of security has only a very minor impact on computational performance, but it does have a significant impact on signature size. The draft of the NIST standard also considers parameter sets targeting security level 3 (equivalent to AES-192); the simple modification is to truncate all hashes to 192 bits. The extension to a parameter set targeting NIST level 1 is straight-forward: hashes are simply truncated to 128 bits; we obtain this by using SHAKE-128 [82] with 128 bits of output.

We define $\text{XMSS}_s^{\text{MT}}$ as an instantiation of XMSS^{MT} using two trees of height 12 each, i.e., a total tree height of 24, which limits the maximum number of signatures per public key to $2^{24} \approx 16.7$ M. Increasing this maximum number of signatures to, for example, $2^{30} \approx 1$ billion increases signature size by only 96 bytes and has negligible impact on verification speed. It does have an impact on key-generation speed and signing latency, but as mentioned in Section 6.3, latency of signing is not very relevant when used by certificate authorities as in our paper.

Multi-tree XMSS is particularly well-suited for efficient batch signing. The idea is to compute one whole tree (of height h/d) on the lowest level and use it on-the-fly to sign $2^{h/d}$ messages. The computational effort per signature is then essentially reduced to one WOTS$^+$ key-pair generation.

We set the Winternitz parameter in $\text{XMSS}_s^{\text{MT}}$ to $w = 256$ to optimize for signature size. Changing to the more common $w = 16$ would increase signature size by about a factor of 2 and speed up verification by about a factor of 8.

E NOTES ON INTERACTIONS WITH TCP

The TLS protocol is layered on top of the TCP transport layer protocol. This means that optimizations and settings that apply to TCP have an effect on the measured behavior. We do not mean to give an exhaustive analysis of these and their interplay with KEMTLS. However, we did see some behavior that will be relevant to anyone trying to reproduce our results in new implementations.

Nagle's algorithm [22, 81] is a congestion-control algorithm that is enabled by default on most systems. It attempts to solve the problem of large streams of TCP packets being sent out, where each packet is very small. It does this by waiting to send undersized TCP packets, where the size is less than the maximum segment size, until all the sent-out data has been acknowledged. Disabling Nagle's algorithm, for example by setting the `TCP_NODELAY` flag on a socket, will mean TCP packets get sent out immediately as TLS messages are written to the socket.

However, this leads to a second interaction with the TCP slow start congestion-control algorithm [17, 22]. This algorithm specifies an initial congestion window size (`initcwnd`). This is the number of packets that can be sent out before receiving an acknowledgement. As more acknowledgements get received, the window increases, but this is not very relevant during the short lifetime of the TLS and KEMTLS handshakes. The default window size is set to 10 on current Linux kernels. This is large enough for most of our algorithm choices to complete the handshake before running into the maximum window size. However, this is only true if they only send out (roughly) one TCP packet per message "flow", sending messages that follow each other at the same time. As an example of this, `ServerHello`, `EncryptedExtensions` and `ServerCertificate` can be sent in the same TCP packet, size of the certificate permitting. Disabling Nagle's algorithm would, for naive implementations, lead to all of the handshake messages being sent separately. This quickly runs into the slow start algorithm, which introduces full round-trip delays.

We have found that these effects do not always show up. It seems implementation strategies, such as asynchronous I/O or synchronous I/O, can also have a great effect on exactly how the messages are picked up. However, we suggest implementors to consider using vectored I/O, such as the `writev` system call. These allow to write multiple TLS messages to the socket at the same time, allowing them to be sent in the same TCP packet. Alternatively, consider explicitly controlling when the socket submits packets to the network, for example by using the `TCP_CORK` mechanism in the Linux kernel [77].

During our experiments, we saw such interactions only with the Kyber and Dilithium KEMTLS instantiation where we included the intermediate CA certificate in the chain. We do not have a clear understanding of why exactly this occurred. We were able to patch our Rustls implementation to use vectored I/O to write to the TCP socket.[8] This appears to have solved the problems and we saw the expected performance without having to turn off Nagle's algorithm or modifying `initcwnd`.

[8] The maintainers of Rustls independently also applied this optimization and it appeared in Rustls 0.18.0.