Paralithium: A SNARK-Friendly Dilithium Variant

Xiong Fan and Chris Peikert

Algorand, Inc.

October 21, 2021

1 Modified Signature Scheme

The key-generation, signing, and verification algorithms for the modified Dilithium signature scheme are specified in Figure 1. All the notation is as in the original Dilithium specification document [BDK$^+$21], with the following additions:

- The SampleInBall algorithm is always written as the composition Shuffle $\circ$ H, where Shuffle is the "inside-out" Fisher–Yates shuffle algorithm described in [BDK$^+$21, Section 2.3 and Figure 2], and H (given an input seed) supplies the pseudorandom bits used by the shuffle. This decomposition of SampleInBall is important to our modifications, so we always make it explicit.

- SS is the concrete instantiation of the *unsalted* subset-sum hash function (for arbitrary-length inputs), as defined in [GLP21, Section 4.2].

- SSS is the concrete instantiation of the *salted* subset-sum hash function, as specified in [GLP21, Section 4.2] using the mode of operation described in [HK06].

1.1 Rationale

The following summarizes the reasons for the changes. *The primary goal is to obtain a more "SNARK-friendly" verification procedure*, i.e., it should be more efficient for a SNARK prover to prove that the verification algorithm accepts on a given public key, message, and signature. This entails minimizing the number of "SNARK-unfriendly" components and computations, and (where necessary) replacing them with ones that are friendlier but are still believed to be sufficiently secure, e.g., subset-sum hashing.

A second important goal is to *reduce the risk associated with a potential future loss in the concrete security of subset-sum hashing.* This is done by relying only on weaker conjectured security properties wherever possible, and designing to limit the scope of the negative implications of any future cryptanalytic improvements.

```
Gen                                                                                                         ▷ original
Gen($\rho \in \{0,1\}^{256}$)                                                                               ▷ updated
01 $\zeta \leftarrow \{0,1\}^{256}$
02 $(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} := \mathrm{H}(\zeta)$      ▷ H is SHAKE-256      ▷ original
02 $(\rho', K) \in \{0,1\}^{512} \times \{0,1\}^{256} := \mathrm{H}(\zeta)$                                  ▷ updated
03 $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
04 $(\mathbf{s}_1, \mathbf{s}_2) \in S_\eta^\ell \times S_\eta^k := \mathsf{ExpandS}(\rho')$
05 $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
06 $(\mathbf{t}_1, \mathbf{t}_0) := \mathsf{Power2Round}_q(\mathbf{t}, d)$
07 $tr \in \{0,1\}^{256} := \mathrm{H}(\rho \parallel \mathbf{t}_1)$                                          ▷ original
07 $tr \in \{0,1\}^{512} := \mathrm{SS}(\rho \parallel \mathbf{t}_1)$                                          ▷ updated
08 return $(pk = (\rho, \mathbf{t}_1), sk = (\rho, K, tr, \mathbf{s}_1, \mathbf{s}_2, \mathbf{t}_0))$          ▷ NB: $tr$ is larger

Sign($sk, M \in \{0,1\}^*$)
09 $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
10 $\mu \in \{0,1\}^{512} := \mathrm{H}(tr \parallel M)$                                                      ▷ original
10 $\mu \in \{0,1\}^{512} := \mathrm{SSS}(tr \parallel M; r), r \in \{0,1\}^{512} := \mathrm{H}(K \parallel \texttt{0x00} \parallel M)$ (or $r \leftarrow \{0,1\}^{512}$)      ▷ updated
11 $\kappa := 0$, $(\mathbf{z}, \mathbf{h}) := \perp$
12 $\rho' \in \{0,1\}^{512} := \mathrm{H}(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing)      ▷ original
12 $\rho' \in \{0,1\}^{512} := \mathrm{H}(K \parallel \texttt{0xFF} \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing)      ▷ updated
13 while $(\mathbf{z}, \mathbf{h}) = \perp$ do
14    $\mathbf{y} \in \tilde{S}_{\gamma_1}^\ell := \mathsf{ExpandMask}(\rho', \kappa)$
15    $\mathbf{w} := \mathbf{A}\mathbf{y}$
16    $\mathbf{w}_1 := \mathsf{HighBits}_q(\mathbf{w}, 2\gamma_2)$
17    $\tilde{c} \in \{0,1\}^{256} := \mathrm{H}(\mu \parallel \mathbf{w}_1)$                                   ▷ original
17    $\alpha \in \{0,1\}^{512} := \mathrm{SS}(\mu \parallel \mathbf{w}_1)$                                     ▷ updated
18    $c \in B_\tau := \mathsf{Shuffle}(\mathrm{H}(\tilde{c}))$ $[= \mathsf{SampleInBall}(\tilde{c})]$           ▷ original
18    $c \in B_\tau := \mathsf{Shuffle}(\mathrm{H}(\mu \parallel \alpha))$                                       ▷ updated
19    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
20    $\mathbf{r}_0 := \mathsf{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)$
21    if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ or $\|\mathbf{r}_0\|_\infty \geq \gamma_2 - \beta$, then $(\mathbf{z}, \mathbf{h}) := \perp$
22    else
23       $\mathbf{h} := \mathsf{MakeHint}_q(-c\mathbf{t}_0, \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0, 2\gamma_2)$
24       if $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$ or the # of 1's in $\mathbf{h}$ is greater than $\omega$, then $(\mathbf{z}, \mathbf{h}) := \perp$
25    $\kappa := \kappa + \ell$
26 return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$                                                        ▷ original
26 return $\sigma = (\alpha, \mathbf{z}, \mathbf{h}, r)$                                                        ▷ updated

Verify($pk = (\rho, \mathbf{t}_1), M, \sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$)                          ▷ original
Verify($pk = (\rho, \mathbf{t}_1), M, \sigma = (\alpha, \mathbf{z}, \mathbf{h}, r)$)                          ▷ updated
27 $\mathbf{A} \in R_q^{k \times \ell} := \mathsf{ExpandA}(\rho)$
28 $\mu \in \{0,1\}^{512} := \mathrm{H}(tr \parallel M)$ where $tr \in \{0,1\}^{256} := \mathrm{H}(\rho \parallel \mathbf{t}_1)$      ▷ original
28 $\mu \in \{0,1\}^{512} := \mathrm{SSS}(tr \parallel M; r)$ where $tr \in \{0,1\}^{512} = \mathrm{SS}(\rho \parallel \mathbf{t}_1)$      ▷ updated
29 $c \in B_\tau := \mathsf{Shuffle}(\mathrm{H}(\tilde{c}))$                                                    ▷ original
29 $c \in B_\tau := \mathsf{Shuffle}(\mathrm{H}(\mu \parallel \alpha))$                                        ▷ updated
30 $\mathbf{w}_1 := \mathsf{UseHint}_q(\mathbf{h}, \mathbf{A}\mathbf{z} - c \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2)$
31 return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ and $[\![\#$ of 1s in $\mathbf{h}$ is $\leq \omega]\!]$ and $[\![\tilde{c} = \mathrm{H}(\mu \parallel \mathbf{w}_1)]\!]$      ▷ original
31 return $[\![\|\mathbf{z}\|_\infty < \gamma_1 - \beta]\!]$ and $[\![\#$ of 1s in $\mathbf{h}$ is $\leq \omega]\!]$ and $[\![\alpha = \mathrm{SS}(\mu \parallel \mathbf{w}_1)]\!]$      ▷ updated
```

Figure 1: Pseudocode “diff” for the modified version of Dilithium. Original lines are shown in red, and their replacements are shown in green.

1.1.1 Minimize SHAKE/Keccak Calls by the Verifier

The main route to more SNARK-friendly verification is to minimize the number of calls to the (SNARK-unfriendly) SHAKE function and its underlying Keccak permutation that need to be performed by the verification algorithm. Where appropriate, SNARK-friendly subset-sum hashing is used instead. Concretely, this is accomplished via the following changes:

Provide the public "A-expansion seed" ρ to key generation, allowing many keys to use the same $\mathbf{A}$ matrix. The large pseudorandom public matrix $\mathbf{A} \in R_q^{k \times \ell}$ is defined as $\mathbf{A} := \mathsf{ExpandA}(\rho)$ (lines 03 and 27). Unfortunately, generating all these output bits requires many calls to the Keccak permutation, which is highly prohibitive for the SNARK prover. By fixing a value of ρ across many keys, the corresponding value of $\mathbf{A}$ can be precomputed and hard-coded into the verification algorithm and corresponding SNARK statement.

We stress that all the security reductions for Dilithium go through just as well in this "shared $\mathbf{A}$" setting, as long as $\mathbf{A}$ is properly distributed using trustworthy randomness (which heuristically follows from modeling H as a random oracle). Note that Dilithium uses a per-key $\mathbf{A}$ matrix to avoid the theoretical risk of an "all-for-the-price-of-one attack," in which the attacker expends enormous computation on a shared $\mathbf{A}$ in order to obtain some "trapdoor" information, which may let it more easily extract secret keys from any public keys that use this $\mathbf{A}$. However, this kind of attack is no easier than extracting a secret key from a single public key, and the Dilithium parameters have been chosen so as to make this extremely infeasible according to current and reasonably extrapolated cryptanalysis.

Compress the larger inputs $M, \mathbf{w}_1$ using subset-sum hashing (lines 28 and 31, updated). The message M is of unbounded size, and the "commitment" value $\mathbf{w}_1$ is more than 6000 bits for typical Dilithium parameters, so applying SHAKE to them (in the original lines 28 and 31) requires several Keccak calls, which is prohibitive for the SNARK prover. Instead, both quantities are instead hashed using subset-sum hashing, which is SNARK-friendly.

For the message digest μ (line 28), the security reductions and analysis for Dilithium show that we only need *distinct* digests for distinct adversarially chosen messages M (essentially, Dilithium uses Fiat–Shamir to sign the digest). This distinctness is assured by the target-collision resistance of the SSS hash function; see Section 1.1.2 below.

For the commitment value $\mathbf{w}_1$, the issues are more subtle. First, the proof that signatures are "zero knowledge" (i.e., they do not leak information about the secret key) just requires the *programmability* of the random oracle for its inputs derived from $\mu, \mathbf{w}_1$. This programmability is clearly preserved when $\mathrm{H}(\mu \parallel \mathbf{w}_1)$ (line 31, original) is replaced with $\mathrm{H}'(\mu, \mathbf{w}_1) := \mathrm{H}(\mu \parallel \alpha)$ (line 31, updated), as long as the values $\alpha = \mathrm{SS}(\mu \parallel \mathbf{w}_1)$ are distinct for the distinct values of $\mathbf{w}_1$ generated across all the simulated signatures produced *just for message digest* μ. Clearly, this is implied by the collision resistance of SS, but the actual needed property is much weaker, and is merely statistical: because each signing query is overwhelmingly likely to yield a distinct message digest μ, there will be just one simulated $\mathbf{w}_1$ per μ, so the distinctness of the H inputs follows immediately.

Second, given that signatures are zero knowledge, unforgeability is tightly equivalent to the hardness of a "self-target" problem involving the interaction between lattices and the hash function. With our modifications, the hash function H is essentially replaced by H' in the self-target problem. This modified self-target problem is analyzed in Section 1.4.

We point out that, unlike for the message digest, there is no apparent benefit to using random salt in the compression of $\mathbf{w}_1$. For zero knowledge and programmability, the randomness of $(\mu, \mathbf{w}_1)$ alone is enough to avoid collisions. For unforgeability (on an as-yet-unsigned digest μ) and the self-target problem, the adversary works offline and would have its choice of salt anyway, so there is no benefit in the signer using random salt.

Compute the challenge c using just one SHAKE call. The original Dilithium specification computes the "challenge" element $c \in B_\tau$ from the message digest μ and the "blinding" value $\mathbf{w}_1$ using *two* sequential calls to SHAKE. First, it computes a binary $\tilde{c} \in \{0,1\}^{256}$ as $\tilde{c} := \mathrm{H}(\mu \parallel \mathbf{w}_1)$ (line 17, original). Then, it uses $\tilde{c}$ as the input to $\mathsf{SampleInBall} = \mathsf{Shuffle} \circ \mathrm{H}$ to compute $c := \mathsf{Shuffle}(\mathrm{H}(\tilde{c}))$ (line 18, original); see [BDK$^+$21, Section 2.3]. Thus, the SHAKE output $\tilde{c}$ is immediately fed back to SHAKE to generate pseudorandom bytes for the shuffle algorithm. All of these computations must also be performed in the verification algorithm (lines 29 and 31, original), thus requiring two SHAKE calls, and thereby several Keccak invocations (because handling $\mathbf{w}_1$, which is rather long, requires multiple Keccak calls).

The modified scheme computes the challenge c using just *one* SHAKE call (lines 18 and 29, updated). This is achieved via the subset-sum compression of $\mathbf{w}_1$ (described above), by eliminating the second of the two sequential SHAKE calls (on $\tilde{c}$), and by using the first SHAKE call to produce enough pseudorandom bits for $\mathsf{Shuffle}$ to produce the challenge, as $c := \mathsf{Shuffle}(\mathrm{H}(\mu \parallel \alpha))$ (line 18, updated). Importantly, when modeling SHAKE as a random oracle, this single call is sufficient for the Fiat–Shamir heuristic and its security analysis (see the above remarks on programmability).

The above changes mean that there is no need for $\tilde{c}$, so it has been eliminated and replaced in the signature by the compressed "blinding" value $\alpha = \mathrm{SS}(\mu \parallel \mathbf{w}_1) \in \{0,1\}^{512}$ (line 26, original versus updated). This is an increase of just 256 bits (32 bytes) in the signature size, which is insignificant compared to its original size.[1]

With these modifications, verification now uses just one call to SHAKE, which uses just one invocation of the underlying Keccak permutation: we map the $512 + 512 = 1024 < 1088$ input bits of $\mu \parallel \alpha$ to 1088 output bits (136 bytes), which essentially always suffice for $\mathsf{Shuffle}$ to successfully complete. In the extremely rare event that this is not the case, the signing algorithm merely rejects and starts over. The verification algorithm also requires that $\mathsf{Shuffle}$ succeed when given the 1088-bit output from a single Keccak call.

1.1.2 Rely Only on TCR for the Message Digest

As mentioned above, a primary required property is that the message digests μ produced by the signing algorithm should be distinct for distinct adversarially chosen messages M. This property is implied by the collision resistance (CR) of SSS (line 10, updated), which in turn is immediately implied by the CR of SS. This is a well studied assumption in which we can have good confidence, following extensive cryptanalysis (both classical and quantum). Yet it would be preferable to rely

[1] Alternatively, we could have replaced $\tilde{c}$ with (a suitable canonical encoding of) $c := \mathsf{Shuffle}(\tilde{c}) \in B_\tau$, or defined it as $\tilde{c} := \mathrm{H}(\mu \parallel \alpha)$ and kept it in the signature (in each case, with a suitable equality check in Verify). All these options are equally secure in terms of unforgeability, since we can efficiently transform a valid signature from any one of these forms to any other form. However, the former option requires defining and implementing a canonical encoding of elements in B_τ, which, while conceptually straightforward, requires additional encoding and decoding functions that are somewhat cumbersome and error prone. For the latter option, $\tilde{c}$ would need to be long enough to supply sufficient pseudorandom bits to $\mathsf{Shuffle}$, and thus would be longer than α or even a naïve encoding of c.

on a weaker assumption, in order to reduce the risk associated with a potential future significant cryptanalytic advance against collision resistance.

Accordingly, the modified design relies only on the *target* collision resistance (TCR), not full collision resistance, of SSS with (pseudo)random salt r for the message digests μ generated by the signing algorithm. It is shown in [HK06] that TCR is implied by either one of two *second preimage resistance*-like (SPR) properties of the underlying subset-sum compression function, called *chosen*-SPR or *evaluated*-SPR. The compression function appears to have somewhat stronger (and certainly does not have weaker) concrete security for these properties than it does for collision resistance. (And again, even in the case of potentially weakly random salt, distinct message digests are still assured, assuming that SSS is CR.)

The cost of relying solely on TCR for the message digest is the inclusion of the salt value $r \in \{0,1\}^{512}$ in the signature (line 26, updated). (As shown in [HK06], the salt itself does not need to be authenticated.) These additional 512 bits (64 bytes) are insignificant compared to the original signature size.

We point out that, given the need to compress large messages M (together with tr) to short digests μ in a SNARK-friendly way, there is no apparent benefit to using SHAKE/Keccak in the derivation of the digest: if collisions can be found in the SNARK-friendly compression stage, then subsequently applying another function to the result does not obviate these collisions. Conversely, if collisions cannot be found in the SNARK-friendly stage, then any additional SNARK-unfriendly steps are just wasted effort.

1.2 (Strong) Binding

In addition to (strong) unforgeability under chosen-message attack ((S)UF-CMA), signature schemes sometimes need other security properties for certain applications. Important ones go under the brief names *message binding* and *key binding*[2], and their conjunction *strong binding* [CDF+21, BCJZ21, CGN20].[3]

- Informally, message binding says that a signature uniquely determines the message it signs. More precisely, it must be infeasible for an adversary to generate (pk, M, M', σ) such that $M \neq M'$, and both $\mathsf{Verify}(pk, M, \sigma)$ and $\mathsf{Verify}(pk, M', \sigma)$ accept.
- Similarly, key binding says that a signature uniquely determines the public key under which it is valid.
- Strong binding is the conjunction of the above two properties. In other words, it must be infeasible for an adversary to generate (pk, pk', M, M', σ) such that $(pk, M) \neq (pk', M')$, and both $\mathsf{Verify}(pk, M, \sigma)$ and $\mathsf{Verify}(pk', M', \sigma)$ accept.

Importantly, in the above attacks, public keys and signatures may be maliciously crafted; they need not be valid outputs of the scheme's specified algorithms. Clearly, strong binding implies (message) binding.

It was shown in [CDF+21] that the original Dilithium scheme has strong binding as long as H is collision resistant (we include the proof below for reference). Our modified design preserves the strong binding property, as long as SS and SSS (with adversarially chosen salt r) are collision resistant; recall that this holds as long as their common compression function is collision resistant.

[2]This is a shorter alternative to "malicious strong universal exclusive ownership."

[3]The work of [CDF+21] also defines another property called *non-resignability*, which we do not consider here.

Importance of recomputing tr in Verify. For strong binding, it is *essential* that the Verify algorithm (both original and modified) recompute tr from the public key, or obtain it from a trustworthy source. An alternative design, in which tr is not used for deriving the message digest, or is included with the public key and Verify merely assumes it is correct, would allow for a simple attack against strong binding (but not against ordinary message binding). The attack works against this alternative design for both the original and modified schemes; for simplicity, we demonstrate it for the original. Let $pk = (\rho, \mathbf{t}_1, tr)$, $pk' = (\rho, \mathbf{t}'_1, tr)$ for arbitrary ρ, tr and special $\mathbf{t}_1 = \mathbf{0}$ and $\mathbf{t}'_1 = \mathbf{e}_1$, the vector of ring elements (polynomials) that is all zeroes except for the first entry, which is unity. Let $\mathbf{z}$ and $\mathbf{h}$ be arbitrary, with $\mathbf{z}$ sufficiently short.

Observe that for *any* $c \in B_\tau$, $\mathbf{Az} - c \cdot \mathbf{t}_1 \cdot 2^d$ and $\mathbf{Az} - c \cdot \mathbf{t}'_1 \cdot 2^d$ differ by $c \cdot \mathbf{e}_1 \cdot 2^d$. Therefore, $\mathsf{UseHint}_q$ applied to either of these (with the same hint $\mathbf{h}$) has a good chance of yielding the same $\mathbf{w}_1$, because $\mathsf{UseHint}_q$ essentially "rounds away" several more than d low-order bits in each entry.[4] Finally, let $M = M'$ be an arbitrary message, and let the common signature be $\sigma = (\tilde{c} = \mathrm{H}(\mu \parallel \mathbf{w}_1), \mathbf{z}, \mathbf{h})$. It is apparent that this signature is accepted for both key-message pairs.

An obvious countermeasure to the above attack is for Verify to reject if $\mathbf{t}_1$ is "too close" to zero. However, this countermeasure appears to be circumventable by a slight generalization of the above attack, which makes $\mathbf{t}_1, \mathbf{t}'_1$ far from zero, but still differ by unity in a single entry. The main idea is to define $\mathbf{t}_1, \mathbf{t}'_1$ so that both $\mathbf{t}_1 \cdot 2^d, \mathbf{t}'_1 \cdot 2^d$ are close to the same $\mathbf{As}_1$, for some short $\mathbf{s}_1$ that serves as their common secret key. For example, define $\mathbf{t}_1$ by rounding $\mathbf{As}_1 + \mathbf{s}_2$ as usual for suitable small $\mathbf{s}_1, \mathbf{s}_2$, then let $\mathbf{t}'_1$ differ from $\mathbf{t}_1$ by unity in a single entry.

For each of these distinct keys, and for the same tr (which is incorrect in at least one case) and the same message M yielding the same message digest μ, run the legitimate signing procedure with the same short blinding term $\mathbf{y}$, which yields the same $\mathbf{w}_1, \tilde{c}, \mathbf{z}$, and, with some probability, the same hint $\mathbf{h}$; if this occurs, the attack succeeds. As above, the same hint arises if $\mathbf{Az} - c \cdot \mathbf{t}_1 \cdot 2^d$ and $\mathbf{Az} - c \cdot \mathbf{t}'_1 \cdot 2^d$ lie in the same $\mathsf{UseHint}_q$ "rounding box," because each hint corresponds to the difference between the corresponding quantity's box and $\mathbf{w}_1$'s box.

Proof of strong binding. The following argument, originally due to [CDF+21], shows that the original Dilithium scheme has strong binding, via a tight reduction based on the collision resistance of H. The proof of the modified design's strong binding, assuming the collision resistance of SS and SSS (for adversarially chosen salt r), is a straightforward adaptation based on the updated lines 28 and 31.

Suppose for the purposes of contradiction that the adversary is able to violate strong binding, i.e., it produces some

$$(pk = (\rho, \mathbf{t}_1), pk' = (\rho', \mathbf{t}'_1), M, M', \sigma = (\tilde{c}, \mathbf{z}, \mathbf{h}))$$

such that $(pk, M) \neq (pk', M')$, and both $\mathsf{Verify}(pk, M, \sigma)$ and $\mathsf{Verify}(pk', M', \sigma)$ accept. Then there are two main cases, each of which yields a collision in H:

1. First, suppose that the key-message pairs yield *different* message digests $\mu, \mu' \in \{0,1\}^{512}$ on line 28 (original). Then by the verification condition on line 31 (original), we have $\tilde{c} = \mathrm{H}(\mu \parallel \mathbf{w}_1) = \mathrm{H}(\mu' \parallel \mathbf{w}'_1)$ for some efficiently computable $\mathbf{w}_1, \mathbf{w}'_1$. This immediately yields a collision $\mu \parallel \mathbf{w}_1 \neq \mu' \parallel \mathbf{w}'_1$ in H.

[4]Indeed, the same $\mathbf{w}_1 = \mathbf{0}$ is *guaranteed* if we take $\mathbf{z} = \mathbf{0}$, and we can increase the chance of success by searching for an $\mathbf{Az}$ that is close to a center of one of $\mathsf{UseHint}_q$'s "rounding boxes."

2. Otherwise, the key-message pairs yield the *same* message digest μ on line 28 (original), i.e.,

$$\begin{aligned}\mu = \mathrm{H}(tr \parallel M) &= \mathrm{H}(tr' \parallel M') \\ \text{where } tr &= \mathrm{H}(\rho \parallel \mathbf{t}_1) \in \{0,1\}^{256} \\ \text{and } tr' &= \mathrm{H}(\rho' \parallel \mathbf{t}'_1) \in \{0,1\}^{256}.\end{aligned}$$

   Then we can obtain a collision in H: because $(pk, M) \neq (pk', M')$, either $M \neq M'$, in which case we have a collision $tr \parallel M \neq tr' \parallel M'$ in the "outer" H call (because tr, tr' have equal length), or $M = M'$ and $pk \neq pk'$ (i.e., $\rho \parallel \mathbf{t}_1 \neq \rho' \parallel \mathbf{t}'_1$). Then either $tr \neq tr'$, in which case we have the same collision as above in the "outer" H calls, or $tr = tr'$, in which case we have a collision $\rho \parallel \mathbf{t}_1 \neq \rho' \parallel \mathbf{t}'_1$ in the "inner" H calls.

1.3 Deterministic versus Randomized Hashing Salt

The original Dilithium specification defines both a randomized and deterministic signing mode; see [BDK+21, Section 3.2]. The only difference between these is in line 12 (original), where $\rho' \in \{0,1\}^{512}$ is defined to be either uniformly random or $\rho' := \mathrm{H}(K \parallel \mu)$; this simply uses $\mathrm{H}_K(\cdot) := \mathrm{H}(K \parallel \cdot)$ as a pseudorandom function to generate a fresh pseudorandom ρ' for each message digest μ. In general, the deterministic mode is preferred because it does not require a source of high-quality randomness at signing time (and low-quality randomness can potentially be catastrophic for security), whereas the PRF assumption is already relied upon in other parts of the design.

Similarly, the modified design defines both a randomized and deterministic mode for generating the hashing salt $r \in \{0,1\}^{512}$ for the message digest (line 10, updated). Here too, the deterministic mode simply uses H_K as a PRF to generate this salt.[5] However, because the message digest is not yet available (it is to be computed using the salt), there is little choice but to use the entire message M to determine the PRF input, since we wish to use distinct PRF inputs for each distinct message. Also, because the same PRF is used for two separate purposes (deriving both r and ρ'), domain separation is used: the all-zeros byte `0x00` is prepended to M (line 10, updated), and the all-ones byte `0xFF` is prepended to μ (line 12, updated). Of course, any other two distinct bytes, or equal-length binary strings, or other sound method of separating M from μ would do just as well.

Note that computing the digest using deterministic salt (line 10, updated) requires two separate hashing passes over the message M: one using H (for the salt), and one using SSS (for the digest). If this is prohibitive (e.g., because M is very long, or is "streamed" and not stored in memory), then the client may opt to use the randomized mode, or to securely "pre-hash" M into a short string, which would then be provided as input to the signing algorithm.

For random salt, we stress that imperfect randomness, or even no randomness at all, does *not* affect the correctness of signature verification, and is *not catastrophic* for security: using imperfect randomness merely corresponds to relying on a corresponding SPR-like property of the compression function, and using no randomness at all corresponds to relying on the collision resistance of the compression function, which is conjectured to hold with only moderately less concrete security than for the SPR-like properties.

[5] Using SHAKE for this purpose is not a problem for the SNARK, because the signature verifier does not (and cannot) re-derive the salt.

1.4 Security Analysis of the New Self-Target Problem

This section analyzes the new "self-target" problem that corresponds to forging in the modified scheme in a public-key-only attack. (We restrict attention to the key-only setting because signature queries are "zero knowledge" assuming the programmability of the random oracle H.) By definition, forging a signature for a message digest μ requires solving the following problem: given the public $\mathbf{A}$ and $\mathbf{t}_1$, find $(\alpha, \mathbf{z}, \mathbf{h})$ where $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$, the number of 1s in $\mathbf{h}$ is at most ω, and

$$\alpha = \mathrm{SS}(\mu \parallel \mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - \mathsf{Shuffle}(\mathrm{H}(\mu \parallel \alpha)) \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2)). \tag{1}$$

Apart from using $\mathbf{t}_1$ instead of $\mathbf{t}$ as an input (we use the former to get a tighter connection between forging and the self-target problem), there are three main differences (one of them merely syntactic) between this modified problem and the self-target problem for the original Dilithium scheme:

1. it applies $\mathrm{SS}(\mu \parallel \cdot)$ to the output of $\mathsf{UseHint}_q$;
2. it eliminates one of the two sequential calls to H, namely, the one that outputs $\tilde{c}$ as the input to $\mathsf{SampleInBall} = \mathsf{Shuffle} \circ \mathrm{H}$ (see Section 1.1.1);
3. it is written as an equation for α instead of for c. This is merely a syntactic difference, because by defining $c = \mathsf{Shuffle}(\mathrm{H}(\mu \parallel \alpha))$, Equation (1) above implies that

$$c = \mathsf{Shuffle}(\mathrm{H}(\mu \parallel \mathrm{SS}(\mu \parallel \mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2)))), \tag{2}$$

which is an equation for c (similar to the original self-target problem). In the other direction, Equation (1) can be obtained from Equation (2) by defining $\alpha = \mathrm{SS}(\mu \parallel \mathsf{UseHint}_q(\cdots))$.

Because H is modeled as a random oracle, each function $\mathrm{H}_\mu(\cdot) := \mathrm{H}(\mu \parallel \cdot)$ is an independent random function, so we can analyze the self-target problem for an arbitrary fixed digest μ. We next adapt the main approach for solving the original self-target problem considered in [BDK⁺21, Section 6.2.1] to the modified problem.

The adversary starts with one or more preimages $\mathbf{w}_1^{(i)}$ of the same subset-sum hash output $\alpha = \mathrm{SS}_\mu(\mathbf{w}_1^{(i)}) := \mathrm{SS}(\mu \parallel \mathbf{w}_1^{(i)})$ in mind, computes

$$c = \mathsf{Shuffle}(\mathrm{H}_\mu(\alpha)),$$

then attempts to find valid $\mathbf{z}, \mathbf{h}$ satisfying (for some i)

$$\mathbf{w}_1^{(i)} = \mathsf{UseHint}_q(\mathbf{h}, \mathbf{Az} - c \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2). \tag{3}$$

By substitution, it is clear that such $(\alpha, \mathbf{z}, \mathbf{h})$ would be a solution to the modified self-target problem.

In order for Equation (3) to hold, by [BDK+21, Lemma 1, Item 2], it must be the case that

$$[\mathbf{A} \mid \mathbf{I}_k] \begin{bmatrix} \mathbf{z} \\ \mathbf{u} \end{bmatrix} = c \cdot \mathbf{t}_1 \cdot 2^d + 2\gamma_2 \cdot \mathbf{w}_1^{(i)} \pmod q \tag{4}$$

for some $\mathbf{u}$ for which $\|\mathbf{u}\|_\infty \leq 2\gamma_2 + 1$. So, to execute this attack the adversary must solve a kind of (possibly) *multi-target* MSIS problem, where the adversary's known and chosen $2\gamma_2 \cdot \mathbf{w}_1^{(i)}$ all are "randomly shifted" by $c \cdot \mathbf{t}_1 \cdot 2^d$, where $c = \mathsf{Shuffle}(\mathrm{H}_\mu(\alpha))$ is a pseudorandom challenge.

First, if SS is collision resistant (which is conjectured to be the case), then the adversary can have just one preimage $\mathbf{w}_1 = \mathbf{w}_1^{(1)}$ of $\alpha = \mathrm{SS}_\mu(\mathbf{w}_1)$ in mind. Then Equation (4) can be simplified as

$$\mathbf{A}\mathbf{z} + \mathbf{u} = \mathbf{t}' := c \cdot \mathbf{t}_1 \cdot 2^d + 2\gamma_2 \cdot \mathbf{w}_1 \pmod q.$$

This can be seen as a variant of the inhomogeneous MSIS problem for instance $[\mathbf{A} \mid \mathbf{I}]$ with target $\mathbf{t}'$. The difference between this variant and the original MSIS problem is that here the adversary can indirectly influence the target $\mathbf{t}'$ by choosing $\mathbf{w}_1$ and computing the (pseudorandom) challenge $c = \mathsf{Shuffle}(\mathrm{H}(\mu \parallel \alpha))$ correspondingly; whereas in the standard MSIS assumption, the adversary is simply given a uniformly random target $\mathbf{t}'$ as input. Overall, the situation here is very similar to the one with the original self-target problem.

On the other hand, if SS is not collision resistant, then the self-target problem may become concretely somewhat easier, but still appears intractable. The adversary may be able to find some collisions, i.e., distinct $\mathbf{w}_1^{(i)}$ such that $\alpha = \mathrm{SS}_\mu(\mathbf{w}_1^{(i)})$; but note that the dependence on the message digest μ, which itself depends on the public key pk, appears to mitigate against pre-computing collisions that work across all messages or public keys. The adversary then needs to solve just one of the instances of Equation (4), where each $2\gamma_2 \cdot \mathbf{w}_1^{(i)}$ on the right-hand side is "randomly" shifted by the same $c \cdot \mathbf{t}_1 \cdot 2^d$. This is a variant of the *multi-target* MSIS problem, specifically: given $(\mathbf{A}, \{\mathbf{t}^{(i)} = 2\gamma_2 \cdot \mathbf{w}_1^{(i)} + c \cdot \mathbf{t}_1 \cdot 2^d\}_i)$, the adversary needs to output, for some i, some valid $(\mathbf{z}_i, \mathbf{u}_i)$ such that $\mathbf{A} \cdot \mathbf{z}_i + \mathbf{u}_i = \mathbf{t}^{(i)} \pmod q$.

Clearly, this multi-target problem is no harder than the above single-target MSIS variant. Moreover, in a hypothetical, pathological setting where *every* $\mathbf{w}_1^{(i)} \in R_q^k$ subset-sum hashes to the same $\alpha = \mathrm{SS}_\mu(\mathbf{w}_1^{(i)})$, the problem is trivially easy, because the adversary can just choose any valid $\mathbf{z}$ and $\mathbf{u}$ such that $\mathbf{A}\mathbf{z} + \mathbf{u} - c \cdot \mathbf{t}_1 \cdot 2^d \bmod q$ is a multiple of $2\gamma_2$, which yields the corresponding $\mathbf{w}$. However, in reality, only certain $\mathbf{w}_1^{(i)}$ will collide under SS_μ and be known to the adversary. It is reasonable to conjecture that the concrete hardness of the problem, as compared with the single-target problem above, decays at most linearly with the number of collisions $\mathbf{w}_1^{(i)}$ known to the adversary.

1.4.1 Self-Target Hardness Based (Loosely) on MSIS, in the ROM

When modelling H as a random oracle (with classical access), and assuming that SS is collision resistant, the standard MSIS problem (loosely) reduces to the modified self-target problem. This gives some additional evidence that the latter is hard, at least in an asymptotic sense.

The following proof sketch is adapted from [BDK+21, Section 6.1], with minor changes reflecting the form of the modified self-target problem. Let $\mathcal{A}$ be an algorithm against the self-target problem; we define a reduction $\mathcal{B}$ against the normal-form $\mathrm{MSIS}_{k,\ell+1,\gamma}$ problem, where

$$\gamma = \max(2\gamma_1, \tau \cdot 2^d + 4\gamma_2 + 2).$$

Given an input matrix $\mathbf{A} = [\bar{\mathbf{A}} \mid \mathbf{t} \mid \mathbf{I}_k] \in R_q^{k \times (\ell+1+k)}$, it defines $\mathbf{t}_1$ from $\mathbf{t}$ as in the scheme (where $\mathbf{t} = 2^d \cdot \mathbf{t}_1 + \mathbf{t}_0$ for "short" $\mathbf{t}_0$), invokes $\mathcal{A}$ on $(\bar{\mathbf{A}}, \mathbf{t}_1)$, and replies to each of $\mathcal{A}$'s distinct queries $\mathrm{H}(\mu \parallel \alpha)$ with a fresh uniformly random bit string s, from which anyone can derive $c = \mathsf{Shuffle}(s)$. If $\mathcal{A}$ returns a solution $(c, \mathbf{z}, \mathbf{h})$ to the self-target problem (in the form of Equation (2)) for some message digest μ, then $\mathcal{B}$ re-runs $\mathcal{A}$ with the same input and randomness, but reprograms the oracle's output on the "winning" query $\mu \parallel \mathrm{SS}(\mu \parallel \mathsf{UseHint}_q(\mathbf{h}, \bar{\mathbf{A}}\mathbf{z} - c \cdot \mathbf{t}_1 \cdot 2^d))$ to be a fresh random bit string s', which induces a distinct challenge $c' \neq c$ with very high probability.[6] If $\mathcal{A}$ outputs a solution $(c', \mathbf{z}', \mathbf{h}')$ to the self-target problem with message digest μ, then $\mathcal{B}$ outputs

$$\mathbf{x} := (\mathbf{z} - \mathbf{z}', c' - c, \mathbf{e} = \bar{\mathbf{A}}(\mathbf{z}' - \mathbf{z}) - \mathbf{t} \cdot (c' - c))$$

as the solution for the given normal-form $\mathrm{MSIS}_{k,\ell+1,\gamma}$ instance. It is apparent by substitution that $\mathbf{A}\mathbf{x} = \mathbf{0}$; below we analyze the norm of $\mathbf{x}$.

We now analyze the probability that $\mathcal{B}$ outputs a valid solution. By the forking lemma, the probability that in the above reduction, $\mathcal{A}$ produces two valid self-target solutions using the same "winning" random-oracle query $\mu \parallel \alpha$ is about $1/Q_{\mathrm{H}}$ times the square of advantage of $\mathcal{A}$ against the self-target problem.

Next, we show that when both runs of $\mathcal{A}$ are successful, $\mathcal{B}$'s output is indeed a solution to its given $\mathrm{MSIS}_{k,\ell+1,\gamma}$ instance. First of all, since the solutions are based on the same random-oracle query $\mu \parallel \alpha$ for the two executions of $\mathcal{A}$, the α part satisfies

$$\alpha = \mathrm{SS}\Big(\mu \parallel \mathsf{UseHint}_q(\mathbf{h}, \bar{\mathbf{A}}\mathbf{z} - c \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2)\Big) = \mathrm{SS}\Big(\mu \parallel \mathsf{UseHint}_q(\mathbf{h}', \bar{\mathbf{A}}\mathbf{z}' - c' \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2)\Big).$$

The inputs to SS must be equal with high probability, otherwise we can build an obvious reduction against the CR property of SS. Therefore, we have that

$$\mathsf{UseHint}_q(\mathbf{h}, \bar{\mathbf{A}}\mathbf{z} - c \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2) = \mathsf{UseHint}_q(\mathbf{h}', \bar{\mathbf{A}}\mathbf{z}' - c' \cdot \mathbf{t}_1 \cdot 2^d, 2\gamma_2).$$

By [BDK+21, Lemma 1, Item 2], we get

$$[\bar{\mathbf{A}} \mid 2^d \cdot \mathbf{t}_1 \mid \mathbf{I}_k] \begin{bmatrix} \mathbf{z} - \mathbf{z}' \\ c' - c \\ \mathbf{u} - \mathbf{u}' \end{bmatrix} = [\bar{\mathbf{A}} \mid \mathbf{t} \mid \mathbf{I}_k] \underbrace{\begin{bmatrix} \mathbf{z} - \mathbf{z}' \\ c' - c \\ (\mathbf{u} + c\mathbf{t}_0) - (\mathbf{u}' + c'\mathbf{t}_0) \end{bmatrix}}_{\mathbf{x}} = 0$$

for some "short" $\mathbf{u}, \mathbf{u}'$ such that $\|\mathbf{u}\|, \|\mathbf{u}'\| \leq 2\gamma_2 + 1$. Since $c \neq c'$ with high probability, the solution output by $\mathcal{B}$ is nonzero, as needed. By the verification condition, we have $\|\mathbf{z} - \mathbf{z}'\|_\infty \leq 2\gamma_1$, and because both c, c' are outputs of $\mathsf{SampleInBall}$, it holds that $\|c' - c\|_\infty \leq 2$. Finally, because $\|c\|_1 \leq \tau$ and $\|\mathbf{t}_0\|_\infty \leq 2^{d-1}$, we have $\|c\mathbf{t}_0\|_\infty \leq \tau \cdot 2^{d-1}$, and similarly for $c'\mathbf{t}_0$. Therefore, the ℓ_∞ norm of the solution $\mathbf{x}$ is at most $\gamma = \max(2\gamma_1, \tau \cdot 2^d + 4\gamma_2 + 2)$, as desired.

References

[BCJZ21] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of Ed25519: Theory and practice. In *IEEE Symposium on Security and Privacy*, pages 1659–1676, 2021. Cited on page 5.

[BDK+21] Shi Bai, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (version 3.1), February 2021. Available at `https://pq-crystals.org/dilithium`. Cited on pages 1, 4, 7, 8, 9, and 10.

[6] We assume without loss of generality that $\mathcal{A}$ queried H on the "winning" input, because $\mathcal{B}$ can "simulate" the query once it receives $\mathcal{A}$'s output in the first run.

[CDF$^+$21] Cas Cremers, Samed Düzlü, Rune Fiedler, Marc Fischlin, and Christian Janson. BUFFing signature schemes beyond unforgeability and the case of post-quantum signatures. In *IEEE Symposium on Security and Privacy*, pages 1696–1714, 2021. Cited on pages 5 and 6.

[CGN20] Konstantinos Chalkias, François Garillot, and Valeria Nikolaenko. Taming the many EdDSAs. In *Security Standardisation Research*, pages 67–90, 2020. Cited on page 5.

[GLP21] Yossi Gilad, David Lazar, and Chris Peikert. Subset-sum hash specification. Technical report, Algorand, Inc., 2021. Cited on page 1.

[HK06] Shai Halevi and Hugo Krawczyk. Strengthening digital signatures via randomized hashing. In *CRYPTO*, pages 41–59, 2006. Cited on pages 1 and 5.