From: Robin Larrieu <robin.larrieu@cryptonext-security.com> via pqc-forum <pqc-forum@list.nist.gov>
To: pqc-forum@list.nist.gov
Subject: [pqc-forum] Integration of Post-Quantum signatures
Date: Thursday, August 18, 2022 04:57:18 AM ET

Dear all,

I would like to point out some minor issues that appear when integrating certain signature schemes. To deploy these algorithms in constrained environments, typically in terms of memory or bandwidth, one has to rely on incremental APIs, such as PKCS-#11's C_SignInit, C_SignUpdate and C_SignFinal functions. A few assumptions must hold for interfaces like that to work. In particular, this implies that the message to be signed must be processed only once (this is an issue in Sphincs+).
Moreover, in PKCS-#11 incremental verification, the signature is available only during C_VerifyFinal, when the whole message has been processed. This implies that the initial hash should not prefix the message with a random nonce (as Falcon and Sphincs+ do); using a suffix should be fine though.
I will give more details on how these issues affect each cryptosystem at the end of this message.

Sure both these issues can be solved by hashing the message first, but this solution lacks transparency for users. Classical signatures do it this way because in general they can only sign fixed-length input; in fact the signature algorithm would be for example RSA-with-SHA256.
But considering that PQ signatures support arbitrary-length inputs, it may be unclear why one should use e.g. Falcon-with-SHA256 instead of plain Falcon, or why plain Sphincs+ is supported in the "One Shot" interface but not in the incremental version. Besides, using the hash-then-sign model somewhat increases the attack surface. For example, using hash-then-sign with Sphincs+ is vulnerable to the obvious collision attack, while Sphincs+ only needs second-preimage resistance according to section 11 of the specification.

Anyway, whether or not protocols and applications should use the

hash-then-sign model is a different topic, and there is currently a discussion about this on another thread. My point is simply that since direct signature of any message is possible with post-quantum schemes, this option should not be discarded merely because of an API issue. Specifically, the standardized signature schemes should be designed to support PKCS-#11 style interfaces, while giving the same results in both the "One Shot" and the incremental interfaces. I suppose NIST would take these constraints into account when writing the standards, but it seemed important to mention it anyway.
Since NIST plans another call for proposals to diversify the signature portfolio, it would also be helpful (though less critical at this point) if algorithm designers consider this in their submissions. Maybe NIST could even include this requirement explicitly in the call for proposals ?

As mentioned in the introduction, there are minor issues in the to-be-standardized algorithms regarding integration in PKCS-#11 style interfaces. Here are details about these issues; as we will see they can be solved by merely swapping fields in the initial hash. Feedback is welcome, especially if algorithm designers can confirm (or maybe object?) that such changes would not affect the security properties, or come up with more elegant solutions.

- Dilithium
Dilithium starts by computing mu = H(H(pk), msg) and uses only this digest afterwards (for the signature, H(pk) is precomputed as part of the secret key). There is no issue here, because in PKCS-#11 the signature/verification key is already available at C_SignInit/C_VerifyInit. Maybe swapping the fields, as in mu = H(msg,H(pk)) would have some merits in a server+HSM situation : the hash could be prepared on the faster server since it only needs the message, while the keys remain on the HSM. I believe there was a discussion about such hash separation on this mailing list several months ago.

- Falcon
Falcon computes the signature challenge c = HashToPoint(nonce, msg) where nonce is randomly chosen and written in the signature. As

mentioned earlier, in PKCS-#11 verification, the signature and in particular the nonce is only available during C_VerifyFinal, which means C_VerifyUpdate does not work.
Swapping the fields as in c = HashToPoint(msg, nonce) would solve this issue.

- Sphincs+
Sphincs+ signature computes R = PRF_msg(SK.prf, optRand, msg); digest = H_msg(R, PK.seed, PK.root, msg);.
This means that the message is processed twice in a row, which is incompatible with the incremental API.
Actually it does not really matter how R is computed for interoperability, so there are workarounds for the signature, but doing so would produce a different signature, making validation of a given implementation less convenient.
Besides, using R as a prefix in the computation of digest raises the same problems as Falcon regarding incremental verification.
Setting digest = H_msg(msg, R, PK.seed, PK.root) would solve both issues: for the signature, R and digest can be computed independently by maintaining 2 hash states.
Variants for R could also be considered. For example, with R=optRand, the message would be processed only once and the private key could be SPX_N bytes shorter by dropping SK.prf (but maybe this would make implementations vulnerable to weak randomness?). Or in the Shake instantiation, having R=shake(msg, SK.prf, optRand); digest = shake(msg, R, PK.seed, PK.root); would allow to process the message just once as well, by copying the state when done to finalize the computation of R and digest.

For readability, I suggest to open new threads to discuss cryptosystem-specific issues, and reserve this thread for general comments.

Best Regards,
Robin Larrieu
CryptoNext Security

--

From: D. J. Bernstein <djb@cr.yp.to> via pqc-forum@list.nist.gov
To: pqc-forum@list.nist.gov
Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
Date: Thursday, August 18, 2022 02:01:02 PM ET
Attachments: smime.p7m

'Robin Larrieu' via pqc-forum writes:
> Moreover, in PKCS-#11 incremental verification, the signature is
> available only during C_VerifyFinal, when the whole message has been
> processed.

With this type of interface, typical applications end up processing streams that haven't been verified yet, and the attacker can trivially feed forged data to the applications.

Since this type of interface is dangerous from a security perspective, creating hassles for this type of interface should be viewed as a positive feature of a spec rather than a negative feature.

It would be useful to instead specify secure signing for streams. For example, the verifier maintains a transcript hash h (starting as context or simply a constant), receives a signed packet (h,p,s) where h doesn't need to be transmitted explicitly, opens the signed packet to recover plaintext p, outputs p, updates h to H(h,p) where H is a hash function, and moves on to the next packet. It should be feasible with current techniques to computer-verify the security properties of such designs.

Instead of sending (p,s) in clear, I would send AONT(p,s) with any AONT in the literature, preferably keyed by h: for example, xor a hash of (h,s) into p, xor a hash of (h,p) into s, and so on for whatever number of rounds. 4 rounds are ample, and even 1 round (just xoring a hash of (h,s) into p), with a very weak hash function, would probably do the job of encouraging applications to call the proper cryptographic functions rather than extracting and releasing p. But the hash function H used to update h has to be strong so that the attacker can't rearrange packets.

An alternative to signing (h,p) is to sign H(h,p), so the receiver takes

(p,s), replaces h with a hash of (h,p), and opens the signed message (h,s). However, this seems to increase the damage done by collisions. Most applications shouldn't notice the cost of an extra hash layer.

If all data is available to the signer in advance then of course one signature on a hash tree suffices, and it's easy to arrange hashes so that packets can be verified in parallel. This covers the "we need to quickly verify firmware on boot" use case.

——D. J. Bernstein

From: Robin Larrieu <robin.larrieu@cryptonext-security.com> via pqc-forum <pqc-forum@list.nist.gov>
To: pqc-forum@list.nist.gov
Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
Date: Friday, August 19, 2022 03:29:06 AM ET

Le 18/08/2022 à 20:00, D. J. Bernstein a écrit :
> 'Robin Larrieu' via pqc-forum writes:
>> Moreover, in PKCS-#11 incremental verification, the signature is
>> available only during C_VerifyFinal, when the whole message has been
>> processed.
> With this type of interface, typical applications end up processing
> streams that haven't been verified yet, and the attacker can trivially
> feed forged data to the applications.

In this scenario, sure this is a problem. The use case I had in mind, and for which this PKCS-#11 subset was designed, is rather:
- You have a large signed document that you want to verify
- You want to delegate the verification to an external crypto module, be it for performance reasons (hardware acceleration), or because you did not want to reimplement the crypto yourself
- The crypto module, or the secure channel to it, cannot handle the whole document at once
In fact, when I said "when the whole message has been processed", it was from the crypto module point of view, that is "once the crypto module received all chunks of the message, hashing each one on the fly because buffering the whole data is not an option". This assumes that the actual application would still be able to store the entire document, and decide what to do with it once the signature has been verified.

Signing streams is a different problem, and obviously signing the whole data at the end of the stream is insecure for the reasons you mention. There are most likely published protocols to do just that, but I did not look into it since I never had to deal with this kind of scenarios so far.

> Since this type of interface is dangerous from a security perspective,
> creating hassles for this type of interface should be viewed as a
> positive feature of a spec rather than a negative feature.

In this case I have to disagree. Creating hassles for potentially useful functionalities of standard interfaces, because some users might use said functionalities for the wrong jobs, feels like a rather negative feature of the spec.

Best Regards,
Robin Larrieu
CryptoNext Security

--

From: Sydney Antonov <ska84@protonmail.com> via pqc-forum <pqc-forum@list.nist.gov>
To: Robin Larrieu <robin.larrieu@cryptonext-security.com>
CC: pqc-forum@list.nist.gov
Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
Date: Friday, August 19, 2022 09:03:17 AM ET

> In this scenario, sure this is a problem. The use case I had in mind,
> and for which this PKCS-#11 subset was designed, is rather:
> - You have a large signed document that you want to verify
> - You want to delegate the verification to an external crypto module, be
> it for performance reasons (hardware acceleration), or because you did
> not want to reimplement the crypto yourself
> - The crypto module, or the secure channel to it, cannot handle the
> whole document at once
> In fact, when I said "when the whole message has been processed", it was
> from the crypto module point of view, that is "once the crypto module
> received all chunks of the message, hashing each one on the fly because
> buffering the whole data is not an option". This assumes that the actual
> application would still be able to store the entire document, and decide
> what to do with it once the signature has been verified.

Wouldn't it be a better design to sign (h, iv) where h is the hash of the first packet or iv if the message is empty and each packet has the hash of the next packet as a prefix apart from the last packet which would have iv as a prefix? This way the application need not store the entire document.

How strong assumptions about the hash function does this rely on if iv is random?

Sydney

From: Robin Larrieu <robin.larrieu@cryptonext-security.com> via pqc-forum <pqc-forum@list.nist.gov>
To: pqc-forum@list.nist.gov
Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
Date: Friday, August 19, 2022 12:43:47 PM ET

Let me reformulate the issue, before this thread drifts off-topic about
"how to authenticate large amounts of data given as a stream".

I started this thread because
1) Vendors of cryptographic products must implement the algorithms
according to various standard interfaces, such as the C_SignInit,
C_SignUpdate, C_SignFinal functions from PKCS-#11 and their counterpart
for verification.
2) Since these interfaces may be pertinent in certain use cases, there
are most likely business application out there that rely on such interfaces
3) The selected signature algorithms, as currently specified, are
incompatible with such interfaces (unless using a hash-then-sign model),
for reasons that were detailed in my initial comment
4) The selected signature algorithms can be made compatible with such
interfaces using minor specification tweaks, as shown again in my
initial comment.

To ensure the widest adoption of post-quantum cryptography, I think NIST
should take these constraints into account when writing the standard,
either by making sure that each algorithm specification is compatible
with usual interfaces, or by standardizing the hash-then-sign method
only (but the latter would seem less satisfying).

Best Regards,
Robin Larrieu
CryptoNext Security

Le 19/08/2022 à 15:02, Sydney Antonov a écrit :
>> In this scenario, sure this is a problem. The use case I had in mind,
>> and for which this PKCS-#11 subset was designed, is rather:
>> - You have a large signed document that you want to verify
>> - You want to delegate the verification to an external crypto module, be

>> it for performance reasons (hardware acceleration), or because you did
>> not want to reimplement the crypto yourself
>> - The crypto module, or the secure channel to it, cannot handle the
>> whole document at once
>> In fact, when I said "when the whole message has been processed", it was
>> from the crypto module point of view, that is "once the crypto module
>> received all chunks of the message, hashing each one on the fly because
>> buffering the whole data is not an option". This assumes that the actual
>> application would still be able to store the entire document, and decide
>> what to do with it once the signature has been verified.
> Wouldn't it be a better design to sign (h, iv) where h is the hash of the
> first packet or iv if the message is empty and each packet has the hash of
> the next packet as a prefix apart from the last packet which would have iv
> as a prefix? This way the application need not store the entire document.
>
> How strong assumptions about the hash function does this rely on if iv
> is random?
>
> Sydney

From: Sydney Antonov <ska84@protonmail.com> via pqc-forum <pqc-forum@list.nist.gov>
To: Robin Larrieu <robin.larrieu@cryptonext-security.com>
CC: pqc-forum@list.nist.gov
Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
Date: Friday, August 19, 2022 06:48:16 PM ET

> 1) Vendors of cryptographic products must implement the algorithms
> according to various standard interfaces, such as the C_SignInit,
> C_SignUpdate, C_SignFinal functions from PKCS-#11 and their counterpart
> for verification.
> 2) Since these interfaces may be pertinent in certain use cases, there
> are most likely business application out there that rely on such interfaces
> 3) The selected signature algorithms, as currently specified, are
> incompatible with such interfaces (unless using a hash-then-sign model),
> for reasons that were detailed in my initial comment
> 4) The selected signature algorithms can be made compatible with such
> interfaces using minor specification tweaks, as shown again in my
> initial comment.

These "minor specification tweaks" essentially and necessarily convert
the signatures to hash-then-sign, which necessarily makes them vulnerable
to collision attacks.

> To ensure the widest adoption of post-quantum cryptography, I think NIST
> should take these constraints into account when writing the standard,
> either by making sure that each algorithm specification is compatible
> with usual interfaces, or by standardizing the hash-then-sign method
> only (but the latter would seem less satisfying).

Instead of weakening signatures for everyone, those requiring a
PKCS-#11 incremental interface can sign e.g. H(pk, m) where H is a
collision-resistant hash function, pk is the public key and m is the
message.

Sydney

--

From: Robin Larrieu <robin.larrieu@cryptonext-security.com> via pqc-forum <pqc-forum@list.nist.gov>
To: pqc-forum@list.nist.gov
Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
Date: Saturday, August 20, 2022 06:26:38 AM ET

Le 20/08/2022 à 00:47, Sydney Antonov a écrit :
>> 1) Vendors of cryptographic products must implement the algorithms
>> according to various standard interfaces, such as the C_SignInit,
>> C_SignUpdate, C_SignFinal functions from PKCS-#11 and their counterpart
>> for verification.
>> 2) Since these interfaces may be pertinent in certain use cases, there
>> are most likely business application out there that rely on such interfaces
>> 3) The selected signature algorithms, as currently specified, are
>> incompatible with such interfaces (unless using a hash-then-sign model),
>> for reasons that were detailed in my initial comment
>> 4) The selected signature algorithms can be made compatible with such
>> interfaces using minor specification tweaks, as shown again in my
>> initial comment.
> These "minor specification tweaks" essentially and necessarily convert
> the signatures to hash-then-sign, which necessarily makes them vulnerable
> to collision attacks.
Actually, the signatures are already hash-then-sign in some sense,
except they hash the message with some additional data for security. The
specification tweak is merely to swap fields in this initial hash, so
that the additional data is consumed at a point where it is available in
a typical PKCS-#11 scenario.
For example, Falcon and Sphincs+ essentially pick a random nonce, then
sign H(nonce, m) and append the nonce to the signature. This does not
work in PKCS-#11 incremental verification, because the signature
(including the nonce) is only available after the message. Signing H(m,
nonce) solves this issue, and should be equivalent in terms of security.

It is a bit more tricky with interfaces that separate the digest and the
signature as an explicit hash-then-sign, but it can work too: the trick
is to use as "digest" a serialization of the hash function internal
state, so the additional data can be appended while signing this "digest".
One would need to standardize the serialization if interoperability is

expected between digest and signature (or digest and verification), but in general the signer will use an implementation of digest and signature from the same vendor, and similarly for the verifier, so this extra standardization should not even be needed.

> Instead of weakening signatures for everyone, those requiring a
> PKCS-#11 incremental interface can sign e.g. H(pk, m) where H is a
> collision-resistant hash function, pk is the public key and m is the
> message.

It is indeed a solution. I think however that it would be preferable for transparency and interoperability if the signature method is the same for the one-shot and the incremental interfaces.
Of course, the modification should not be done if it "weakens the signatures for everyone", but it seems that the algorithms can be made compatible with an incremental interface without changing their security properties.

Best regards,
Robin Larrieu
CryptoNext Security

From: Sydney Antonov <ska84@protonmail.com> via pqc-forum <pqc-forum@list.nist.gov>
To: Robin Larrieu <robin.larrieu@cryptonext-security.com>
CC: pqc-forum@list.nist.gov
Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
Date: Saturday, August 20, 2022 10:09:20 AM ET

> Actually, the signatures are already hash-then-sign in some sense,
> except they hash the message with some additional data for security. The
> specification tweak is merely to swap fields in this initial hash, so
> that the additional data is consumed at a point where it is available in
> a typical PKCS-#11 scenario.
> For example, Falcon and Sphincs+ essentially pick a random nonce, then
> sign H(nonce, m) and append the nonce to the signature. This does not
> work in PKCS-#11 incremental verification, because the signature
> (including the nonce) is only available after the message. Signing H(m,
> nonce) solves this issue, and should be equivalent in terms of security.

It's not equivalent in terms of security: swapping the fields would make
Falcon and Sphincs+ vulnerable to collisions in the internal state of
the hash function.

Sydney

From: Scott Fluhrer (sfluhrer) <sfluhrer@cisco.com> via pqc-forum <pqc-forum@list.nist.gov>
To: Robin Larrieu <robin.larrieu@cryptonext-security.com>, pqc-forum@list.nist.gov
Subject: RE: [pqc-forum] Integration of Post-Quantum signatures
Date: Saturday, August 20, 2022 10:11:39 AM ET

> ———Original Message———
> From: 'Robin Larrieu' via pqc-forum <pqc-forum@list.nist.gov>
> Sent: Saturday, August 20, 2022 6:26 AM
> To: pqc-forum@list.nist.gov
> Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
>
> For example, Falcon and Sphincs+ essentially pick a random nonce, then sign
> H(nonce, m) and append the nonce to the signature. This does not work in
> PKCS-#11 incremental verification, because the signature (including the
> nonce) is only available after the message. Signing H(m,
> nonce) solves this issue, and should be equivalent in terms of security.

No, they are not equivalent. With H(m, nonce), then (with the on-line hash functions we use in practice) if someone finds two message prefixes m, m' whose immediate hash state are the same, then for an arbitrary message postfix n, we have found a signature collision between (m,n) and (m',n), independent of what the nonce is selected as; for example, this is known to be insecure if H is SHA-1. In contrast, H(nonce, m) has no similar known weakness to results in a collision.

I cannot speak for the Falcon team, however the Sphincs+ team selected H(nonce, m) specifically to avoid depending on the hash function collision resistance (because Sphincs+ doesn't depend on it anywhere else in the scheme).

.

From: Robin Larrieu <robin.larrieu@cryptonext-security.com> via pqc-forum <pqc-forum@list.nist.gov>
To: pqc-forum@list.nist.gov
Subject: Re: [pqc-forum] Integration of Post-Quantum signatures
Date: Saturday, August 20, 2022 11:23:06 AM ET

Le 20/08/2022 à 16:11, Scott Fluhrer (sfluhrer) a écrit :
> No, they are not equivalent. With H(m, nonce), then (with the on-line
> hash functions we use in practice) if someone finds two message
> prefixes m, m' whose immediate hash state are the same, then for an
> arbitrary message postfix n, we have found a signature collision
> between (m,n) and (m',n), independent of what the nonce is selected
> as; for example, this is known to be insecure if H is SHA-1. In
> contrast, H(nonce, m) has no similar known weakness to results in a
> collision.

Fair enough. My reasoning was that a collision on the internal state was a much stronger attack than a collision on the hash function (as the former trivially implies the latter), and I incorrectly identified this as a complete break of the hash function. In particular I did not realize that the known collisions on SHA-1 worked one way but not the other. Please consider my whole point moot and excuse all this noise.

Best regards,
Robin Larrieu
CryptoNext Security