and Safari always make two keys. Firefox (version 132 and later) always reuses one. Checking it is almost too easy: compare the last 32 bytes of the hybrid share with the 32 bytes of the plain X25519 share. Same bytes means reused. That one yes/no answer turned out to be my best signal.

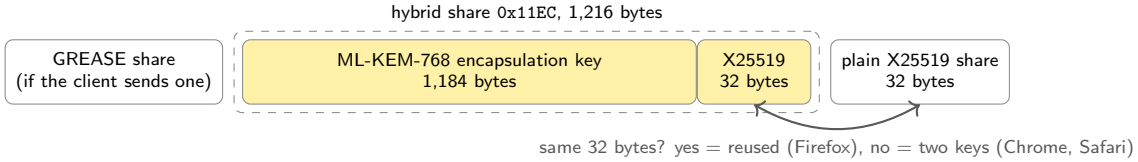

Figure 1: Inside the key_share extension. The comparison under the boxes is the whole X25519 reuse check.

3 Why I did this

Most public post-quantum measurement looks at servers: which sites support ML-KEM and how fast that number grows. I wanted to look at the other side of the handshake, the clients. My hunch was that post-quantum adoption had accidentally made impersonation harder. A real browser gets all these new details for free from its TLS library. A bot tool has to copy each one by hand and keep up with every release, so every new field is one more chance to get something wrong. I also decided early not to build my own forger. Everything on the attacking side comes from tools anyone can download, so the results describe what bot operators really have today.

4 How the whole thing works

Figure 2 is the full pipeline; the steps below follow the data left to right.

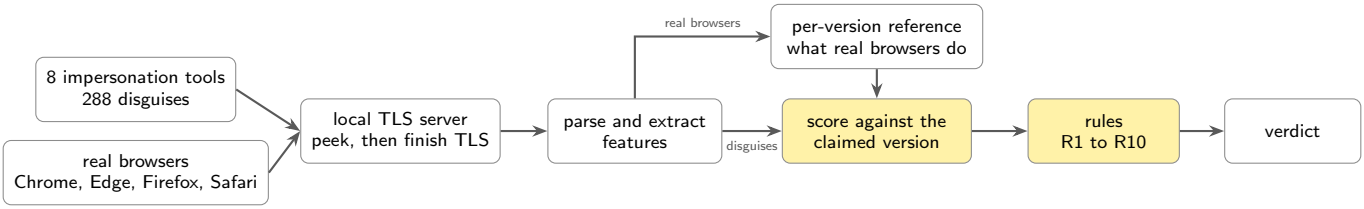

Figure 2: The pipeline. Yellow boxes are where a disguise actually gets judged.

4.1 Step 1: capture

I run a small TLS server on my own machine, on loopback. When a client connects, the server first **peeks** at the raw bytes on the socket (the `MSG_PEEK` flag reads data without removing it). That matters: I get an exact copy of the ClientHello, but the bytes are still there for the normal TLS library to read, so the handshake can finish like nothing happened. I need it to finish, because the HTTP/2 fingerprint only exists *after* TLS, when the client starts sending HTTP/2 frames. So the server completes TLS and records the HTTP/2 SETTINGS, WINDOW_UPDATE, PRIORITY, pseudo-header order, normal header order and the User-Agent. Safari needed a test certificate authority it trusted, otherwise it refuses to continue. Every disguise was captured three times over fresh connections, which is how I can tell whether extension order changes between connections.

4.2 Step 2: parsing

A ClientHello is just bytes in a fixed layout, so parsing is walking it in order: a 5-byte record header, a 4-byte handshake header, the version, 32 random bytes, the session ID, the cipher list, compression methods, and then the extensions. Each extension is a 2-byte type, a 2-byte length and its data. Inside key_share, each entry is again a 2-byte group code, a 2-byte length and the key itself. From that I pull out the features in Table 2.

Table 2: Features I extract from every hello.

Feature	How I get it
PQ key share	Which hybrid group is offered (Kyber draft or ML-KEM), its position and its length.
ML-KEM key valid?	The check in Step 3.
X25519 reused?	Last 32 bytes of the hybrid share compared with the plain X25519 share.
ML-DSA support	Whether `0x0904`–`0x0906` appear in signature_algorithms.
GREASE placement	Where GREASE values sit in each list, including the front of signature_algorithms.
Extension order	Whether the order changes across the three connections (real Chrome always shuffles).
JA4 + HTTP/2	JA4 computed the standard way (GREASE dropped everywhere) and the Akamai-format HTTP/2 string.

4.3 Step 3: is the ML-KEM key even real?

A lazy tool could fill the 1,184-byte slot with random bytes and hope nobody looks. So I check the key the way FIPS 203 says a receiver should. The first 1,152 bytes of an ML-KEM-768 key are 768 numbers packed 12 bits each, and every one of them must be smaller than $q = 3329$. The last 32 bytes are a seed with no rule. A random 12-bit number passes with probability $3329/4096 \approx 0.81$, so all 768 passing by luck has a chance of about 10^{-69}. In practice: random bytes always fail. In my data every hybrid share passed, so no tool was faking the key; their mistakes were elsewhere.

4.4 Step 4: the reference, meaning what real browsers actually do

I can only call something wrong if I know what right looks like for that exact version. So I measured real browsers myself instead of trusting memory. Older Chrome came from official Chrome for Testing builds, one per milestone from 124 to 156, run on GitHub Actions (my Windows laptop's endpoint security blocked the unsigned builds). Firefox release builds 128–159 ran headless on Linux. Safari 26 ran on macOS 14, 15 and 26, and Safari 27 on macOS 15. On Windows I captured installed Chrome 156 and Edge 156 with fresh profiles, plus headless and Playwright-driven runs. That gave 78 real-browser profiles in October: 33 Chrome milestones, 7 Windows runs, 32 Firefox releases and 6 Safari captures.

Two browser facts I had to respect. Safari's post-quantum support comes from macOS, not from the Safari version, and Safari's User-Agent always pretends to be macOS 10.15.7, so a Safari 26 client with no post-quantum share is *normal* and must not be flagged. And Firefox 128–131 sends no post-quantum share at all.

4.5 Step 5: scoring against the version it claims

This is the step that keeps the results honest. If I compared every disguise with the newest Chrome, almost all of them would look wrong, but mostly because the profiles are old. A Chrome 120 profile with no ML-KEM is *correct*, because real Chrome 120 had none. So for each disguise I read which browser and version it claims (from the User-Agent, or from the profile name if it sends no User-Agent), look up what that real version does, and only count differences from *that*. Disguises claiming versions I never measured (Firefox 102–123, Safari 14–18) are left unscored rather than guessed. That is why 227 of the 288 were scored in October.

4.6 Step 6: the rules

Each rule is one consistency check between "what you claim to be" and "what you sent". Table 3 has them in plain words.

Table 3: The detection rules and their hits on Chrome disguises, October / November (R5 is counted across all disguises).

Rule	Fires when the client...	Oct / Nov
R1	claims Chrome 131+ but sends no ML-KEM hybrid share	4 / 3
R2	claims Chromium and sends no ML-KEM share at all (the "old profile" rule, not version-aware)	88 / 103
R3	sends a hybrid share whose ML-KEM key fails the Step 3 check	0 / 0
R4	claims a version that should advertise ML-DSA but doesn't	0 / 0
R5	sends an ALPS value no real browser sends (not post-quantum)	18 / 47
R6	claims Chromium but keeps the same extension order on every connection	50 / 94
R7	matches real Chrome/Edge on JA4 but not on HTTP/2	4 / 0
R8	claims Chrome 152+ but has no GREASE at the front of signature_algorithms	5 / 7
R9	claims Chromium and reuses the X25519 key	12 / 36
R10	reuses, or doesn't reuse, X25519 against the habit of the family it claims	12 / 36

R10 is my favourite because it is not hardcoded. It *learns* the right answer for each browser family from the real browsers I measured. If the real browsers in a family ever disagreed with each other, the rule would switch itself off for that family instead of guessing. R6 is different from the rest: it needs at least two connections from the same client, so it can never decide a single-connection verdict.

4.7 Step 7: a detector you can point at one connection

The same rules run as a small detector that listens on `https://localhost:8443/`, replays a saved capture, or takes one ClientHello plus its User-Agent, and gives one of four verdicts:

- **unknown**: the User-Agent names no browser, or I have no measured reference for that version, so there is nothing to compare.
- **outdated browser or old disguise**: only R2 fired, so it could just be someone with an old browser.
- **likely disguise**: any other rule fired, and it lists which ones.
- **looks real**: nothing contradicts what it claims.

4.8 Step 8: keeping it fresh

Browsers now ship every couple of weeks, so a snapshot goes stale fast. A monthly GitHub Actions run repeats everything with the newest tools and browsers, and I read each tool's release history to measure how many days it took to ship a profile after a Chrome version went stable.

4.9 One example, start to finish

Say a script uses impit with a Chrome profile. Its User-Agent says Chrome, so Step 5 looks up what that Chrome version really sends. Step 1 captures the hello three times, and Step 2 finds a valid `0x11EC` hybrid share, so R1, R2 and R3 stay quiet: the post-quantum part *exists* and the ML-KEM key passes the Step 3 check. But the last 32 bytes of the hybrid share are identical to the plain X25519 share. Real Chrome never does that, so R9 fires, and R10 fires too because reuse goes against the Chrome family's habit. The detector says **likely disguise** and lists R9 and R10. Now flip it: a curl_cffi profile that claims to be a recent Firefox sends two different X25519 keys. To JA3 and JA4 that looks fine, but real Firefox 132+ always reuses, so R10 fires. Same check, opposite direction, which is exactly why the rule has to know which family the client claims. And a real recent Firefox? It reuses the key, as Firefox always does, so nothing fires and it **looks real**.

5 What I found

Table 4: The main numbers.

	October	2 November
Chrome disguises that get the post-quantum part wrong for their claimed version	20 of 166 (12%)	42 of 214
. . . of those claiming Chrome 131 or newer	15 of 81 (19%)	–
Disguises flagged by the X25519 rule (R10), all families	46	72
Real-browser profiles flagged by R10	0 of 78	0 of 75
All rules together, scored disguises	161 of 227	210 of 283

Most mistakes are just old profiles. 150 of the 166 Chrome disguises claim a Chrome older than 150, so judging by version matters a lot. The newest named Chrome profile of every tool sent a valid ML-KEM share, and no tool faked the key.

X25519 reuse is the sharpest signal. Real browsers are consistent within each family, and the tools are consistent within themselves, which is exactly the problem. curl_cffi, tls-client, got-scraping and rnet never reuse, which is wrong whenever they pretend to be Firefox. primp and impit always reuse, which is wrong whenever they pretend to be Chrome or Safari. Only uTLS does it per family, like real browsers.

A perfect JA4 match is not enough. CycleTLS, set up with real Chrome 156's JA4 string, produced a JA4 identical to Chrome's. But it sent no post-quantum share at all and listed x25519 twice in supported_groups, apparently filling the slot of the ML-KEM group it doesn't implement. A JA4 allowlist lets it in; its HTTP/2 fingerprint gives it away. Smaller leaks: primp's 3-byte ALPS value, impit sending the word `impit` as its ALPS protocol, and 50 of the 166 Chrome disguises never shuffling their extensions.

The tools are falling behind. curl_cffi shipped its first Chrome 131 profile 17 days after it went stable, and tls-client after 7. For Chrome 150 it was 39 and 64 days. On 3 November, the day Chrome 157 went stable, both still stopped at Chrome 152, five versions behind.

Who passed everything. rnet's Chrome 131–145 profiles, curl_cffi chrome131–151 and uTLS chrome_131/133 passed every TLS check. uTLS's Chrome 155 and Safari 26.3 hellos were also TLS-perfect, caught only at HTTP/2 because my test client pairs uTLS with Go's standard HTTP/2 code. No rule ever fired on installed, headless or Playwright-driven Chrome and Edge, so the rules catch impersonation libraries, not automated real browsers.

6 What I'm careful not to claim

- These are tool *profiles*, each counted once. They say nothing about which profiles real bots actually use.
- Zero false positives on 78 real profiles is not a false-positive rate. I never tested enterprise policies, extensions, proxies or middleboxes, and Chrome for Testing has no staged rollouts, so installed Chrome 152–155 may differ.
- One Windows machine over loopback for disguises; Chrome history and Firefox only on Linux; Safari only 26 and 27; iOS approximated by macOS Safari.

7 Questions people ask me

Isn't this a guide for bots? Not really. Everything I found is visible to anyone running these tools, I built no forger, and it helps defenders more: a detector can add the X25519 check today, while each tool must fix every browser it copies.

Why does the X25519 check work so well? It is free for browsers and costly for imitators: a browser gets it right from its TLS library, but a tool must know the answer *depends on which browser it pretends to be*, and most picked one answer.

Why measure browsers myself? "What Chrome 140 sends" isn't written down anywhere in full, and measuring also found the Chrome 152 GREASE change.

What's next? Compare installed Chrome 152–155 with the Chrome for Testing results, measure Chrome 157, older Safari and real iOS devices, and repeat the catch-up measurement once curl_cffi and tls-client ship Chrome 153+ profiles.