Verifying Post-Quantum Signatures in 8 kB of RAM

Ruben Gonzalez[1], Andreas Hülsing[2], Matthias J. Kannwischer[3], Juliane Krämer[4], Tanja Lange[2], Marc Stöttinger[5], Elisabeth Waitz[6], Thom Wiggers[7], and Bo-Yin Yang[8]

[1] Hochschule Bonn-Rhein-Sieg, Bonn, Germany
[2] Eindhoven University of Technology, Eindhoven, The Netherlands
[3] Max Planck Institute for Security and Privacy, Bochum, Germany
[4] Technische Universität Darmstadt, Darmstadt, Germany
[5] Hessen3C, Wiesbaden, Germany
[6] Elektrobit Automotive GmbH, Erlangen, Germany
[7] Radboud University, Nijmegen, The Netherlands
[8] Academica Sinica, Taipei, Taiwan
streaming-pq-sigs@kannwischer.eu

All post-quantum signatures have a larger combined size of public key and signature compared to RSA and DL-based signatures. This can pose challenges if signatures need to e verified in very constrained devices and is especially important when the payload of the signed message is much smaller than the signature, due to additional transmission overhead required for the signature. Such short messages are for example used in the real-world use case of feature activation in the automotive domain. Feature activation is the remote activation of features that are already implemented in the soft- and hardware of the car. For example, an additional infotainment package. Usually, a short activation code is protected with a signature to prevent unauthorized activation of the feature.

We focus on verification of signatures and cover NIST PQC round-3 candidates Dilithium, Falcon, Rainbow, GeMSS, and SPHINCS$^+$. We assume an ARM Cortex-M3 with 8 kB of memory and 8 kB of flash for code; a practical and widely deployed setup in, for example, the automotive sector. This amount of memory is insufficient for most schemes. Rainbow and GeMSS public keys are too big; SPHINCS$^+$ signatures do not fit in this memory. To make signature verification work for these schemes, we stream in public keys and signatures. Due to the memory requirements for efficient Dilithium implementations, we stream in the public key to cache more intermediate results.

We show that this way signature verification can be done keeping only small data packets in constrained memory. When streaming the public key, the device needs to securely store a hash value of the public key to verify the authenticity of the streamed public key. During signature verification, the public key is incrementally hashed, matching the data flow of the streamed public key. We implemented and benchmarked the proposed public key and signature streaming approach for four different signature schemes (Dilithium, SPHINCS$^+$, Rainbow, and GeMSS). Although for Dilithium streaming the public key is not strictly necessary, the saved bytes allow us to keep more intermediate results in memory. This results in a speed-up.

Table 1. Communication overhead in bytes and milliseconds at 500 kbit/s and 20 Mbit/s. GeMSS requires to stream in the public key *nb_ite* times (4 for `gemss-128`). All other schemes require streaming in the public key and signed message once.

	streaming data			streaming time	
	$\|pk\|$	$\|sig\|$	total	500 kbit/s	20 Mbit/s
`sphincs-s`[a]	32	7 856	7 888	126.2 ms	3.2 ms
`sphincs-f`[b]	32	17 088	17 120	273.9 ms	6.9 ms
`rainbowI-classic`	161 600	66	161 666	2 586.7 ms	64.7 ms
`gemss-128`	352 188	33	1 408 785[c]	22 540.6 ms	563.5 ms
`dilithium2`	1 312	2 420	3 732	59.7 ms	1.5 ms
`falcon-512`	897	690	1 587	25.4 ms	0.6 ms

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple` [c] $4 \cdot |pk| + |sig|$

For comparison, we also implemented the lattice-based scheme Falcon for which streaming small data packets is not necessary in our scenario as the entire public key and signature fit into RAM. The source code is published and available at https://git.fslab.de/pqc/streaming-pq-sigs. We demonstrate that the proposed streaming approach is very well suited for constrained devices with a maximum utilization of 8 kB RAM and 8 kB Flash.

The talk will highlight details of the implementations and show how the sizes and speeds are obtained. We summarize the results in the following tables.

Table 1 lists the public key, signature sizes, and the time needed for streaming them into the device at 500 kbit/s and 20 Mbit/s.

Table 2 presents the speed results for our implementations. The studied signature schemes rely on either SHA-256 (`rainbowI-classic`, `sphincs-sha256`) or SHA-3/SHAKE (`dilithium2`, `falcon-512`, and `gemss-128`). In a typical HSM-enabled device SHA-256 would be available in hardware and SHA-3/SHAKE will also be available in the future. However, on the Nucleo-F207ZG no hardware accelerators are available. Hence, we resort to software implementations instead. For SHA-256 we use the optimized C implementation from SUPERCOP.[1] For SHA-3/SHAKE, we rely on the ARMv7-M implementation from the XKCP.[2]

Table 3 presents the memory requirements of our implementations.

[1] https://bench.cr.yp.to/supercop.html
[2] https://github.com/XKCP/XKCP

Table 2. Cycle count for signature verification for a 33-byte message. Average over 1 000 signature verifications. Hashing cycles needed for verification of the streamed in public key (hashing and comparing to embedded hash) are reported separately. We also report the verification time on a practical HSM running at 100 MHz and also the total time including the streaming at 20 Mbit/s.

	w/o pk vrf.	w/ pk verification			w/ streaming
		pk vrf.	total	time[e]	20 Mbit/s
`sphincs-s`[a]	8 741k	0	8 741k	87.4 ms	90.6 ms
`sphincs-f`[b]	26 186k	0	26 186k	261.9 ms	268.7 ms
`rainbowI-classic`	333k	6 850k[d]	7 182k	71.8 ms	136.5 ms
`gemss-128`	1 619k	109 938k[c]	111 557k	1 115.6 ms	1 679.1 ms
`dilithium2`	1 990k	133k[c]	2 123k	21.2 ms	21.8 ms
`falcon-512`	581k	91k[c]	672k	6.7 ms	8.2 ms

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple` [c] SHA-3/SHAKE
[d] SHA-256 [e] At 100 MHz (no wait states)

Table 3. Memory and code-size requirements in bytes for our implementations. Memory includes stack needed for computations, global variables stored in the .bss section and the buffer required for streaming. Code-size excludes platform and framework code as well as code for `SHA-256` and `SHA-3`.

	memory				code
	total	buffer	.bss	stack	.text
`sphincs-s`[a]	6 904	4 928	780	1 196	2 724
`sphincs-f`[b]	7 536	4 864	780	1 892	2 586
`rainbowI-classic`	8 168	6 848	724	596	2 194
`gemss-128`	8 176	4 560	496	3 120	4 740
`dilithium2`	8 048	40	6 352	1 656	7 940
`falcon-512`	6 552	897	5 255	400	5 784

[a] `-sha256-128s-simple` [b] `-sha256-128f-simple`