Faster verifications and smaller signatures: Trade-offs for ALTEQ using rejections

Arnaud Sipasseuth[0000−0003−1048−4822]

KDDI Research Inc, Japan
xan-shipasata@kddi.com

Abstract. In this paper, we introduce a new probability function parameter in the instantiations of the Goldreich-Micali-Wigderson with Fiat-Shamir and unbalanced challenges used in ALTEQ, a recent NIST PQC candidate in the call for additional signatures.
This probability set at 100% does not bring any changes in the scheme, but modifies the public challenge generation process when below 100%, by injecting potential rejections in otherwise completely valid inputs. From a theoretical point of view, this does not improve the asymptotical hardness of the scheme and negatively affects the efficiency of the signatory, and might itself seem trivial. However, from a practical point of view, implementation-wise and performance-wise, this triviality allows an extra degree of freedom in optimizing parameters, as the heuristic security level is also increased against forgers: previously valid combinations now can be deemed invalid. This allows us to make trade-offs to reduce the computational load in verifiers, accelerating verifications, marginally reduce the signature size, at the cost of making signatures slower and unlikely to be constant-time. In particular, this extra degree of freedom allows to make implementation choices that enable smoother and faster executions of the aforementioned protocols, especially in the context of parallelization using vectorized instructions. We also demonstrate the usefulness of our proposal to ALTEQ for other options, when slowing down the signing process is not an issue: significantly smaller signatures but longer verifications, or lower public key sizes. The ideas presented apply to any primitive, and can be used beyond ALTEQ.

Keywords: Post-Quantum Cryptography · Signature scheme · Alternate Trilinear Forms · AVX2

1 Introduction

Most of the cryptographic research on primitives in the past few years have been focused on building cryptosystems based quantum-resilient problems, motivated by the threat of quantum computers. Indeed, it is unclear if practical quantum computers will ever the light of the day, but most of the currently used cryptosystems today, namely RSA [17] would be broken by quantum algorithms [20]. A call for standardization has been launched by the NIST and has met an unsatisfactory conclusion recently, standardizing a few schemes [15]. Unsatisfactory,

as it seems like the currently selected portfolio is not as diverse as it expected: an additional call for signatures has been launched as a result [16]. The intent is clear: the NIST asked to focus on short signatures and fast verification speed, or *any form of significant advantage over the currently standardized schemes.*

One of the potential advantages over any of the selected schemes, obviously, would be to remain secure if the currently chosen schemes are completely broken. The easiest way to construct such a scheme is to use a *distinct* primitive. The most popular family of primitives are currently lattice-based, hash-based, code-based, multivariate-based and isogeny-based. One family tends to fly under the radar though: group action-based cryptography. Recently, this research topic has gained some traction: new workshops have been planified this year to focus on group action-based cryptography[1], and at least three group action-based constructions have been submitted to answer NIST's call: ALTEQ [6], MEDS [7] and LESS [1], the latter two being code-based groups but ALTEQ and MEDS sharing the same class of complexity as far as theoretically hard problems are concerned (Tensor Isomorphism Class).

In this paper, we focus on the cryptographic construction used by ALTEQ, which framework can be used for any group action-based cryptography primitive. The ALTEQ signature scheme, relying on the (ATFE) problem, is constructed via a Fiat-Shamir (FS) transform [10] over a Goldreich-Micali-Wigderson (GMW) framework [12], with unbalanced challenges. Using this framework without carefully checking the security conditions can lead to devastating attacks: for example, ALTEQ requires their entries to be invertible matrices, but did not implement a check. Because of this blunder, a forgery attack was found by Saarinen [18] less than two days after ALTEQ's publication, shortly followed by a quick fix from Beullens [3] (who also provided further analysis, affecting somehow similar schemes such as [1] and [7]). The fix was implemented, leaving us with the current version[2].

Despite this update, ALTEQ did not change its parameters, and merely the means to compute the algebraical data composing the signature. The updated scheme could use new parameters: the current version of their code would yield better performance from parameter changes, since the initial parameters were also chosen to optimize performance in the first, no longer used, initial version. Changing the parameters of their construction requires some work, but is very customizable. The GMW-FS framework with unbalanced challenges needs to fix three parameters to be instantiated:

- r, which decides the number of primitive computations. r determines the verification speed, thus lowering it directly accelerates the verification.

[1] `https://aimath.org/workshops/upcoming/postquantgroup/`, `https://www.ihp.fr/en/news-research-activities/t3-2024-post-quantum-algebraic-cryptography-0`

[2] Following the update of `https://eprint.iacr.org/2024/364` *after* the ACISP deadline, the ALTEQ parameters used for practical comparison in this work are no longer relevant. This work's contribution is however positively affected (larger trade-offs).

- K, which decides the number of challenges solved using the secret key, and is the main culprit in the signature size: lowering K directly decreases the signature size.
- C, which decides the amount of possibilities for the challenge: it essentially decides the size of the public key.

Thus if we want to get closer to NIST's requests (although not guaranteeing that we can fulfil them), we should aim to decrease both r and K. This led ALTEQ to propose what they call "large public key" parameters alongside "fully equilibrated" parameters, with low r and K but with incredibly large parameter C: to achieve λ bits of security, the following formula needs to be verified:

$$\binom{r}{K} C^K \geq 2^\lambda$$

While there are broad choices of parameters available for any value λ, we often get into awkward choices where we have to pick between slightly higher values for one parameter just to meet security requirements, but heavily complexifying the optimizations in the process. In particular, we aim in this paper to provide trade-offs that extend the available parameters, allowing to optimize vectorized implementations as an example.

Contributions We propose to generalize the GMW-FS framework with unbalanced challenges by adding a fourth parameter p to the original triplet (r, K, C), which simply changes the formula to achieve λ-bits of security to

$$p^{-1} \binom{r}{K} C^K \geq 2^\lambda$$

where p is a probability function that is almost fully customizable and is equal to 1 in the classic case. In particular, the function is defined by

$$p = \frac{x}{2^b} \text{ with } x \in \mathbb{N}, 0 < x \leq 2^b$$

and rather than implementing a whole new function in the ALTEQ code, this is simple to implement by modifying the challenge generation to generate b *extra* bits, by fully exploiting the fact that the Keccak family of hash functions[13], namely SHA3, have a flexible output size with customizable security. Then a combination will be deemed valid whenever *this extra sample of b bits is lower than x*, effectively giving us a multiplicative probability p of acceptance. This parameter p has almost no impact on the computational time of the verification, thus we can accelerate the verification process by using $p < 1$ to reduce r, which can also be used to decrease the signature size by reducing K. However, the lower is p, the more the average signing time increases, thus for a proper trade-off we need to carefully analyse each set of parameters and the targeted use case. In particular, we demonstrate how the introduction of p can facilitate the implementation of the ALTEQ verification function, by simultaneously decreasing its signature size and accelerating its verification: a win-win situation

if having a non-constant time, slower signature process is not a problem. We show other examples enabled by our technique, such as decreasing the public key size by 34% without touching (r, K), or decreasing the signature size by 18% by reducing K while increasing r to an AVX2 friendly value. Note that this contribution is mostly targeted towards the scope of fixing practical parameters, when implementing a scheme with low-level languages like C or plain assembly: there is no *asymptotical* change on the hardness of the constructions, nor is there a paradigm shift in cryptanalysis.

Organization of the paper In section 2, we describe the ATFE problem and its variants, especially the ones that are relevant to us, briefly reintroduce the ALTEQ cryptosystem and the associated structures, which is in short the ATFE primitive used for a GMW-FS construction with unbalanced challenges. In section 3, we present our change, and explain how we determine the impact on the security and the formulas to compute both the failure rate associated with the signatory and the potential gains implementation wise. In section 4, we apply our suggestion to ALTEQ, and show how we can have more "friendly" parameters implementation wise, especially for AVX2. The code used to obtain all tables is present in the appendix 5. We finally conclude and list open questions that arise from this paper in section 5, and discuss future research directions.

2 Background

Here we give some reminders and notations about the essential information about ALTEQ that is necessary to understand this paper. Referring to [6] is recommended, but not necessary.

2.1 Basic notations

Mathematical notations

1. $q, n \in \mathbb{N}^*$: field orders and dimensions respectively. q is a prime power.
2. $\mathbb{F}_q$: the finite field of order q.
3. $\mathbb{F}_q^n$: vector space of $n \times 1$ vectors over $\mathbb{F}_q$.
4. $\mathrm{GL}(n, q)$: group of invertibles $n \times n$ matrices over $\mathbb{F}_q$.
5. For $u \in \mathbb{F}_q^n$ and $A \in \mathrm{M}(n, q)$, u^{t} and A^{t} denote their transposes.
6. For $a, b \in \mathbb{N}^*$, $\binom{a}{b}$ is the binomial coefficient.
7. For $a < b \in \mathbb{N}$, $[\![a, b]\!] = [a, b] \cap \mathbb{Z}$
8. For $m \in \mathbb{N}^*$, let $[m] = [\![1, m]\!]$.
9. Given S finite, $a \in_R S$ means a is a uniformly random sample from S.

Cryptographic scheme parameters

1. λ the desired bit-security level.
2. (n, q) the algebraic structure parameters, n, q as defined above.
3. (r, K, C) the GMW-FS framework parameters.

Gross notation abuse To save space and to reduce repetitive clutter we denote

- any collection $\{X_i\}_{i\in S}$ by $\{X_i\}_S$, it should be clear that i serves as an index of items X and is an element of S.
- $\mathsf{App}(\mathsf{S}, s)$ appends an element s to the end of an *ordered* list S.
- for S an *ordered* set of cardinal n, $||\mathsf{S}$ represents a concatenation with all its elements i.e $||s_1||...||s_n$. This is mostly used in hash function parameters where ordering is important, as a permutation change the hash output.

2.2 Algebraic structures in ALTEQ

A trilinear form (TF) on $\mathbb{F}_q^n$ is a map $\phi : \mathbb{F}_q^n \times \mathbb{F}_q^n \times \mathbb{F}_q^n \to \mathbb{F}_q$ that is $\mathbb{F}_q$-linear in each argument. It is *alternating* (ATF) if and only if
$\forall u, v \in \mathbb{F}_q^n$, $\phi(u, u, v) = \phi(u, v, u) = \phi(v, u, u) = 0$.
$\mathrm{ATF}(n, q)$ denotes the linear space of all ATFs on $\mathbb{F}_q^n$. $\mathrm{GL}(n, q)$ naturally acts on $\mathrm{ATF}(n, q)$: for all $u, v, w \in \mathbb{F}_q^n$, $A \in \mathrm{GL}(n, q)$,
A sends ϕ to $\phi \circ A$, defined as $(\phi \circ A)(u, v, w) := \phi(A^{\mathrm{t}}(u), A^{\mathrm{t}}(v), A^{\mathrm{t}}(w))$
This action defines an equivalence relation $\cong$ on $\mathrm{ATF}(n, q)$, namely
$\phi \cong \psi$ means $\exists A \in \mathrm{GL}(n, q)$ s.t $\phi = \psi \circ A$.

The ALTEQ cryptosystem is based on the hardness of the following ATFE problems:

- Search version: given $\phi \cong \psi$, find $A \in \mathrm{GL}(n, q)$ s.t $\phi = \psi \circ A$
- Identification version: given $\phi, \psi \in \mathrm{ATF}(n, q)$, is $\phi \cong \psi$ true?

Note that there are two ways in ALTEQ to represent an ATF. One is a compressed form and the other is called uncompressed. Each time a group action is computed, the ATF is first uncompressed, vectorized, and the result is compressed and unvectorized. In the current implementation, it is a tedious operation that slow the scheme. Another key algebraic structure in ALTEQ is the use of "column matrices", which were essential in patching Saarinen's attack *while* improving ALTEQ's performance. It is also not essential in this paper to know the details, just keep in mind that it is a convenient form for computing group actions. For every $A \in \mathrm{GL}(n, q)$, A^{col} is its column representation *when it exists*. It is analogue to the LU decomposition: with a permutation matrix P, a decomposition PLU always exist, but forcing $P = Id$ removes the guaranteed existence of such decomposition.

2.3 ATFs and group actions in algorithms

Algebraic operations. The following functions define the computations

- $\{\phi_i \circ A_i\}_{[c]} \leftarrow \mathsf{ActATF}(\{\phi_i, A_i^{col}\}_{[c]})$
- $\{\phi_i \circ A_i^{-1}\}_{[c]} \leftarrow \mathsf{InvAct}(\{\phi_i, A_i^{col}\}_{[c]})$
- $\{A_i^{col}\}_{[c]} \leftarrow \mathsf{ColDec}(\{A_i\}_{[c]})$
  If A^{col} does not exist for a given A, a flag is raised to indicate failure.
- $\{C_i\}_{[c]} \leftarrow \mathsf{ColMul}(\{A_i^{col}, B_i^{col}\}_{[c]})$
  Note that given A, B in column form, this return $C = A \times B$ and not C^{col}.

Randomness generation. Hashing is done using the Keccak (SHA-3) family of functions [13]. Expanders use AES-CTR [2] and take a seed of λ-bits as an input. Names are self-explicit:

- H is a hash function that takes an input of arbitrary length and output a binary string from $\{0,1\}^{2\lambda}$
- expCha is used for generating "unbalanced" challenges. (r, K, C) being fixed, it will output r indexes $\{b_i\}_{[r]} \in [\![0, C]\!]^r$ such that *exactly* $r - K$ indexes have $b_i = 0$ (and thus exactly K indexes have $b_i \in [C]$)[3].
- expATF outputs a random $\phi \in \mathrm{ATF}(n, q)$.
- $\mathsf{expCols}$ outputs a random decomposition A^{col} of some $A \in \mathrm{GL}(n, q)$
- $\mathsf{expSeeds}$ outputs some specified number of seeds of the same size λ-bits.

2.4 The ALTEQ cryptosystem

We briefly present the generic pseudocodes of ALTEQ's setup, signature and verification processes in figure 1. Their description here is slightly modified compared to the original document [6], to make it simpler to understand and lighter to write without affecting the essential features of the scheme.

Current data sizes of ALTEQ Following the description, we have

$$\begin{aligned}
\mathsf{pk}_{\text{size}} &: C \cdot \binom{n}{3} \cdot \lceil \log_2(q) \rceil + \lambda \\
\mathsf{sk}_{\text{size}} &: \lambda \\
\mathsf{Sig}_{\text{size}} &: (r - K + 2) \cdot \lambda + K \cdot n^2 \cdot \lceil \log_2(q) \rceil
\end{aligned}$$

3 The change: affecting both forgers and signatories

3.1 Attack method of the forger

Before we introduce our modification, we present the attack angle we consider to make sure we remain safe post-modification. This attack is the only attack that has been done in practice on ALTEQ: Saarinen's attack [18]. Since then, it has been patched, but the methodology is sound (even if currently ineffective). Let us introduce the attack in the general case.

To forge a signature, an attacker has to first pick K positions among r, then one position among the public ATF $\phi_{i \in [\![C]\!]}$ for each of those K matrices, and compute the group action from those chosen ATFs by those K matrices. Then, pick one seed, and compute the ATFs corresponding to the group action of ϕ_0 by the matrix expanded from one seed.

[3] In the original ALTEQ specification [6] (and in the code), the role of index C and 0 are swapped. We just inverted in this paper for convenience.

KGen

1 : $\mathsf{sk} \leftarrow_R \{0,1\}^\lambda$
2 : $\{\delta_i\}_{[\![0,C]\!]} \leftarrow \mathsf{expSeeds}(\mathsf{sk}, C+1)$
3 : $\phi_0 \leftarrow \mathsf{expATF}(\delta_0)$
4 : $\{\Delta_i^{col}\}_{[C]} \leftarrow \mathsf{expCols}(\{\delta_i\}_{[C]})$
5 : $\{\phi_i\}_{[C]} \leftarrow \mathsf{InvAct}(\{\phi_0, \Delta_i^{col}\}_{[C]})$
6 : $\mathsf{pk} \leftarrow (\{\phi_i\}_{[C]}, \delta_0)$
7 : **return** $(\mathsf{pk}, \mathsf{sk})$

Vf$(\mathsf{pk}, M, \mathsf{Sig})$

1 : $\phi_0 \leftarrow \mathsf{expATF}(\delta_0),\ a \leftarrow 0,\ b \leftarrow 0$
2 : $\{c_i\}_{[r]} \leftarrow \mathsf{expCha}(\mathsf{Ch})$
3 : **for** $i \in [r]$ **do**
4 : **if** $c_i = 0$ **then** $a \leftarrow a+1$
5 : $D_i'^{col} \leftarrow \mathsf{expCols}(\mathsf{s}_a)$
6 : **else** $b \leftarrow b+1$
7 : $D_i'^{col} \leftarrow D_b^{col}$
8 : $\{\psi_i'\}_{[r]} \leftarrow \mathsf{ActATF}(\{\phi_{c_i}, D_i'^{col}\}_{[r]})$
9 : $\mathsf{Ch}' \leftarrow \mathsf{H}(\mathsf{H}(M)||\{\psi_i'\}_{[r]})$
10 : $\mathsf{F} \leftarrow (\mathsf{Ch} \neq \mathsf{Ch}')\ or\ (\text{bad } \{D_i^{col}\})$
11 : **if** F **then return** No
12 : **return** Yes

Sign(sk, M)

1 : $\{\delta_i\}_{[\![0,C]\!]} \leftarrow \mathsf{expSeeds}(\mathsf{sk}, C+1)$
2 : $\phi_0 \leftarrow \mathsf{expATF}(\delta_0)$
3 : $\beta \leftarrow_R \{0,1\}^\lambda$
4 : $\{\mathsf{s}_i\}_{[r]} \leftarrow \mathsf{expSeeds}(\beta, r)$
5 : $\{B_i^{col}\}_{[r]} \leftarrow \mathsf{expCols}(\{\mathsf{s}_i\}_{[r]})$
6 : $\{\psi_i\}_{[r]} \leftarrow \mathsf{ActATF}(\{\phi_0, B_i^{col}\}_{[r]})$
7 : $\mathsf{Ch} \leftarrow \mathsf{H}(\mathsf{H}(M)||\{\psi_i\}_{[r]})$
8 : $\{c_i\}_{[r]} \leftarrow \mathsf{expCha}(\mathsf{Ch})$
9 : $\mathsf{S} \leftarrow \{\},\ \mathsf{I}_\Delta \leftarrow \{\}$
10 : **for** $i \in [r]$ **do**
11 : **if** $c_i = 0$ **then** $\mathsf{App}(\mathsf{S}, \mathsf{s}_i)$
12 : **else** $\mathsf{App}(\mathsf{I}_\Delta, i)$
13 : $\{\Delta_{c_i}^{col}\}_{\mathsf{I}_\Delta} \leftarrow \mathsf{expCols}(\{\delta_{c_i}\}_{\mathsf{I}_\Delta})$
14 : $\{D_i\}_{[K]} \leftarrow \mathsf{ColMul}(\{\Delta_{c_i}^{col}, B_i^{col}\}_{\mathsf{I}_\Delta})$
15 : $\{D_i^{col}\}_{[K]} \leftarrow \mathsf{ColDec}(\{D_i\}_{[K]})$
16 : **if** ColDec failed **then** go to line 3
17 : **return** $\mathsf{Sig} = (\mathsf{Ch}, \mathsf{S}, \{D_i^{col}\}_{[K]})$

Fig. 1: The ALTEQ cryptosystem

The forgery goes on by attempting to compute the challenge from the chosen ATFs, then expanding the challenge and succeeding if the initial choices for the K positions among r, and the C choices of public ATFs in each of the K positions match. Thus, each attempt has a success chance of 1 out of $\binom{r}{K}C^K$:

- $\binom{r}{K}$ for the K choices among r
- C^K for the C choices of public ATFs for each of the K positions.

Note that because the challenge is attached to a message, knowing a signature for one message does not help forging for another message assuming the hashes and the expanders are secure.

Saarinen's attack used a zero matrix instead of K invertible matrices, which effectively reduced the number of combinations to $\binom{r}{K}$ since all K ATFs were

similarly set to the zero value. Now that ALTEQ checks invertibility, Saarinen's attack does not apply anymore, but the general framework remain sound. There are of course other attacks, of algebraic nature, that targeted mostly ALTEQ's predecessor [21,5]. Purely algebraic attacks do not concern this paper, as we do not modify the algebraic structure.

3.2 Introducing the probability parameter p

Let us remind the objective of this paper:

- Have better control of the parameters (r, K, C), to shorten signatures and/or accelerate verifications.
- Keep the security analysis as close as possible to the original ALTEQ.

Thus the idea in this paper comes by following this train of thought:

- The security of the verification is enforced essentially by the public, secure function expCha, and the security of SHA3 and AES-CTR.
- expCha's outputs decide if a signature is valid or not.
- expCha essentially gives out random positions for the challenges.
- With the secret key, every challenge can be solved[4].

This is the core difference between a forger and a signatory: a signatory can answer any challenge thanks to being able to solve the ATFE instances requested. However a signatory has *no control* over what is the challenge going to be: expCha, SHA3 and AES-CTR are considered almost unpredictable and irreversible. But what if some of the ATFs input are considered invalid by those same functions, *before* or *at the same time* a challenge is even requested? Essentially, the difference between a signatory and a forger becomes essentially an advantage factor of $\binom{r}{K}C^K$ in success chance! The signatory himself will not be able to guarantee that any series of ATFs leads to a challenge he can answer, since the series of r ATFs can be rejected. However, the forger is equally affected, worsening its attack angle we presented above.

So the idea is to reject previously valid combinations, and the more we reject, the harder it is for *both* signatories and forgers to produce a valid combination. To do so, we make use of the fact that Keccak can expand any number of bits we wish, and usually just expand enough bits to create the challenge. We use this property to extend *more* than the challenge. We present two options, where in both options $\mathsf{H}(m)$ must be maintained as part of the decision process to make sure every message comes with its own "combination challenge", i.e *knowledge of valid combinations/signatures for a message do not help learning the valid combinations for another message*. This should also help the reader visualize how we consider security risks.

[4] All theoretically, but not all of them in practice if we limit the form of matrices to be in column form, for example.

- First option, simultaneously with the challenge: we request the hash that produces the challenge Ch to produce b extra bits, and then denote $y \in \mathbb{N}$ the b-bits integer produced. If $y < x \leq 2^b$ then we accept the challenge generated, otherwise we reject it, giving a probability of acceptance of $p = x/2^b$. This affects the signatory, where every permutation is no longer valid, but the signatory can still sign *any* valid permutation: thus the signatory has to search for a valid permutation, and can sign any valid permutation with very high probability (the same as in ALTEQ, resampling depending on the success of column factorization). A forger on the other hand, needs to make sure his permutation is valid before checking if his combination choice is correct.
- Second option, post-challenge: we request that a hash of the signature choice, i.e from data obtained after the computation of the challenge, produce b extra bits, and proceed as above. It is not clear if this process is more or less expensive that the latter one: we request an extra hash, but this extra hash has lower input size and we don't extend the output of the challenge hash which has an extremely large input. We could even include the challenge Ch obtained as a parameter: the point of this second option is to have the input of $\mathsf{H}(m)$ influence the decision process to maintain statistical independence between distinct messages, but use a lower size entry for the extra b-bits.

Among all those options, we chose the first one in this paper. The first one is the simplest to implement, probably has the lowest cost for a verifier, and also simplifies the visualization of the attack cost which we explain in the next subsection. The second one is more or less an idea on how we could link the rejection to both the message and the signature combination in a more efficient manner while maintaining the security: it is not clear how to proceed and is better left as an open question as it only affects the signatory (and the forger), which we already assumed will be hindered.

3.3 Cost for a forger, parametrization

We previously needed to guarantee that the number of combinations $\binom{r}{K}C^K$ was above 2^λ. Now, even if a valid combination is found, there is only a chance $x/2^b$ that the combination is part of a valid signature. So if previously, every forging choice, i.e a choice of K positions among r and a choice of K ATFs among C, has 1 chance out of $\binom{r}{K}C^K$ to be correct for a successful forgery, this chance has now a factor $p = x/2^b \leq 1$, making the probability of success to be $p \leq 1$ out of $\binom{r}{K}C^K$, i.e 1 out of $p^{-1}\binom{r}{K}C^K$, hence our reasoning of the security requirement of

$$p^{-1}\binom{r}{K}C^K \geq 2^\lambda \text{ or } x^{-1}\binom{r}{K}C^K \geq 2^{\lambda-b}$$

in particular, a perfect equality can easily be set for $b = \lambda$ and $x = \binom{r}{K}C^K$ if we aim to maximize the success probability given fixed values (r, K, C).

Note that the failure rate is basically $1 - p$, i.e $(2^b - x)/2^b$: we cannot have values of x that are too low, i.e values $\binom{r}{K}C^K$ that are much lower than 2^λ: doing so would slow down the signatory by a significant factor, as like the forger, it also needs to find a valid combination before answering the challenge itself. We stress again that we assume in this paper that the *only* advantage a signatory has over a forger is a factor $\binom{r}{K}C^K$, which is the original security assumption in ALTEQ, proven to be "EUF-CMA" [18][5]. It is possible this assumption is wrong but we have not seen any evidence suggesting the opposite.

3.4 Permuting the "valid" challenges: not a security concern

Since the signatory now has to find a valid permutation of ATFs *before* answering the challenge using the secret key, we propose a smarter way than recomputing from scratch all the ATFs, which uses expensive calls to ActATF. Instead, notice how any permutation of the ATFs provide a new hash value through SHA3, both the challenge and the b extra bits. Thus we just need to repermute the computed ATFs until the combination is valid.

A concern is now whether we lose security by doing so, or how probable that a combination exists and can be found by permutation. Let us recall that

- The number of permutations is $r!$. For the lowest proposed ALTEQ parameter $r = 16$ we have $\log_2(r!) \geq 44$, and for the second lowest $r = 39$ we have $\log_2(r!) \geq 153$. If we sample them randomly, for example via a Fisher-Yates method, until we are likely to find back a previously sample above 50% chance, that is still the square root of the size, which is plenty of tries. Otherwise we can systematically generate them all incrementally, but we will lose the randomness of the order of permutations.
- Supposing we want to fix x, b to limit the number of resampling for practical uses, we can safely assume that we will always have enough permutations available. At least as long as we permute them by units and properly sample among all permutations.

Using permutations rather than recomputing the ATFs is a trick a forger can use: and since a forger can use it then there is no reason a signatory should not use it, as the end result does not affect the algebraic security. In particular, there is no guarantee that a "success at first-try" does not hold an invalid permutation when correctly signing either. Plus, we argue that using permutations *might* improve security: imagine that giving random consecutive bit-strings with some holes (K holes per signature) allows to have some chance to recover the initial random seed that was expanded, then a permutation would hide the structure. Of course, since AES-CTR is believed to be secure we have no reason to enter such a scenario. But if the initial order is secure why would a permutation not be? Could we even forge an attack knowing that the signature is the result of a permutation, when the initial data is completely unknown? This seems highly

[5] assuming the hardness of ATFE, SHA3 and AES-CTR.

unlikely.

One other concern using permutations is its potential expensive cost. However, we do not need purely random permutations: since the initial data is already random (generated via AES-CTR in ALTEQ), it should not be a worry. Thus we could rely on completely deterministic incremental algorithms. When r is prime, S_r can be generated by two elements: a cycle and a permutation, which lead to simple and fast algorithms to generate all permutations. In our case we aim for r to be a multiple of the register size, so generic deterministic algorithms such as Heap's algorithm [14] or Trotter's algorithm [22] could work, removing the necessity of any call to AES-CTR or SHA3 for this part of the procedure. Note that the above are generative algorithms for the set of all permutations, but they can be adapted to compute the next permutation in line and discard the previously generated one (through any kind of order with or without a storage of an intermediate state).

A more intensive read on the matter can be found in [19]. For the rest of this paper, we can assume we sample "randomly", with a Fisher-Yates method for example [11], but in practice it might be a wiser choice to iterate through them all. Heap's algorithm *can easily generate billions of permutations per second*[6] so asking for a *few hundreds permutations* should not be a problem, especially that we do not need to record them in our case. The main problem though, might be the cost of hashing: to deem if a permutation is valid, we *currently* need to re-hash the whole ATFs, but this is a problem that might be better to tackle in further work. Nevertheless, this only concerns the signatory: a verifier preferring AVX2 parameters, our main target in this paper, is unaffected.

3.5 Challenge from vectorized data and permuting them

One of the main objectives of this paper is to obtain parameters that are friendly for vectorization, which affects the verification process since rejections and permutations have little impact on the verification itself. The current vectorization process in ALTEQ is done automatically by `GCC` detecting convenient for loops after the data has been manually interwoven (`GCC` did not interweave data for us). Vectorization works best when interweaving of data is done smoothly and does not leave any hole: having convenient data sizes even allow to have interweaving "in-place", with lower memory requirements. For example, Dilithium's handwritten assembly code [9] uses specific instructions to interweave data, but this is more complex to do in ALTEQ due to unmatching data sizes, noting that the ALTEQ code is generic `ANSI C` and works for *all* parameters, unlike Dilithium's handwritten assembly per parameter set. In particular, we wish for the similar operations to be repeated over concatenated data of size multiple of the register size. For AVX2, this is 256-bits registers, and 512-bits registers for

[6] see slide 19/21 of a 20+ years old presentation by Sedgewick himself `https://sedgewick.io/wp-content/uploads/2022/03/2002PermGeneration.pdf`.

AVX512. Since ALTEQ for example uses 32-bits integer entries, this corresponds to packs of $R = 8$ integers in AVX2 or $R = 16$ integers in AVX512, where we denote R as a number of entries per vector.

ALTEQ's code regarding the computation of ATFs, i.e the function ActATF, first interweave the different ATFs (to put them into "vectorized form"), then untangle the ATFs after the group action computation ("unvectorize" them). The untangling is necessary to compute the challenge Ch through concatenation and hashing. We argue that we can in fact change the technicalities of the challenge generation, assuming that r, the number of ATFs, are a multiple of the register size i.e $r \mod R = 0$.

- Ch is generated from hashing ATFs concatenated with a message hash.
- This by no means use any form of algebraic structure: this process is in fact primitive independent. The ATFs are only seen as bit-strings.
- We could then hash the interwoven data and not affect Ch's security.
- Thus in theory, we do not need to untangle data and directly hash the "vectorized form": there is no loss of randomness in the bit-string shape, and this automatically improves the verifier performance.

However, when r is not a multiple of the register sizes, then a problem arise with this approach: there is a lot of trailing zeroes/random data at the end of each data position string (i.e each position in the ATF). We would need to either concatenate the data, adding some extra operations over non-constant data shift sizes, or consider that the trailing zeroes are part of the challenge computations, extending an already big entry size by a useless string of size as big as at most $R - 1$ ATFs with no added security.

As the challenge generation needs to be public and used in all parties, everything we do to accelerate the verifier needs to be reflected in the signing process: thus, the concept of permuting data to generate new challenges will be impacted if data stays in vectorized form. We suggest a simple technique to get around the problem: instead of permuting r ATFs, we permute packs of R ATFs. This reduces the amount of interweaving/untangling that we need to perform in the signature process, but this can only be done realistically when p is not too low compared to the amount $(r/R)!$ available (which for $r = 16$ for example is only $(r/R)! = 2$ for AVX2). We could also resort to some partly hardcoded attempts using AVX2 intrinsics to shuffle 32-bit integers *within* registers, such as `vpermilps`, `vperm2i128`, `vperm2i128`, `vpermd`, etc... But this would require further deeper studies into the properties of permutation generation, as a hybrid algorithm hardcoded/exhaustive algorithm must be researched (we have not found any). Again, note that only the signing process would be slowed down: the verification remains unaffected by the permutation operations.

3.6 Modified procedure

To describe the modified procedure we first define new functions/system parameters:

- p is a probability $p = x/2^b$, where x, b are system parameters.
- L is a system parameter representing how many permutations we would like to attempt before rerolling r new ATFs. If using exhaustive permutation generation algorithms, L can be set exactly to $r!$: no point doing more. For the rest of this paper, we will arbitrarily set L to obtain 99% success chance before resampling new ATFs, which should be several orders of magnitude less than $r!$.
- H_p is exactly the same as the hash function H, but outputs an additional boolean that is True with probability p and False otherwise. This is done by requesting H to produce an extra integer y of b-bits and answering whether $y/2^b < p$. Since Keccak has flexible output this is trivial to implement in practice (given the original function).
- $\mathsf{permATF}$ is a function that takes into parameter a string of r ATFs and a random integer seed $\alpha \in [\mathsf{L}]$, and output another string of r ATFs that are a permutation of the input following the random seed α.

Which transform the current ALTEQ scheme to one we present in figure 2. We stress that p, L are system parameters, and it is up to the cryptographer to set the limits to its convenience. In section 4, we present of modifications of (r, K, C), keeping p to the *maximal value required* (i.e minimizing p^{-1}) to reach λ-bits of security, i.e

$$p = \frac{\binom{r}{K} C^K}{2^\lambda} \quad \text{so that} \quad p^{-1}\binom{r}{K} C^K = 2^\lambda$$

but it is possible to use another approach: limit p and see if (r, K, C) can be adapted accordingly. Whenever p is shown in a table, namely in section 4, we will display an approximation instead of the rational exact value. L will also be calculated in the tables of section 4, as the *minimal number of permutations needed* to reach a 99% chance to get *at least one valid permutation* out of L permutation samples for a signatory.

Note on Beullens' attack After the ACISP deadline, a new attack by Beullens [4] forced ALTEQ to adapt, now enforcing *distinct* ATF pools *per position*, with seeds now appended with a salt and a *round position index*. This does not mean the above technique does not *strictly* apply: there are other ways to counter the attacks *and* allow permutations, as increasing the salt size and/or the seed size and not using a round position index, for example.

Concerning side-channel attacks Searching for the correct permutation lowers the chance of a constant-time implementation, however the added permutation procedure only deal with *public data*: no operation using secret information is used during the new non-constant-time part of the signing function, and is even information that is used by the verifier *in the original* ALTEQ. In fact, the signatory could even publish publicly the list of the permutations that were tried and failed: it might be a long list, but a long list with zero secret information.

$\mathsf{Vf}(\mathsf{pk}, M, \mathsf{Sig})$

1 : $\phi_0 \leftarrow \mathsf{expATF}(\delta_0),\ a \leftarrow 0,\ b \leftarrow 0$
2 : $\{c_i\}_{[r]} \leftarrow \mathsf{expCha}(\mathsf{Ch})$
3 : **for** $i \in [r]$ **do**
4 : **if** $c_i = 0$ **then** $a \leftarrow a + 1$
5 : $D_i'^{col} \leftarrow \mathsf{expCols}(\mathsf{s}_a)$
6 : **else** $b \leftarrow b + 1$
7 : $D_i'^{col} \leftarrow D_b^{col}$
8 : $\{\psi_i'\}_{[r]} \leftarrow \mathsf{ActATF}(\{\phi_{c_i}, D_i^{'col}\}_{[r]})$
9 : $\mathsf{Ch}', \mathsf{F} \leftarrow \mathsf{H}_p(\mathsf{H}(M)||\{\psi_i'\}_{[r]})$
10 : **if** not F **then return** No
11 : $\mathsf{F} \leftarrow (\mathsf{Ch} \neq \mathsf{Ch}')$ *or* (bad $\{D_i^{col}\}$)
12 : **if** F **then return** No
13 : **return** Yes

$\mathsf{Sign}(\mathsf{sk}, M)$

1 : $\{\delta_i\}_{[\![0,C]\!]} \leftarrow \mathsf{expSeeds}(\mathsf{sk}, C + 1)$
2 : $\phi_0 \leftarrow \mathsf{expATF}(\delta_0)$
3 : $\beta \leftarrow_R \{0,1\}^\lambda, \alpha \leftarrow 0$
4 : $\{\mathsf{s}_i\}_{[r]} \leftarrow \mathsf{expSeeds}(\beta, r)$
5 : $\{B_i^{col}\}_{[r]} \leftarrow \mathsf{expCols}(\{\mathsf{s}_i\}_{[r]})$
6 : $\{\psi_i\}_{[r]} \leftarrow \mathsf{ActATF}(\{\phi_0, B_i^{col}\}_{[r]})$
7 : $\mathsf{Ch}', \mathsf{F} \leftarrow \mathsf{H}_p(\mathsf{H}(M)||\{\psi_i'\}_{[r]})$
8 : **while** not F and $\alpha < \mathsf{L}$
9 : $\{\psi_i\}_{[r]} \leftarrow \mathsf{permATF}(\{\psi_i\}_{[r]}, \alpha)$
10 : $\alpha \leftarrow \alpha + 1$
11 : $\mathsf{Ch}', \mathsf{F} \leftarrow \mathsf{H}_p(\mathsf{H}(M)||\{\psi_i'\}_{[r]})$
12 : **if** not F **then** go to line 3
13 : $\{c_i\}_{[r]} \leftarrow \mathsf{expCha}(\mathsf{Ch})$
14 : $\mathsf{S} \leftarrow \{\},\ \mathsf{I}_\Delta \leftarrow \{\}$
15 : **for** $i \in [r]$ **do**
16 : **if** $c_i = 0$ **then** $\mathsf{App}(\mathsf{S}, \mathsf{s}_i)$
17 : **else** $\mathsf{App}(\mathsf{I}_\Delta, i)$
18 : $\{\Delta_{c_i}^{col}\}_{\mathsf{I}_\Delta} \leftarrow \mathsf{expCols}(\{\delta_{c_i}\}_{\mathsf{I}_\Delta})$
19 : $\{D_i\}_{[K]} \leftarrow \mathsf{ColMul}(\{\Delta_{c_i}^{col}, B_i^{col}\}_{\mathsf{I}_\Delta})$
20 : $\{D_i^{col}\}_{[K]} \leftarrow \mathsf{ColDec}(\{D_i\}_{[K]})$
21 : **if** ColDec failed **then** go to line 3
22 : **return** $\mathsf{Sig} = (\mathsf{Ch}, \mathsf{S}, \{D_i^{col}\}_{[K]})$

Fig. 2: ALTEQ injected with success probability p, sample limit L

4 Applying p, L to ALTEQ: parameters trade-off

4.1 ALTEQ parameters, and vectorization

The current ALTEQ parameters have not changed since their original submission. We present the parameters below in table 1[7].

For vectorized implementations using the interweaving of the ATFs, such as what the C code of ALTEQ suggests doing with strongly hinting GCC to use vectorization, it is essential for maximal efficiency that both r and K parameters are a multiple of the vector size. C is not too important, as it needs to be kept non-interweaved in their implementation. One can see in table 1, that

[7] Note that the ALTEQ team recently retracted their level V parameters after noticing some miscalculations, but this does not affect the ideas in this paper.

parameter set	r	K	C	security level of ALTEQ (bit)
I	84	22	7	128.1
	16	14	458	130.6
III	201	28	7	192.0
	39	20	229	192.7
V	119	48	8	256.0
	67	25	227	256.2

Table 1: (r, K, C) per security level in the original ALTEQ

most of the times they are not too far off. In particular, $r = 16$ does not need to be changed: it is already the size of two registers for AVX2 or one for AVX512.

Currently, we can see that r is not divisible by 8 in most parameters: this in practice leads to useless computations, as the last vector will contain useless values. In particular, it was reported in [18] that ALTEQ's updated code has slower setup performance for $C < R = 8$ in AVX2, which is not completely unexpected since their vector-friendly code assume to compute per packs of R elements as you can put R elements per vector. Every part of the cryptosystem is affected. In this paper we prefer to focus on verification performance, so it is more important to set r to a multiple of a register size: in particular, when operations are done *per vectors*, the number of operations should be considered by the value $\lceil r/R \rceil$ rather than r: for example, $r = 33$ and $r = 40$ **have the same number** $\lceil r/R \rceil = 5$ of AVX2 vector operations ($R = 8$).

4.2 Decreasing r to AVX2 friendly values, without touching (K, C)

In table 2, we attempt to decrease r to $r-(r \mod R)$ for AVX2 where $R = 8$, i.e we do *exactly one less vector operation* and slightly decrease the signature size. For some parameters, the success rate is not too low, which means we can still obtain signatures somehow quickly. However some parameters have a chance of success lower than 3%, and one has less than 1%. To obtain a 99% probability of *at least one success out of* L *samples*, we need $\mathsf{L} = 812$ for the "large public key" parameters of 192-bits security, and $\mathsf{L} = 189$ for the "fully equilibrated" parameters of 256-bits which can be expensive whenever permutations are slow to generate (when using AES-CTR to feed the Fisher-Yates algorithm, for example). Even if the permutations were not slow to generate using deterministic iterative algorithms such as Heap's which does only one swap per permutation, the fact that we might compute L hashes can be crippling for signing performance: the point in this part is to accelerate the verification.

4.3 Decreasing the signature size through better r/K trade-offs

For some applications, getting a slightly slower verification is preferable if we can get smaller signatures in return. To decrease the signature size significantly,

security lvl	new r	K	C	r modif	$\approx$ of p	L for 99%	$\lceil r/R \rceil$ modif
I	80	22	7	-4	0.311	13	-9%
	16	14	458	0	1	1	0%
III	200	28	7	-1	0.888	3	-3%
	32	20	229	-7	0.005	812	-20%
V	112	48	8	-7	0.024	189	-6%
	64	25	227	-3	0.275	15	-11%

Table 2: Lowering r to an AVX2-friendly value with p, no changes to (K, C)

the main parameter to target for signature size reduction is not r but K. Each of the K matrix have $n^2 \log_2(q)$-bits, while each of the $r - K$ seeds have λ-bits. Those sizes are entirely dependent of (r, K, C), which allows us to make "easier trade-offs". In particular for ALTEQ parameters, we have:

- for $\lambda = 128$, we have $n = 13$ and $42\lambda < 32 \times n^2 < 43\lambda$
- for $\lambda = 192$, we have $n = 20$ and $66\lambda < 32 \times n^2 < 67\lambda$
- for $\lambda = 256$, we have $n = 25$ and $78\lambda < 32 \times n^2 < 79\lambda$

which means, for example, that a decrease by one in K is approximately equivalent to a decrease by 42 or 43 to $r - K$ in the signature size, thus increasing r by 40 but reducing K by 1 could be (marginally) worth it. However, note that balancing is a hard task since the impact of K is much greater than r on the security and C plays a central role in the way K impacts the signature's security: in particular, for low values of K such as in the "large public key" parameters (i.e large C), decreasing K must be compensated by large increases of r. This leads to an increase of the verification time (which is mostly r dependent) but we are merely exploring the option in this part.

However, with the introduction of our success probability parameter p, we can limit the impact on how much we would have to increase r to compensate the decrease of K, *without changing* C. We showcase some examples in table 3, while aiming for $(r \mod 8 = 0)$ for AVX2 to keep a simpler verification implementation (that could still be slower since r is increased), and to easily count the number of supplementary vector operations. We can see that some parameters in table 3 are probably unlikely to be practical anywhere (except maybe with some other changes, but not as it is), but the rest of the parameters seem affordable: keep in mind that we are doing permutations to resample, we are not *exactly* multiplying by 3391 the signing time, especially since the performance bottleneck is mainly ActATF, with functions ColDec and to a lesser extent ColMul having non-negligible computational time.

4.4 Decreasing the public key size C without touching (r, K)

In the case of the "large public key" parameters, the parameters were crafted to have short signatures at the cost of a large public key. Short signatures are

security lvl	new r	new K	C	r/K modif	$\approx$ of p	L for 99%	Sig modif	$\lceil r/R \rceil$ modif
I	112	18	7	+28/-4	0.001	3391	-13%	+27%
	24	12	458	+8/-2	0.676	5	-12%	+50%
III	256	24	7	+55/-4	0.0001	45376	-10%	+23%
	104	16	229	+61/-4	0.024	189	-14%	+160%
V	200	38	8	+71/-10	0.02	208	-18%	+66%
	72	24	227	+5/-1	0.024	190	-3%	0%

Table 3: Examples of $+r/\text{-}K$ using p (no C changes), aiming AVX2-friendly r

usually good to reduce communication sizes, but many signatures must be produced to counteract the fact that an initial setup with a large key must be first managed. In the last subsection we did not touch C and modified r/K instead. This time, we aim to see if we can reduce just C, which should reduce the communication threshold of when it starts to be beneficial to use those "large public key" parameters rather than the "fully equilibrated" ones. This part obviously applies mostly to the "large public key" parameters, as there is not much to reduce when $C = 7, 8$: for the sake of the experiment we nevertheless present some examples applied to the "fully equilibrated" parameters. Those examples can be found in table 4.

security lvl	r	K	new C	$\approx$ of p	L for 99%	pk modif
I	84	22	6	0.03	125	-14%
	16	14	300	0.016	271	-34%
III	201	28	6	0.01	333	-14%
	39	20	180	0.013	327	-21%
V	119	48	7	0.001	2750	-12%
	67	25	190	0.013	341	-16%

Table 4: Reducing C using p, no change to (r, K)

5 Conclusion

We extended the GMW-FS framework with unbalanced challenges with an extra probability parameter p. Thanks to this success probability, we are a bit more flexible in parameter setting, which allows to obtain some more convenient parameters especially for AVX implementations, and accelerate the verification speed. While this does hinder the signature speed, the NIST announcement focused on signature size and verification speed for which we both improve. We show a practical example on optimizing some ALTEQ parameters, and showcase other uses of this parameter to obtain lower key sizes, or lower signature sizes with more flexibility. Since the idea does not depend on the ATFE problem it-

self, it could be used for any group-based cryptography primitive: maybe the primitives from LESS [1] and MEDS [7] could also benefit from this.

Future works While it is clear that verification will be accelerated just from reducing the computational cost from r alone even in non-vectorized implementations, the unique drawback here is the heavier signature process. We first need to properly choose and implement how we are going to perform the successive permute-then-hash-again failure management. Is there a specific secure method that let us keep some of the data? After all, we are hashing a permutation of a previous entry: maybe there is a more practical way to securely generate the challenges and the rejection, that does not force us to restart the whole challenge generation and validation process at every rejection. This could imply a deeper rework of the GMW-FS protocol, or an ALTEQ overhaul. Obviously, the gains can theoretically be at most polynomial, since the hardness of finding correct permutations are a security guarantee against forgers, but the examples of p, L values we showed in this paper do not reach insurmountable levels either. Furthermore, we only modified parts of the parameters, but did not attempt to modify all values (r, K, C) simultaneously: maybe some interesting quadruplets (p, r, K, C) are left to be found. Once this is done, a natural following work would be to transform the current generic code of ALTEQ that works for all parameters, to a more efficient code of ALTEQ that works for all AVX2-optimized parameters: we should obtain faster performance even when r is unchanged (such as $r = 16$).

Acknowledgments. All members of the ALTEQ team for various comments and discussions, as well as Kazuhide Fukushima for help in various procedures. Despite never interacting with them, Beullens and Saarinen's comments on [pqc-forum] inspired this work: hence we present their comments as the same level as an academic paper.

Disclosure of Interests. Naturally, all members of the ALTEQ team.

References

1. Baldi, M., Barenghi, A., Beckwith, L., Biasse, J.F., Esser, A., Gaj, K., Mohajerani, K., Pelosi, G., Persichetti, E., Saarinen, M.J., Santini, P., Wallace, R.: Less (linear equivalence signature scheme) (2023), `https://www.less-project.com/`
2. Barker, E., Kelsey, J.: Recommendation for random number generation using deterministic random bit generators. NIST Special Publication **800**, 90A (2015)
3. Beullens, W.: Battle report (first 30 hours, add'l sigs round 1) (2023), `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/5JMFqozi1Bc/m/qnWnsAtxBQAJ`
4. Beullens, W.: Trivial multi-key attacks + attack on alteq (2024), `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/tjhrrmv837w/m/sjxHooYgBAAJ`
5. Beullens, W.: Graph-theoretic algorithms for the alternating trilinear form equivalence problem. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023. pp. 101–126. Springer Nature Switzerland, Cham (2023)

6. Bläser, M., Duong, D.H., Narayanan, A.K., Plantard, T., Qiao, Y., Sipasseuth, A., Tang, G.: The alteq signature scheme: Algorithm specifications and supporting documentation. NIST PQC Submission (2023), https://pqcalteq.github.io/ALTEQ_spec_2023.09.18.pdf
7. Chou, T., Niederhagen, R., Persichetti, E., Ran, L., Randrianarisoa, T.H., Reijnders, K., Samardjiska, S., Trimoska, M.: Meds: Matrix equivalence digital signature (2023), https://www.meds-pqc.org/
8. Computational Algebra Group, U.o.S.: Magma online (2018), https://magma.maths.usyd.edu.au/calc/, https://magma.maths.usyd.edu.au/calc/
9. Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-dilithium algorithm specifications and supporting documentation. Round-2 submission to the NIST PQC project **35** (2021)
10. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)
11. Fisher, R.A., Yates, F.: Statistical tables for biological, agricultural and medical research. Hafner Publishing Company (1953)
12. Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. Journal of the ACM (JACM) **38**(3), 690–728 (1991)
13. Guido Bertoni and Joan Daemen and Seth Hoffert and Michaël Peeters and Gilles Van Assche and and Ronny Van Keer: eXtended Keccak code package, https://github.com/XKCP/XKCP
14. Heap, B.: Permutations by interchanges. The Computer Journal **6**(3), 293–298 (1963)
15. NIST: Post-Quantum Cryptography Standardization (2022), https://csrc.nist.gov/news/2022/pqc-candidates-to-be-standardized-and-round-4
16. NIST: Post-Quantum Cryptography Standardization (2023), https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures
17. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**(2) (1978)
18. Saarinen, M.J.O.: Official comment: Alteq (2023), https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/-LCPCJCyLlc/
19. Sedgewick, R.: Permutation generation methods. ACM Computing Surveys (CSUR) **9**(2), 137–164 (1977)
20. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing **26**(5), 1484–1509 (1997)
21. Tang, G., Duong, D.H., Joux, A., Plantard, T., Qiao, Y., Susilo, W.: Practical post-quantum signature schemes from isomorphism problems of trilinear forms. In: Dunkelman, O., Dziembowski, S. (eds.) Advances in Cryptology – EUROCRYPT 2022. pp. 582–612. Springer International Publishing, Cham (2022)
22. Trotter, H.F.: Algorithm 115: Perm. Commun. ACM **5**(8), 434–435 (aug 1962). https://doi.org/10.1145/368637.368660, https://doi.org/10.1145/368637.368660

MAGMA code

This is the MAGMA code to input into MAGMA online [8] to obtain the data presented in all tables of this paper. The code has been made easily modifiable for

anybody to "play" with any parameter set and manually search for "acceptable" compromises.

```
ListProbaSuccessPerTry:=function(pb_suc_one, aimed_prob)
  pb_fail := 1.0 - pb_suc_one;
  nb_tries := 1;
  while (1-pb_fail) lt aimed_prob do
    pb_fail *:= 1.0 - pb_suc_one;
    nb_tries +:=1;
  end while;
  return nb_tries, 1-pb_fail;
end function;

/* Those are the original ALTEQ parameters */
//lbd:=128; n:=13; r:=84; K:=22; C:=7;
//lbd:=128; n:=13; r:=16; K:=14; C:=458;
//lbd:=192; n:=20; r:=201; K:=28; C:=7;
//lbd:=192; n:=20; r:=39; K:=20; C:=229;
//lbd:=256; n:=25; r:=119; K:=48; C:=8;
lbd:=256; n:=25; r:=67; K:=25; C:=227;

print "original:\n","n =",n,"| r =",r,"| K =",K,"| C =",C,"| lbd =",lbd;

Size1:= (r-K+2)*lbd + K*(n^2)*32;
print "original security:",Floor(Log(2,Binomial(r,K) *
    (C^K))),"signature size:",Size1;

/* With "p" made to match lambda security, make your own modifs */
new_r:=r;
new_r:=Floor(new_r/8);new_r*:=8; /* Modif for AVX2 friendly r */
K:=24;
NewC:=C;
print "param try:\n","n =",n,"| r =",new_r,"| K =",K,"| C =",NewC;

p:=((Binomial(new_r,K) * (NewC^K)) / (2^lbd));
"success rate on one combination:",0.0 + p;

Size2:=(new_r-K+2)*lbd + K*(n^2)*32;
print "new security:",Floor(Log(2,(p^-1)*Binomial(r,K)*(C^K))),"new
    signature size:",Size2;

AimedProb:=0.99;
SamplesForGoal:=ListProbaSuccessPerTry(p, AimedProb);
print "samples to get",AimedProb,"success rate:",SamplesForGoal;

print "Signature Size Gain:",(Size1 - Size2)/Size1 + 0.0;
print "Public Key Gain (without seed):",(C - NewC)/C + 0.0;
print "AVX2 chg:", (Ceiling(r/8) - Ceiling(new_r/8))/Ceiling(r/8) + 0.0;
```