Measuring TLS key exchange with post-quantum KEM

Krzysztof Kwiatkowski[1], Nick Sullivan[1], Adam Langley[2], Dave Levin[3], Alan Mislove[4]
[1]Cloudflare, [2]Google, [3]University of Maryland, [4]Northeastern University

NIST is in the process of selecting new post-quantum cryptographic algorithms that are secure against both quantum (PQ) and classical computers. NIST has selected a few candidates from among all submissions for further consideration and study.

Our goal is to understand how these algorithms act when used by *real* clients over *real* networks, particularly candidate algorithms with a significant difference in public-key or ciphertext sizes. Our focus is on how different key sizes affect handshake time in the context of Transport Layer Security (TLS) as used on the web in HTTPS. Our two primary candidates are NTRU-based HRSS and isogeny-based SIKE. The following table shows a few characteristics for both algorithms. Performance characteristics are from running the BoringSSL speed test on an Intel Skylake CPU.

KEX	Public Key size (bytes)	Ciphertext size (bytes)	Secret size (bytes)	KeyGen (op/sec)	Encaps (op/sec)	Decaps (op/sec)	NIST level
HRSS (commit 78c88c9)	1138	1138	32	3952.3	76034.7	21905.8	1
SIKE/p434	330	346	16	367.1	228.0	209.28	1

Note that HRSS shows a significant speed advantage, while SIKE has a size advantage. We will deploy these two algorithms on both the server side (using Cloudflare's infrastructure) and the client side (using Chrome Canary); both sides will collect telemetry information about TLS handshakes using these two PQ algorithms to see how they perform in practice.

In 2018, Adam Langley conducted an experiment with the goal of evaluating the likely latency impact of a post-quantum key-exchange in TLS. Chrome was augmented with the ability to include a dummy, arbitrarily-sized extension in the TLS `ClientHello` (fixed number of bytes of random noise). As expected, he observed that using PQ schemes with small keys confers a network latency benefit on roughly 5% of network devices (if only performance can be improved). However, Langley also observed a peculiar phenomenon: 5% of the clients experienced *much* higher increases in latency than expected. We are working on performing a wide-scale experiment combining client- and server-side data collection to (1) evaluate the performance of the key exchange algorithms on real users' devices, and (2) more

thoroughly evaluate and ascribe root causes to these unexpected latency increases. In particular, we would like to learn more about the characteristics of those networks, what are the causes of increased latency, and how the performance cost of isogeny-based algorithms impacts the TLS handshake.

The key questions we want to address are as follows:

- How does key size and running time of the PQ algorithm influence client and network performance? (This is why we chose to evaluate HRSS and SIKE; they represent algorithms with very different tradeoffs.)
- How do clients with different OSes, architectures, and network connectivity respond when switching to new PQ algorithms?
- How do network middleboxes behave when clients use new PQ algorithms and which networks are problematic middleboxes deployed in?
- What is a good ratio for speed-to-key size (or how much faster could SIKE get to achieve the client-perceived performance of HRSS)?
- What "unknowns" exist in the deployment of PQ algorithms? Such as problems that may appear while migrating TLS to post-quantum key exchange
- How do different properties of the client's network affect the performance of TLS with different post-quantum key exchanges? Can we identify specific autonomous systems, device configurations, local network configurations that favor one algorithm or another? How is performance affected in the long tail?
- Are there any properties that can be determined by the client ahead of making a connection that allow it to choose one algorithm over another?

Experiment Design

Below, we describe a large-scale experiment involving both server- and client-side data collection from real users on real networks around the world.

Server side (Cloudflare)

Cloudflare is operating the server side of the TLS connections. We will enable the CECPQ2 (HRSS + x25519) and CECPQ2b (SIKE + x25519) key agreement algorithms on all TLS-terminating edge servers (see TLS Integration). In this experiment, the `ClientHello` will contain a CECPQ2 or CECPQ2b public key.

Cloudflare is in an advantageous position when it comes to analyzing TLS traffic coming from a variety of sources. Thanks to Cloudflare's highly efficient request logging system we are able to collect details of clients connecting to our network. This data can be used for analysis and post-processing. We will be recording details like TLS handshake duration, as well as details about TLS handshake size, client IP,

HTTP User-Agent header (containing details like browser version, OS, CPU) and device type (mobile or desktop). Unlike Langley's original experiment, we now have full implementations of the key exchange algorithms at both client and server; it is therefore possible that we will observe slowdowns from different client types. Correlating handshake times with such per-client data may help to reveal implementation issues related to particular device types.

Since Cloudflare only measures the server side of the connection, it is impossible to determine the time it takes for a `ClientHello` sent from Chrome to reach Cloudflare's edge servers; however, we can measure the time it takes for the TLS `ServerHello` message, which contains the ciphertext encapsulated with a post-quantum KEM, to reach the client and for the client to respond. This ciphertext has different sizes depending on the key-exchange algorithm used. For SIKE, the ciphertext is 24 bytes longer than the public key; for HRSS, public key and ciphertext are exactly the same size. By measuring the time between TLS `ServerHello` and the resulting `Finished` message from the client, we should get an idea about the time it takes a handshake to complete on the client side and the network-induced delay.

Handshake duration will be calculated on the server side. The calculated value is the time between when the TLS layer starts to offload TLS (processing `ClientHello`) and the TLS handshake successfully completes. We will also calculate the time between the server sending TLS `ServerHello` and receiving TLS `Finished` message.

One of our primary sets of questions involves why some clients appear to experience a much greater increase in latency under these new schemes than with traditional key exchange. We hypothesize that this is partly due to two in-network phenomena: middleboxes and buffer-bloated wireless links. As described below, we cannot perform additional active measurements from the client-side. To infer whether clients are subject to middleboxes or lossy/bloated links, we will collect additional data server-side: we will collect the clients' IP addresses and autonomous system (AS) numbers, and will use them to perform subsequent active measurements (such as traceroutes) to identify shared network infrastructure among clients with large increases to latency. We describe these active measurements more below.

Client side (Chrome)

Chrome is operating the client side of the TLS connection. We will consider two CPU architectures: x86-64 and aarch64. Chrome will enable CECPQ2 and CECPQ2b for the following mix of architecture and OSes:

- x86-64: Windows, Linux, macOS, ChromeOS
- aarch64: Android

This choice of platforms is dictated by the performance of SIKE, which heavily depends on the capabilities of the platform. To be applicable in real-world implementations, we use an instruction only found in modern CPUs. Note too, that implementation of SIKE in BoringSSL will run faster on Intel Broadwell microarchitecture because we can take advantage of optimization techniques as described by Faz-Hernández et al..

Chrome will measure the latency of the entire TLS handshake. Chrome clients should be grouped into the following sets: {SIKE, HRSS and control (key exchange with classical cryptography)} x {x86-64, aarch64}. This gives us 6 groups. It is important to make sure sets of servers in each group to be constant, in order to get consistent and useful results.

It is important to note that this will be running on *real users'* browsers, as part of Chrome Canary. As a result, we cannot collect personally identifying information. Thus, Chrome will provide histograms of aggregate measures, only for those handshakes which have used post-quantum key exchange. We augment these client-side data aggregates with server-side measurements, described above, and active measurements, described next.

Follow-up active measurements

Our high-level expectation is to get similar results as Langley's original experiment in 2018—that is, speedups for the 5th percentile and slowdowns for the 95th. Unfortunately, data collected purely from real users' connections may not suffice for diagnosing the root causes of why some clients experience excessive slowdowns. To this end, we will perform follow-up experiments based on per-client information we collect server-side.

Our primary hypothesis is that the excessive slowdowns, like the ones Langley observed, are largely due to in-network events, such as middleboxes or bloated/lossy links. As a first-pass analysis, we will investigate whether the slowed-down clients share common network features, like common ASes, common transit networks, common link types, and so on. To determine this, we will run traceroute from vantage points close to our servers back towards the clients. (We will adhere to standard ethical guidelines of not overloading any particular links or hosts in our active measurements, as well as apply standard blacklists like those employed by scans.io).

If we are able to identify seemingly problematic networks, we will seek to obtain machines within those networks from which we can run our own client-side experiments. From machines we control, we can run more intensive experiments, such as TTL-limited probes to identify where middleboxes might be located; varying packet sizes to study hop-by-hop maximum transmission units (MTUs) in the network; and studying whether some client locations are subject to slowdowns for *all* destinations, or just some.

Data collection summary

The set of metrics we will collect includes:

- transfer latency
- processing time for TLS `ClientHello` on server side
- processing time for the TLS handshake on client side
- latency and processing time in go-keyless

- processing time for client authentication, if such was requested
- latency and processing time for the `Finished` message
- client IP address and AS number, to be used to guide follow-up active measurements, in an effort to identify link type and the presence of middleboxes.

Compared to Langley's original experiment in 2018, we add server-side data collection and follow-up active measurements. As a result, we expect to be able to identify some of the root causes of the slowdowns and new opportunities for optimization. Additionally, our experiment includes full implementations of the key exchange algorithms at both client- and server-side, allowing us to evaluate at scale—on real users' devices—the performance of key exchange algorithms. Taken together, our experiments will shed light on the real-world impact that devices and networks have on these newly proposed PQ key exchange algorithms.

TLS Integration

We will implement two key exchange algorithms and add them to BoringSSL. These are hybrid schemes, meaning that they mix classical and PQ algorithms for improved security. We concatenate output from X25519 with output from the post-quantum key exchange. The integration is similar to the design suggested in i.a. draft-stebila-tls-hybrid-design-00 (see points 3.3.1 and 3.4.1), some SIDH specific details can also be found in draft-kiefer-tls-ecdhe-sidh-00.

In brief:

- The scheme combines two key exchange mechanisms: classical X25519 with either quantum-resistant SIKE (CECPQ2b) or NTRU-HRSS (CECPQ2)
- Support for the algorithm is advertised by including the scheme's ID in the supported_groups extension and including the post-quantum and X25519 public keys in the key_share extension of `ClientHello`
- The scheme applies to TLS 1.3 only

In the TLS `ClientHello`, BoringSSL sends a public key for one of the post-quantum schemes. Additionally, it will always include an X25519 pubic key in order to avoid an additional round trip caused by servers not supporting either CECPQ2 or CECPQ2b.

Implementation of our algorithms will differ slightly from the original proposals. Namely, we will use HMAC-SHA256 instead of SHA-3 based constructions. This change should not affect the overall performance.