Rainbow on Cortex-M4

Tung Chou[1], Matthias J. Kannwischer[2] and Bo-Yin Yang[1,3]

[1] Research Center for Information Technology and Innovation, Academia Sinica, Taipei, Taiwan
{blueprint,by}@crypto.tw
[2] Max Planck Institute for Security and Privacy, Bochum, Germany
matthias@kannwischer.eu
[3] Institute of Information Science, Academia Sinica, Taipei, Taiwan

Abstract. We present the first Cortex-M4 implementation of the NISTPQC signature finalist Rainbow. We target the Giant Gecko `EFM32GG11B` which comes with 512 kB of RAM which can easily accommodate the keys of RainbowI.
We present fast constant-time bitsliced $\mathbb{F}_{16}$ multiplication allowing multiplication of 32 field elements in 32 clock cycles. Additionally, we introduce a new way of computing the public map $\mathcal{P}$ in the verification procedure allowing vastly faster signature verification.
Both the signing and verification procedures of our implementation are by far the fastest among the NISTPQC signature finalists. Signing of `rainbowIclassic` requires roughly 957 000 clock cycles which 4× faster than the state of the art Dilithium2 implementation and 45× faster than Falcon-512. Verification needs about 239 000 cycles which is 5× and 2× faster respectively. The cost of signing can be further decreased by 20% when storing the secret key in a bitsliced representation.

Keywords: Rainbow, NISTPQC, Cortex-M4, MQ signatures, finite field arithmetic

1 Introduction

The advance of large scale quantum computers is threatening all conventional public-key cryptography currently deployed due to Shor's algorithm [Sho94]. Hence, researchers are looking into quantum-safe replacements for existing protocols. In 2016, the American National Institute of Standards and Technology (NIST) [NIS] called for proposals to replace their existing standards for digital signatures, public-key encryption (PKE), and key-encapsulation mechanisms (KEM). In 2020, the third and final round of the standardization process (NISTPQC) with 7 remaining finalists and 8 alternate candidates started. Out of these remaining schemes, 6 are digital signature schemes (3 finalists and 3 alternate candidates). They can be grouped into three major families; each of which has its own advantages and disadvantages: Hash-based signatures (SPHINCS+ [ABB+20]) have small keys, but large signatures; Lattice-based signatures (Dilithium [BLD+20] and Falcon [FHK+20]) have medium keys and medium signatures; Multivariate Quadratic (MQ-)based signatures (GeMSS [CFM+20] and Rainbow [DCK+20a]) have very small signatures, but large keys. The sixth signature scheme is Picnic built on top of zero knowledge proofs which does not quite fit any of those families.

Rainbow has a reputation extremely for fast verification (and signing), and comes with very small signatures. However, while implementations of both hash-based signatures and lattice-based signatures have received broad attention from the community, there appears to be only very little work on implementations of MQ-based schemes, even though the aforementioned characteristics of Rainbow make it particularly suitable either for root certificates, for any cases where the key can be built into the application, or in any situation not calling for frequent downloading or updating.

Especially, implementing multivariate schemes and in particular UOV-based schemes on embedded platforms seem to be poorly explored which may be due to the large public keys of these schemes. It may also be due to the implementation complexity of the schemes as effectively and securely implementing the Gaussian elimination and field arithmetic on an embedded platform is difficult.

So we try to bridge that gap in this work and present Cortex-M4 implementations of the smallest Rainbow instances `rainbowIclassic`, `rainbowIcircumzenithal`, and `rainbowIcompressed`. As the public keys for those parameter sets are 157.8 kB, 58 kB, and 58 kB respectively these are still within reach of fitting onto some embedded platforms. The larger Rainbow parameter sets (level 3 and level 5) have public keys of 258 kB up to 1885 kB and, thus, may be arguably unsuitable for embedded platforms. However, our methods are not simply tailored only for $\mathbb{F}_{16}$ and `rainbowI`. If there are larger Cortex-M's, part of our methods will extend straightforwardly and be useful for `rainbowIII` and `rainbowV` which use $\mathbb{F}_{256}$.

Contribution. We present optimized Cortex-M4 implementations of all three third-round instances of Rainbow with the parameter set aiming at NIST security level 1. Our achieved speed-ups mostly come from bitsliced multiplication of $\mathbb{F}_{16}$ elements making use of the conditional execution instructions available on the Cortex-M4. This multiplication can be even faster by switching to a direct $\mathbb{F}_{16}$ representation ($\mathbb{F}_{16} = \mathbb{F}_2[x]/(x^4 + x + 1)$) rather than the tower field representation mandated by the Rainbow specification. We argue that the specification should be changed. Furthermore, we choose an approach [CCNY12] which was not previously used in the literature describing how to evaluate MQ public maps [CLP$^+$18, DCK$^+$20a, BPSV19]. In Rainbow, this is vastly faster. As verification for `rainbowIclassic` only consists of hashing and evaluating the public map, our approach results in blazingly fast verification.

As our target platform (the `EFM32GG11B`) comes with a `SHA256` and `AES` hardware accelerator, we also present how Rainbow implementations can benefit from faster symmetric operations.

Code. Our Open-Source Cortex-M4 implementation of Rainbow is available at `https://github.com/rainbowm4/rainbowm4`.

Related Work. Rainbow and its near relative TTS [YC05] was implemented in a sequence of papers [CCC$^+$08, CCC$^+$09, CLP$^+$18] for Intel platforms, but they tend to be for prior parameters and use different Intel-specific optimizations different from what we use. [CLP$^+$18] mostly uses the VPSHUFB table look-up instruction in AVX2 instruction set for $\mathbb{F}_{16}$ and is inapplicable to the ARM Cortex-M4, and [CCC$^+$09] uses Wiedemann to evaluate matrix inverses which is not constant time.

There is a large body of work targeting the Cortex-M4 on the other NISTPQC finalists Dilithium [GKS20], Falcon [Por19], Kyber [ABCG20], Saber [KRS19, BMKV20, CHK$^+$21], and NTRU [KRS19, CHK$^+$21]. Unfortunately, these lattice-based finalists use modular integer arithmetic and polynomial multiplication for relatively many coefficients, which means their optimizations do not carry over to Rainbow.

There are very few implementations on the Cortex-M4 for MQ. In particular, the work covering Rainbow is very limited, especially with the most recent parameter sets. The only M4 work on Rainbow to our knowledge was from Moya Riera's Bachelor thesis [MR19], which optimized level 1 parameter sets of the second round Rainbow. However, this implementation used look-up tables for $\mathbb{F}_{16}$ arithmetic which is not constant-time on all Cortex-M4 platforms, and additionally (despite the smaller parameters) was considerably slower than our work.

There is large body of work targeting the Cortex-M4 on the other NISTPQC finalists Dilithium [GKS20], Falcon [Por19], Kyber [ABCG20], Saber [KRS19, BMKV20, CHK+21], and NTRU [KRS19, CHK+21]. Unfortunately, these lattice-based finalists use modular integer arithmetic and polynomial multiplication for relatively many coefficients, which means their optimizations do not carry over to Rainbow.

Applicability to other contexts. Our techniques for evaluating quadratic systems (Section 3.3) and solving linear systems (Section 3.2) equally benefit Unbalanced Oil and Vinegar [KPG99] and all its derivatives. Our constant-time matrix solving (Section 3.2) generalize the Gauss-Jordan Elimination used in key generation of code-based cryptography, e.g., as briefly mentioned in McBits [Cho17].

Structure. In Section 2 we introduce Rainbow and the features of the Cortex-M4 that proved useful when implementing Rainbow. Section 3.1 introduces fast bitsliced $\mathbb{F}_{16}$ arithmetic for the Cortex-M4 which is used in the core operations within Rainbow. Section 3.2 shows how the matrix inversion in the signing operation can be implemented efficiently and in constant time. Section 3.3 describes how the public map can be evaluated in variable time and, consequently, how verification can be sped up. Section 4 shows how a different $\mathbb{F}_{16}$ representation can speed up Rainbow even further, although it would require a change of the specification. In Section 5, we present the performance of the resulting implementations and compare it to other NISTPQC finalists.

2 Preliminaries

We introduce the Rainbow signature in Section 2.1 and describe useful features of the Cortex-M4 for Rainbow in Section 2.3.

2.1 Recap of Multivariate Signatures

A Multivariate Quadratic Public Key Cryptosystem works on a field $\mathbb{K} = \mathbb{F}_q$ which is called the "base field". For Rainbow I this is $\mathbb{F}_{16}$. It has a public map $\mathcal{P} = T \circ \mathcal{Q} \circ S : \mathbb{K}^n \to \mathbb{K}^m$ where T and S are typically affine but is here (for Rainbow) linear. So, $S : \mathbf{w} \mapsto \mathbf{x} = \mathrm{M}_S\mathbf{w}$ and $T : \mathbf{y} \mapsto \mathbf{z} = \mathrm{M}_T\mathbf{y}$. The map $\mathcal{Q} : \mathbf{x} \mapsto \mathbf{y}$, called the *central map* must be quadratic and be easily invertible. The various MPKCs are characterized by the construction of their $\mathcal{Q}$'s, obviously it must be hard to decompose $\mathcal{P} : \mathbf{w} \mapsto \mathbf{z}$ into its component maps. Usually $n > m$ and we have a digital signature.

2.2 Summary of Rainbow

Rainbow was proposed by Ding and Schmidt in 2004 [DS05], with a multi-stage Unbalanced Oil and Vinegar (UOV) structure. Since 2008 it has always appeared with exactly two stages [DYC+08], and this is what we describe below.

2.2.1 The Central Map in Rainbow

Modern variants of Rainbow($\mathbb{F}_q, v_1, o_1, o_2$) is parametrized by four integers q, v_1, o_1, o_2 [DS05, DYC+08].

- There are two "segments" of central maps in each which we designate "oil" and "vinegar" variables. In the first segment the *vinegar* variables are the x_i for $i \in V_1 = \{1, \ldots, v_1\}$ and the *oil* variables are the x_i for $i \in O_1 = \{v_1+1, \cdots, v_2 := v_1+o_1\}$. In the second segment, the *vinegar* variables has the index set $V_2 = \{1, \cdots, v_2 := v_1+o_1\}$ and the *oil* variables the index set $O_2 = \{v_2+1, \cdots, n = v_3 = v_2+o_2 = v_1+o_1+o_2\}$.

- The central map $\mathcal{Q}$ has $m = o_1 + o_2$ structured quadratic equations $\mathbf{y} = (y_{v_1+1}, \ldots, y_n) = (q_{v_1+1}(\mathbf{x}), \ldots, q_n(\mathbf{x}))$, where (notice the unusual indexing):

$$\begin{aligned} y_k = q_k(\mathbf{x}) &= \sum_{i=1}^{v_1} \sum_{j=i}^{v_2} \alpha_{ij}^{(k)} x_i x_j, \text{for } k \in O_1; \\ y_k = q_k(\mathbf{x}) &= \sum_{i=1}^{v_2} \sum_{j=i}^{n} \alpha_{ij}^{(k)} x_i x_j, \text{for } k \in O_2. \end{aligned}$$

- *Note that in every q_k, where $k \in O_1$, there is no cross-term $x_i x_j$ where both i and j are in O_1.* So given all the y_i in the first stage with $v_1 < i \leq v_2$, and all the vinegar variables x_j with $j \leq v_1$, we can easily compute the corresponding oil variables $x_{v_1+1}, \ldots, x_{v_2}$ by solving a linear system.

  Similarly, *in every q_k, where $k \in O_2$, there is no cross-term $x_i x_j$ where both i and j are in O_2.* So given all the y_i in the second stage with $v_2 < i \leq n$, and all the vinegar variables x_j with $j \leq v_2$, we can easily compute $x_{v_2+1}, \ldots, x_{v_n}$ by solving a linear system.

- An inverse image $\mathbf{x}$ of $\mathcal{Q}$ such that $\mathcal{Q}(\mathbf{x}) = \mathbf{y}$, can be found as follows:

  1. Randomly guess the initial vinegar variables $\bar{\mathbf{x}} = (x_1, \ldots x_{v_1})$ and from that and $(y_{v_1+1}, \ldots, y_{v_2})$ solve for $(x_{v_1+1}, \ldots, x_{v_2})$ via Gauss-Jordan elimination. If there is no solution, restart from the beginning.
  2. Having now the values $\bar{\mathbf{x}} = (x_1, \ldots x_{v_2})$, from that and $(y_{v_2+1}, \ldots, y_n)$ again solve for $(x_{v_2+1}, \ldots, x_n)$ via Gauss-Jordan elimination. If there is no solution, restart from the beginning.

The procedure is obviously extensible to any number of stages. A toy example of the central map $\mathcal{Q}$ in Rainbow can be found in Appendix A.

2.2.2 Procedures of MPKC Signatures including Rainbow

An MPKC signature system comprises three main procedures: key generation, signing messages, and verifying signatures. Signing and verification are much more important because a signature key is not expected to change often.

Key generation The user randomly chooses a secret key which consists of invertible S, T, and $\mathcal{Q}$, then computes $\mathcal{P} = T \circ \mathcal{Q} \circ S$ as the public key. S^{-1}, T^{-1} and the parameters in $\mathcal{Q}$ is kept as the private key. We mostly follow [DCK+20a, Sec. 4] and its reference implementation (aside from doing multiplications more efficiently), as it is a faster approach for key generation of Rainbow compared to the alternative, which is MQ key polynomial interpolation of $T \circ \mathcal{Q} \circ S$ [Wol04].

Signing The signer first computes the hash value of the message as the digest $\mathbf{z} \in \mathbb{K}^m$. With the secret key, the signer computes $\mathbf{y} = T^{-1}(\mathbf{z})$, $\mathbf{x} = \mathcal{Q}^{-1}(\mathbf{y})$, and $\mathbf{w} = S^{-1}(\mathbf{x}) \in \mathbb{K}^n$ which is the signature of the message. This is common to all multivariate signatures although the details of computing $\mathcal{Q}^{-1}(\mathbf{y})$ vary with specific schemes.

Verification To verify a signature $\mathbf{w} \in \mathbb{K}^n$ of a message, the user evaluates the public polynomial $\mathcal{P}(\mathbf{w}) = \mathbf{z}$ and checks whether the digest of the message is equal to $\mathbf{z}$.

Table 1: Parameters of Rainbow [DCK+20a].

security	NIST Round	Round 2	Round 3
128 bits	Field	$\mathbb{F}_{16}$	$\mathbb{F}_{16}$
	(v_1, o_1, o_2)	32, 32, 32	36, 32, 32
	$n \to m$	96 → 64	100 → 64
192 bits	Field	$\mathbb{F}_{256}$	$\mathbb{F}_{256}$
	(v_1, o_1, o_2)	68, 36, 36	68, 32, 48
	$n \to m$	140 → 72	148 → 80
256 bits	Field	$\mathbb{F}_{256}$	$\mathbb{F}_{256}$
	(v_1, o_1, o_2)	92, 48, 48	96, 36, 64
	$n \to m$	188 → 96	196 → 100

2.2.3 Parameters of Rainbow

Ding et al. [DCK+20a] chose the parameters for security requirements in Table 1. Previously, against a Rainbow cryptosystem with m equations and n variables, the most pertinent attacks are substituting $n - m$ variables at random and trying to solve for the remaining m variables ("Direct Attack"), and a structural attack which involves solving an associated quadratic system with n variables and $n + m - 1$ equations ("Rainbow Band Separation") [DYC+08]. Recently, Beullens posted the new "Intersection" and "Rectangular MinRank" attacks against Rainbow [Beu20]. The Rainbow team acknowledged these attacks, emphasizing that Round-3 Rainbow still meets its planned security levels [DCK+20b].

2.2.4 Computational Costs of Rainbow Signing

The signer, as above, calculates the hash digest $\mathbf{z}$ of message and inverts $\mathcal{P}$ with the secret key T, S, and $\mathcal{Q}$, and does

$$\mathbf{z} \in \mathbb{K}^m \overset{T^{-1}}{\longmapsto} \mathbf{y} \overset{\mathcal{Q}^{-1}}{\longmapsto} \mathbf{x} \overset{S^{-1}}{\longmapsto} \mathbf{w} \in \mathbb{K}^n \quad ,$$

where $\mathbf{w}$ is the signature. Inverting the central map $\mathcal{Q}$ is clearly slower than inverting S, T. While inverting $\mathcal{Q}$ with given $\mathbf{y}$, the signer randomly guesses vinegar variables $\bar{\mathbf{x}} = (x_1, \ldots x_{v_1})$ and solve $(x_{v_1+1}, \ldots, x_{v_2})$ by

$$\begin{aligned} y_{v_1+1} &= \bar{\alpha}^{(v_1+1)}_{v_1+1} x_{v_1+1} + \cdots + \bar{\alpha}^{(v_1+1)}_{v_2} x_{v_2} + \bar{\beta}^{(v_1+1)}_{V_1} \\ &\vdots \\ y_{v_1+o_1} &= \bar{\alpha}^{(v_2)}_{v_1+1} x_{v_1+1} + \cdots + \bar{\alpha}^{(v_2)}_{v_2} x_{v_2} + \bar{\beta}^{(v_2)}_{V_1} \quad . \end{aligned} \tag{1}$$

Here $(\bar{\beta}^{(v_1+1)}_{V_1}, \ldots, \bar{\beta}^{(v_2)}_{V_1})$ is evaluated as quadratic forms in $\bar{\mathbf{x}}$. This is obtained from evaluation of secret-quadratic equations with secret values $\bar{\mathbf{x}}$ and the matrix

$$\begin{bmatrix} \bar{\alpha}^{(k)}_{i} & \cdots & \bar{\alpha}^{(k)}_{i'} \\ & \ddots & \\ \bar{\alpha}^{(k')}_{i} & & \bar{\alpha}^{(k')}_{i'} \end{bmatrix}, \text{ where } i, i' \text{ and } k, k' \in O_1 \quad ,$$

which we call $\texttt{matVO}(\bar{\mathbf{x}})$. If $\texttt{matVO}(\bar{\mathbf{x}})$ is a singular matrix the initial guesses are discarded and the process is restarted. The signer will repeat this procedure to solve for x_i with $i \in O_2$, that is the variables $x_{v_2+1}, \ldots, x_n$ as we have now values of x_i for $i \in V_1 \cup O_1 = V_2$, using also the values $y_{v_2+1}, \ldots, y_n$.

Clearly, the main computation cost of signing is solving linear equations and computing the matrices matVO($\bar{\mathtt{x}}$) *from vinegar variables* $\bar{\mathtt{x}}$*, twice.*

Note that randomness (vinegar variables and salt) are generated using AES counter mode according to the spec, with every byte sampled providing two random $\mathbb{F}_{16}$ elements.

2.2.5 Variations on the Basic Rainbow

In NIST round 2 and 3, Rainbow's authors included *circumzenithal* and *compressed* variants, expanding most of the public key using AES counter mode from a seed, and storing only parts of the keys not producible in this way. The private key can be derived from the private matrices S and T and this public key and stored separately. This method, first appearing in [PBB10], reverses the normal procedure of deriving the public key from the private key during key generation. Note that a *circumzenithal* arc or rainbow is a meteorological phenomenon resembling an inverted rainbow. In the *compressed* variant, the entire private key is additionally generated on the fly from the public key seed and the private seed for S and T. These variations obviously trade key sizes for the time recomputing keys.

2.3 Cortex-M4

The Cortex-M4 is NIST's primary microcontroller optimization target for the post-quantum competition. The Cortex-M4 is a 32-bit processor that implements the `ARMv7E-M` instruction set which comes with a number of powerful instructions. For example, the DSP instructions [KRS19, BMKV20, BFM+18] as well as the single-cycle long multiplication instructions [GKS20, CHK+21, SJA19] proved to be very beneficial for implementing post-quantum cryptography.

However, for implementing Rainbow, we mostly rely on instructions that are also present in the `ARMv7-M` instruction set (a subset of `ARMv7E-M`) which is, for example, implemented by the Cortex-M3 microarchitecture. However, Cortex-M3 cores usually come with considerably less RAM which makes them arguably less suitable for Rainbow implementations.

The following features of `ARMv7-M` are particularly useful for implementing Rainbow:

Conditional execution. The feature benefiting Rainbow the most, is conditional execution. Using the `it` instruction one can execute up to four instructions conditionally on a flag value. For example,

```
ite EQ
addeq r0, r1
addne r0, r2
```

either adds `r1` or `r2` to `r0` depending on the `Z` flag (equal) being set or not.

Note that the ARMv7-M manual [ARM18, Section A4.1.2] states that

> ”If the flags do not satisfy this condition, the instruction acts as a NOP, that is, execution advances to the next instruction as normal, including any relevant checks for exceptions being taken, but has no other effect“

Hence, it is safe to use single-cycle instructions with secret-dependent conditions in constant-time code as the run-time will be one cycle irrespective of the condition flags. In future ARM architectures it needs to be carefully evaluated if this is still the case. An `it` block can consist of up to four instructions of which the first must be the *then* branch and the following can be either *then* or *else*. The `it` instruction encodes which instructions of an `it` block belong to which branch, e.g., `itttt`, `ittee`, and `itete`. The conditions that can be used are the same as those for branch instructions (`eq`, `ne`, `cs`, `cc`, `mi`, `pl`, `vs`, `vc`, `hi`, `ls`, `ge`, `lt`, `gt`, `le`) and the flags can be set using arithmetic instructions (e.g., adds, subs) or explicit comparison instructions (e.g. cmp, tst). The conditions within an it block must be the same for all instructions (or the opposite for the *else* branch). it* instructions takes 1 cycle each on the M4 (unless it is the second of a pair of 2-byte instructions, which doesn't happen in our implementations).

Barrel shifting. Standard data-processing instructions (e.g., add, eor, and) allow to have a flexible second operand, i.e., the second argument can be shifted or rotated without changing the latency (1 cycle) for each instruction. For example,

```
add r0, r1, r2, LSL#2
```

will shift r2 left by two bit positions add it to r1 and store the result to r0. Similarly, other shifts and rotations can be used (lsr, asr, ror, rrx).

Special immediates. Standard data-processing instructions can also be used with a constant as a second operand. mov's are limited to 16 bits immediates 0x0000XYZW while immediates for other instructions are limited to an 8-bit value 0xXY shifted by some amount, or the special patterns 0x00XY00XY, 0xXY00XY00, and 0xXYXYXYXY.

3 Implementation Building Blocks

This section introduces the novel implementation approaches that can be used to speed up Rainbow implementations. Section 3.1 introduces fast bitsliced $\mathbb{F}_{16}$ multiplication which is useful throughout all aspects of Rainbow. We can speed up the multiplication further by switching to a direct $\mathbb{F}_{16}$ representation which is described in Section 4. Section 3.2 shows how we can adapt constant-time $\mathbb{F}_{16}$ matrix inversion to benefit from the fast bitsliced multiplication. This speeds up the signing procedure of Rainbow and can also be adapted for $\mathbb{F}_{256}$ parameter sets. Section 3.3 presents a novel approach for evaluating the public map $\mathcal{P}$ which is the core operation of Rainbow verification. We exploit that verification can run in variable time depending on both the public key and the signature. This also works for other parameter sets of Rainbow.

3.1 $\mathbb{F}_{16}$ multiplication

The core operation within Rainbow is arithmetic in a finite field. As mentioned before, for rainbowI parameter sets this field is $\mathbb{F}_{16}$ (for the higher levels it is $\mathbb{F}_{256}$). The $\mathbb{F}_{16}$ representation used within Rainbow is the tower field representation:

$$\mathbb{F}_{16} := \mathbb{F}_4[y]/(y^2 + y + x)$$

with

$$\mathbb{F}_4 := \mathbb{F}_2[x]/(x^2 + x + 1)$$

Hence, an element is represented by four bits e_i with $e = (e_3 \cdot x + e_2) \cdot y + e_1 \cdot x + e_0$. These bits are packed into a nibble with e_0 at the least significant bit position. Two elements are packed into a byte with the least significant nibble in the lower half of the byte.

One approach of multiplying two $\mathbb{F}_{16}$ elements is using Karatsuba multiplication [KO63] and is, for example, used in the reference implementation of Rainbow. It allows us to implement a $\mathbb{F}_{16}$ multiplication using three $\mathbb{F}_4$ multiplications.

Given $a = a_1 \cdot y + a_0$ and $b = b_1 \cdot y + b_0$ where $a_i, b_i \in \mathbb{F}_4$,

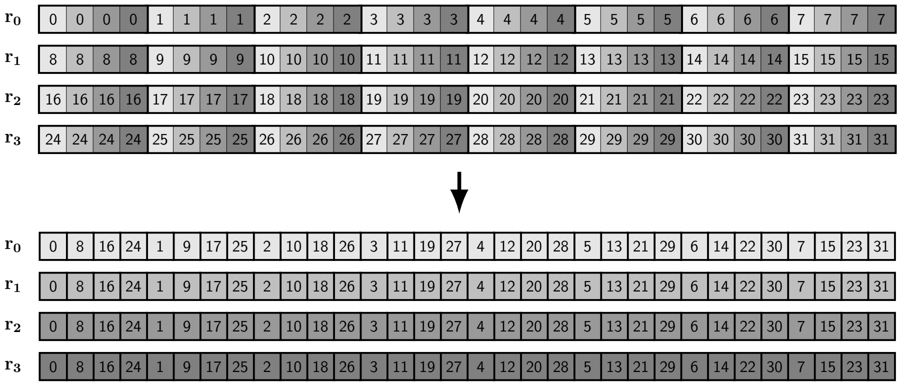
r0 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7
r1 8 8 8 8 9 9 9 9 10 10 10 10 11 11 11 11 12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 15
r2 16 16 16 16 17 17 17 17 18 18 18 18 19 19 19 19 20 20 20 20 21 21 21 21 22 22 22 22 23 23 23 23
r3 24 24 24 24 25 25 25 25 26 26 26 26 27 27 27 27 28 28 28 28 29 29 29 29 30 30 30 30 31 31 31 31
r0 0 8 16 24 1 9 17 25 2 10 18 26 3 11 19 27 4 12 20 28 5 13 21 29 6 14 22 30 7 15 23 31
r1 0 8 16 24 1 9 17 25 2 10 18 26 3 11 19 27 4 12 20 28 5 13 21 29 6 14 22 30 7 15 23 31
r2 0 8 16 24 1 9 17 25 2 10 18 26 3 11 19 27 4 12 20 28 5 13 21 29 6 14 22 30 7 15 23 31
r3 0 8 16 24 1 9 17 25 2 10 18 26 3 11 19 27 4 12 20 28 5 13 21 29 6 14 22 30 7 15 23 31

Figure 1: Bitsliced representation. The upper part denotes 32 elements in standard representation packed in 4 registers. Numbers denote the index of the field element. Shades of gray denote the different bits within a $\mathbb{F}_{16}$ element. The lower part denotes the bitsliced field elements with the least significant bit of each element packed in `r0`.

$$\begin{aligned}
a \cdot b &= (a_1 \cdot y + a_0) \cdot (b_1 \cdot y + b_0) \\
&= (a_1 \cdot b_1) \cdot y^2 + (a_0 \cdot b_1 + a_1 \cdot b_0) \cdot y + a_0 \cdot b_0 \\
&= (a_1 \cdot b_1) \cdot y^2 + ((a_0 + a_1) \cdot (b_0 + b_1) + a_0 \cdot b_0 + a_1 \cdot b_1) \cdot y + a_0 \cdot b_0 \\
&= (a_1 \cdot b_1) \cdot (y + x) + ((a_0 + a_1) \cdot (b_0 + b_1) + a_0 \cdot b_0 + a_1 \cdot b_1) \cdot y + a_0 \cdot b_0 \\
&= ((a_0 + a_1) \cdot (b_0 + b_1) + a_0 \cdot b_0) \cdot y + a_0 \cdot b_0 + a_1 \cdot b_1 \cdot x
\end{aligned}$$

However, this approach is rather slow on 32-bit (or larger) platforms as it utilizes the available 32-bit arithmetic inefficiently. We, hence, opt for bitslicing the field elements into four registers and implementing the multiplication using only logic operations. This is particularly useful when 32 or more $\mathbb{F}_{16}$ elements need to be multiplied by a single $\mathbb{F}_{16}$ elements which is almost always the case in Rainbow.

Bitslicing. As two $\mathbb{F}_{16}$ elements fit into one byte, we can fit eight $\mathbb{F}_{16}$ elements into one 32-bit register. However, we can achieve significantly faster $\mathbb{F}_{16}$ multiplication routines that run in constant time, when we bitslice the field elements into 4 separate registers holding a total of 32 elements. To make use of fast bitsliced multiplication, we need a way of converting a packed nibble representation of $\mathbb{F}_{16}$ elements into a bitsliced representation. A straightforward approach would load each field element individually, mask out the desired bit and pack it into the corresponding registers in the same order as the inputs. However, it is much more efficient to load 32 elements at once into four registers, and reorganizing the elements in an interleaved fashion as illustrated in Figure 1. Each row corresponds to a register containing 8 field elements. The colors denote the bit within the field element where light gray is the least significant bit, while dark gray is the most significant bit. This approach is similar to the one proposed by Chou for McBits [Cho17]. [1] This interleaving can be implemented efficiently in 28 cycles as shown in Appendix B. The same code can be used for the transformation from bitsliced representation to normal representation. The correct order of the field elements will be restored when reversing the bitslicing. Note that addition in $\mathbb{F}_{16}$ is bitwise XOR and, hence, behaves the same on the bitsliced representation.

[1] We may regard this as computing a transposition of binary matrices.

Bitsliced Multiplication. We first consider $\mathbb{F}_4$ multiplication, then use it to construct $\mathbb{F}_{16}$ multiplication, and then apply multiple simplifications to achieve a minimal instruction sequence. There are multiple approaches to arrive at the same instruction sequence, but we find this description the most intuitive to follow.

$\mathbb{F}_4$ multiplication Recall that a $\mathbb{F}_4$ element ($\mathbb{F}_2[x]/(x^2+x+1)$) is represented by two bits a_0, a_1, s.t., $a = a_1 \cdot x + a_0$. When multiplying two elements $a = a_1 \cdot x + a_0$ and $b = b_1 \cdot x + b_0$, we obtain $c = a \cdot b = c_1 x + c_0$. As $x \cdot x \equiv x + 1$, $x \cdot (x+1) \equiv 1$, and $(x+1) \cdot (x+1) \equiv x \pmod{x^2+x+1}$, it is easy to see that we can compute c_0, c_1, by computing
$c_0 = a_0 \cdot b_0 + a_1 \cdot b_1$
$c_1 = a_1 \cdot b_0 + (a_0 + a_1) \cdot b_1$

where $\cdot$ denotes logical `AND` and $+$ denotes `XOR`. This can be very efficiently computed on bitsliced elements.

Constructing $\mathbb{F}_{16}$ multiplication When multiplying two $\mathbb{F}_{16}$ elements $a = (a_3 \cdot x + a_2) \cdot y + a_1 \cdot x + a_0$ and $b = (b_3 \cdot x + b_2) \cdot y + b_1 \cdot x + b_0$, we can rewrite them by using two $\mathbb{F}_4$ elements as $a = \alpha_1 \cdot y + \alpha_0$, $b = \beta_1 \cdot y + \beta_0$ with $\alpha_0, \alpha_1, \beta_0, \beta_1 \in \mathbb{F}_4$. We can, thus, write

$$\begin{aligned}
a \cdot b &= (\alpha_1 \cdot y + \alpha_0) \cdot (\beta_1 \cdot y + \beta_0) \\
&= (\alpha_1 \cdot \beta_1) \cdot y^2 + (\alpha_0 \cdot \beta_1 + \alpha_1 \cdot \beta_0) \cdot y + \alpha_0 \cdot \beta_0 \\
&= (\alpha_0 \cdot \beta_1 + \alpha_1 \cdot \beta_0 + \alpha_1 \cdot \beta_1) \cdot y + \alpha_0 \cdot \beta_0 + (\alpha_1 \cdot \beta_1) \cdot x \\
&= \gamma_1 y + \gamma_0
\end{aligned}$$

We can now consider γ_0 and γ_1 separately and substitute the $\mathbb{F}_4$ multiplication.

$$\begin{aligned}
\gamma_0 &= \alpha_0 \cdot \beta_0 + (\alpha_1 \cdot \beta_1) \cdot x \\
&= (a_1 \cdot b_0 + (a_0 + a_1) \cdot b_1) \cdot x + a_0 \cdot b_0 + a_1 \cdot b_1 + \\
&\quad ((a_3 \cdot b_2 + (a_2 + a_3) \cdot b_3) \cdot x + a_2 \cdot b_2 + a_3 \cdot b_3) \cdot x \\
&= a_0 \cdot b_0 + a_1 \cdot b_1 + a_3 \cdot b_2 + (a_2 + a_3) \cdot b_3 + \\
&\quad (a_1 \cdot b_0 + (a_0 + a_1) \cdot b_1 + a_3 \cdot b_2 + (a_2 + a_3) \cdot b_3 + a_2 \cdot b_2 + a_3 \cdot b_3) \cdot x \\
&= a_0 \cdot b_0 + a_1 \cdot b_1 + a_3 \cdot b_2 + (a_2 + a_3) \cdot b_3 + \\
&\quad (a_1 \cdot b_0 + (a_0 + a_1) \cdot b_1 + (a_2 + a_3) \cdot b_2 + a_2 \cdot b_3) \cdot x \\
&= c_1 \cdot x + c_0
\end{aligned}$$

Hence, the least significant bits of the result can be computed as

$$\begin{aligned}
c_0 &= a_0 \cdot b_0 + a_1 \cdot b_1 + a_3 \cdot b_2 + (a_2 + a_3) \cdot b_3 \\
c_1 &= a_1 \cdot b_0 + (a_0 + a_1) \cdot b_1 + (a_2 + a_3) \cdot b_2 + a_2 \cdot b_3
\end{aligned}$$

We proceed similarly for γ_1:

$$\begin{aligned}
\gamma_1 &= \alpha_0 \cdot \beta_1 + \alpha_1 \cdot \beta_0 + \alpha_1 \cdot \beta_1 \\
&= (a_1 \cdot b_2 + (a_0 + a_1) \cdot b_3) \cdot x + a_0 \cdot b_2 + a_1 \cdot b_3 + \\
&\qquad (a_3 \cdot b_0 + (a_2 + a_3) \cdot b_1) \cdot x + a_2 \cdot b_0 + a_3 \cdot b_1 + \\
&\qquad (a_3 \cdot b_2 + (a_2 + a_3) \cdot b_3) \cdot x + a_2 \cdot b_2 + a_3 \cdot b_3 \\
&= a_2 \cdot b_0 + a_3 \cdot b_1 + (a_0 + a_2) \cdot b_2 + (a_1 + a_3) \cdot b_3 + \\
&\qquad (a_3 \cdot b_0 + (a_2 + a_3) \cdot b_1 + (a_1 + a_3) \cdot b_2 + (a_0 + a_1 + a_2 + a_3) \cdot b_3) \cdot x \\
&= c_3 \cdot x + c_2
\end{aligned}$$

And hence,

$$
\begin{aligned}
c_2 &= a_2 \cdot b_0 + a_3 \cdot b_1 + (a_0 + a_2) \cdot b_2 + (a_1 + a_3) \cdot b_3 \\
c_3 &= a_3 \cdot b_0 + (a_2 + a_3) \cdot b_1 + (a_1 + a_3) \cdot b_2 + (a_0 + a_1 + a_2 + a_3) \cdot b_3
\end{aligned}
$$

Now that we have established how $a \cdot b$ is calculated, we need to come up with an instruction sequence that does so efficiently. Consider the most common multiplication case within Rainbow: We have a large number (≥ 32) of field elements $a^{(i)}$ which are multiplied by a single field element b and then added to a bitsliced accumulator $c^{(i)}$. This is, for example, the case in the matrix-vector multiplication. In this case, it is best to bitslice $a^{(i)}$ and keep b in nibble-sliced representation. For sake of explanation, we assume that we are multiplying exactly 32 elements $a^{(0)}, \ldots, a^{(31)}$ which are bitsliced into four registers. The register containing the least significant bits of $a^{(0)}, \ldots, a^{(31)}$ is denoted as $a_0 = a_0^{(0)}, \ldots, a_0^{(31)}$ and similarly for $a_1, \ldots, a_3$ and $c_0, \ldots, c_3$. b is stored in the least significant four bits of a register, with b_0 denoting the least significant bit.

Algorithm 1 shows the instruction sequence that implements the computation of the product and accumulates it into $c_0, \ldots, c_3$. If only a multiplication is needed, but no accumulation, $c_0, \ldots, c_3$ first need to be initialized to zero. The instruction sequence heavily relies on using conditional execution to only execute the additions of a_i if certain bits of b are set. We compute $a_0 + a_1$ and $a_2 + a_3$ in two separate registers `tmp0`, `tmp1` as those are used both in c_1, c_3 and c_0, c_1, c_3 respectively. Also, we save another cycle by storing $(b_2 \cdot a_2) + (b_3 \cdot a_3)$ in a temporary register `tmp2` and $(b_2 \cdot a_3) + (b_3 \cdot (a_2 + a_3))$ in `tmp3` which is required to compute c_1, c_2 and c_0, c_1, c_3 respectively. Another shortcut that we have been using is line 16, which is functionally equivalent to computing

```
mov tmp3, #0
tst b, #4
```

in a single cycle. In total our instruction sequence requires 32 clock cycles, i.e., one clock cycle for each field multiplication.

This approach is directly extensible to parameter sets using $\mathbb{F}_{256}$ (RainbowIII and RainbowV).

3.2 $\mathbb{F}_{16}$ Matrix Inversion

Besides $\mathbb{F}_{16}$ multiplication, the Rainbow signature requires solving two matrix equations. Since if A^{-1} exists, $A\mathbf{x} = \mathbf{b} \leftrightarrow \mathbf{x} = A^{-1}\mathbf{b}$, we may without much loss of generality consider matrix inversion as a part of the signing procedure. As it operates on secret inputs, it is required to be constant-time which is not the case in a straightforward implementation of Gaussian elimination. We use an adapted version of the constant-time Gauss-Jordan elimination first presented by Bernstein, Chou, and Schwabe [BCS13]. Rainbow's constant time variant is illustrated in Algorithm 2 and is essentially the same as in the Rainbow reference implementation. However, for an implementation we need to choose how to implement the field arithmetic.

Field inversion. For $\mathbb{F}_{16}$ inversion (line. 9) the most efficient implementation uses a constant-time table look-up. As the number of possible input values is small (16), we can pack the look-up table ($16 \cdot 4$ bit) into the 16-bit immediate arguments of 4 `mov` instructions and then select the right bits by shifting them into the right place. The code for the $\mathbb{F}_{16}$ representation used in Rainbow is shown in Algorithm 3. For larger fields (e.g., $\mathbb{F}_{256}$) this approach does not work, and one would rather store a table in flash, loop through it, and conditionally select the right element. For $a = 0$, the inverse doesn't exist and special treatment is needed, i.e., the entire matrix inversion fails and `fail` $= 1$. In that case, the matrix gets discarded and one samples a new set of vinegar variables.

Algorithm 1 $\mathbb{F}_{16}$ Multiply and Accumulate Instruction Sequence

```
Input:  32 $\mathbb{F}_{16}$ elements bitsliced into $a_0$, $a_1$, $a_2$, $a_3$
Input:  1 $\mathbb{F}_{16}$ element in the least significant nibble of $b$
Input:  32 $\mathbb{F}_{16}$ elements bitsliced in the accumulator $c_0$, $c_1$, $c_2$, $c_3$
Output: Each of the elements in $a_i$ multiplied by b and added to $c_i$
tst $b$, #1
itttt ne                                  ▷ conditional exec. if $b\&1 \neq 0$
eorne $c_0$, $c_0$, $a_0$                 ▷ $c_0$ += $b_0 \cdot a_0$
eorne $c_1$, $c_1$, $a_1$                 ▷ $c_1$ += $b_0 \cdot a_1$
eorne $c_2$, $c_2$, $a_2$                 ▷ $c_2$ += $b_0 \cdot a_2$
eorne $c_3$, $c_3$, $a_3$                 ▷ $c_3$ += $b_0 \cdot a_3$
eor tmp0, $a_0$, $a_1$                    ▷ tmp0 $= a_0 + a_1$
eor tmp1, $a_2$, $a_3$                    ▷ tmp1 $= a_2 + a_3$
tst $b$, #2
itttt ne                                  ▷ conditional exec. if $b\&2 \neq 0$
eorne $c_0$, $c_0$, $a_1$                 ▷ $c_0$ += $b_1 \cdot a_1$
eorne $c_1$, $c_1$, tmp0                  ▷ $c_1$ += $b_1 \cdot (a_0 + a_1)$
eorne $c_2$, $c_2$, $a_3$                 ▷ $c_2$ += $b_1 \cdot a_3$
eorne $c_3$, $c_3$, tmp1                  ▷ $c_3$ += $b_1 \cdot (a_2 + a_3)$
mov tmp2, #0
ands tmp3, tmp2, $b$, lsr #3              ▷ Set tmp3=0; set cs flag if $b\&4 \neq 0$
itttt cs                                  ▷ conditional exec. if $b\&4 \neq 0$
eorcs tmp2, tmp2, $a_2$                   ▷ tmp2 $= b_2 \cdot a_2$
eorcs tmp3, tmp3, $a_3$                   ▷ tmp3 $= b_2 \cdot a_3$
eorcs $c_2$, $c_2$, $a_0$                 ▷ $c_2$ += $b_2 \cdot a_0$
eorcs $c_3$, $c_3$, $a_1$                 ▷ $c_3$ += $b_2 \cdot a_1$
tst $b$, #8
itttt ne                                  ▷ conditional exec. if $b\&8 \neq 0$
eorne $c_2$, $c_2$, $a_1$                 ▷ $c_2$ += $b_3 \cdot a_1$
eorne $c_3$, $c_3$, tmp0                  ▷ $c_3$ += $b_3 \cdot (a_0 + a_1)$
eorne tmp2, tmp2, $a_3$                   ▷ tmp2 $= b_2 \cdot a_2 + b_3 \cdot a_3$
eorne tmp3, tmp3, tmp1                    ▷ tmp3 $= b_2 \cdot a_3 + b_3 \cdot (a_2 + a_3)$
eor $c_0$, $c_0$, tmp3                    ▷ $c_0$ += $b_2 \cdot a_3 + b_3 \cdot (a_2 + a_3)$
eor $c_1$, $c_1$, tmp2                    ▷ $c_1$ += $b_2 \cdot a_2 + b_3 \cdot a_3$
eor $c_1$, $c_1$, tmp3                    ▷ $c_1$ += $b_2 \cdot a_3 + b_3 \cdot (a_2 + a_3)$
eor $c_2$, $c_2$, tmp2                    ▷ $c_2$ += $b_2 \cdot a_2 + b_3 \cdot a_3$
eor $c_3$, $c_3$, tmp3                    ▷ $c_3$ += $b_2 \cdot a_3 + b_3 \cdot (a_2 + a_3)$
```

Algorithm 2 Matrix inversion using constant-time Gaussian elimination (for us $\mathbb{F} = \mathbb{F}_{16}$)

```
Input: Matrix $A \in \mathbb{F}^{o \times o}$
Output: Inverse $A^{-1} \in \mathbb{F}^{o \times o}$
Output: fail $\in \{0, 1\}$, 1 if $A$ is not invertible
$A' \leftarrow (A|I_o) \in \mathbb{F}^{o \times 2 \cdot o}$
fail $\leftarrow 0$
for $i \leftarrow 0, \ldots, o-1$ do
    for $j \leftarrow i+1, \ldots, o-1$ do                    ▷ make sure $A'_{i,i} \neq 0$
        $p \leftarrow A'_{i,i}$
        for $k \leftarrow i, \ldots, 2 \cdot o - 1$ do
            if $p = 0$ then $A'_{i,k} \leftarrow A'_{i,k} + A'_{j,k}$      ▷ needs to be constant-time
    if $A'_{i,i} = 0$ then fail $\leftarrow 1$                  ▷ needs to be constant-time
    $p^{-1} \leftarrow A'^{-1}_{i,i}$                              ▷ constant-time inversion in $\mathbb{F}$
    for $k \leftarrow i, \ldots, 2 \cdot o - 1$ do                ▷ normalize row $i \rightarrow A'_{i,i} = 1$
        $A'_{i,k} \leftarrow p^{-1} \cdot A'_{i,k}$
    for $j \leftarrow 0, \ldots, o-1$ do               ▷ subtract from other rows $\rightarrow A'_{j,i} = 0$
        if $j = i$ then continue
        for $k \leftarrow i, \ldots, 2 \cdot o - 1$ do
            $A'_{j,k} \leftarrow A'_{j,k} + A'_{j,i} \cdot A'_{i,k}$
$(I_o|A^{-1}) \leftarrow A'$
return $A^{-1}$, fail
```

Note that a field element at index i can be efficiently retrieved from a packed matrix representation (starting at address a) using the following instruction sequence:

```
lsrs i, i, #1
ldrb pivot, [a, i]
ite cs
lsrcs pivot, pivot, #4
andcc pivot, pivot, #0xF
```

Field multiplication. The optimal choice for implementing $\mathbb{F}_{16}$ multiplication is less obvious. To achieve the fastest multiplication one would want to keep the entire extended matrix A' in bitsliced representation. However, when making sure that the pivot element

Algorithm 3 15 cycle table lookup to a 16 element look-up table encoded into the immediate arguments of 4 `mov` instructions. The i-th bit of each constant encodes one bit of the inverse of i. For example, the inverse of $yx + y + x + 1$ (encoded as `0xF`) is y (`0x4`), i.e., the 15-th bit is only set in the third constant (`0xFA30`).

```
movw $t$, #0x58D6
lsr.w $e_{inv}$, $t$, $e$
and.w $e_{inv}$, #1
movw $t$, #0x2B9C
lsr.w $t$, $t$, $e$
and.w $t$, #1
orr.w $e_{inv}$, $e_{inv}$, $t$, lsl #1
movw $t$, #0xFA30
lsr.w $t$, $t$, $e$
and.w $t$, #1
orr.w $e_{inv}$, $e_{inv}$, $t$, lsl #2
movw $t$, #0x65F0
lsr.w $t$, $t$, $e$
and.w $t$, #1
orr.w $e_{inv}$, $e_{inv}$, $t$, lsl #3
```

$$\begin{array}{cccccccc} & & & & \multicolumn{4}{c}{\text{bitsliced}} \\ \hline a_{00} & a_{01} & \dots & a_{0o} & 1 & 0 & \dots & 0 \\ a_{10} & a_{11} & \cdots & a_{1o} & 0 & 1 & \dots & 0 \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\ a_{o0} & \dots & & a_{oo} & 0 & \dots & & 1 \end{array} \xrightarrow{\text{constant-time Gaussian elimination, } b = a^{-1}} \begin{array}{cccccccc} & & & & \multicolumn{4}{c}{\text{bitsliced}} \\ \hline 1 & 0 & \dots & 0 & b_{00} & b_{01} & \dots & b_{0o} \\ 0 & 1 & \dots & 0 & b_{10} & b_{11} & \cdots & b_{1o} \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\ 0 & \dots & & 1 & b_{o0} & \dots & & b_{oo} \end{array}$$

Figure 2: Partially bitsliced inversion. Input is in normal representation. Output is in bitsliced representation.

is not zero in lines 4 to 7 and when inverting the pivot element in line 9, one needs to access individual field elements which is tedious and inefficient when working on a bitsliced matrix. Hence, it is faster to keep the matrix in normal (packed nibble-) representation, only perform the bitslicing ad hoc just before multiplying and convert back just after. It is notable, that that individual element accesses only occur to the left half of the matrix. Hence, we can bitslice the right half and keep it bitsliced throughout the computation. This is illustrated in Figure 2. As the output of the matrix inversion is always the input to matrix multiplication, it is possible to return the bitsliced inverse.

An additional speed-up is achieved by letting the inner loops in line 6 and line 14 always start at $k = 0$. This does not change the result, but greatly simplifies the loop control and the overhead of accessing the packed elements. Overall, this results in a small speed-up even though the number of additions and multiplications is slightly increased.

Avoiding matrix inversion As the inverse of the matrix is multiplied by the variables y directly after inversion and is not used at any other point in the Rainbow signature generation, one can also eliminate the matrix inversion and simply solve for x in $Ax = y$. The Gauss elimination proceeds similar to Algorithm 2, but one cannot benefit from bitslicing the right part of the matrix. This approach is 33 000 cycles faster than inverting the matrix first and then multiplying. Unfortunately, according to the Rainbow specification the vinegar variables and the matrix are sampled from the same PRG. In the first layer, a new matrix is sampled until it is invertible before the vinegar variables of the second layer are sampled. If one wants to merge these steps one would have to change the way the matrix and variables are sampled or would have to roll back the PRG in case the matrix is not invertible before sampling another matrix. Therefore, we only use this approach to eliminate the inversion in the second layer of Rainbow.

3.3 Evaluating the Public Map $\mathcal{P}$

One of the key advantages of Rainbow, is a very simple verification procedure: One applies the public map $\mathcal{P}$ to the signature z and verifies that the result matches the (randomized) hash of the message. The application of $\mathcal{P}$ consists of the substitution of the variables $z_1, ..., z_n$ into the system of equations represented by the public key. The public key is stored as a Macaulay matrix $A \in \mathbb{F}^{\left(\!\binom{n}{2}\!\right) \times m}$ which allows us to sequentially load it exactly once while processing the variables.

Macaulay matrix indexing: Here, by writing the index set as $\left(\!\binom{n}{2}\!\right) \times m$ we mean that the indices in $A_{i,j,k}$ satisfy $0 \leq i \leq j < n$, $0 \leq k < m$.

The standard procedure to compute $\mathcal{P}$ (which can be in constant time) is illustrated in Algorithm 4 and requires $\left(\!\binom{n}{2}\!\right) \cdot m + \left(\!\binom{n}{2}\!\right)$ field multiplications. The documentation of

Algorithm 4 Traditional way of computing the public map $\mathcal{P}$

```
Input: Public Key $A \in \mathbb{F}^{\left(\!\binom{n}{2}\!\right) \times m}$ in Macaulay form
Input: Variables $z \in \mathbb{F}^n$
Output: $\mathcal{P}(z) \in \mathbb{F}^m$
$h \in \mathbb{F}^m \leftarrow 0$
for $i \leftarrow 0, \ldots, n-1$ do
    for $j \leftarrow i, \ldots, n-1$ do
        $t \in \mathbb{F} \leftarrow z_i \cdot z_j$
        for $k \leftarrow 0, \ldots, m-1$ do
            $h_k \leftarrow h_k + A_{i,j,k} \cdot t$
return $h$
```

Algorithm 5 Our way of computing public map $\mathcal{P}$ in variable time

```
Input: Public Key $A \in \mathbb{F}^{\left(\!\binom{n}{2}\!\right) \times m}$ in Macaulay form
Input: Variables $z \in \mathbb{F}^n$
Output: $\mathcal{P}(z) \in \mathbb{F}^m$
$h' \in \mathbb{F}^{|\mathbb{F}| \times m} \leftarrow 0$
for $i \leftarrow 0, \ldots, n-1$ do
    for $j \leftarrow i, \ldots, n-1$ do
        $t \in \mathbb{F} \leftarrow z_i \cdot z_j$
        for $k \leftarrow 0, \ldots, m-1$ do
            $h'_{t,k} \leftarrow h'_{t,k} + A_{i,j,k}$
$h \leftarrow h'_1$
for $t \in \mathbb{F} \setminus \{0, 1\}$ do
    for $k \leftarrow 0, \ldots, m-1$ do
        $h_k \leftarrow h_k + h'_{t,k} \cdot t$
return $h$
```

the UOV-derived NIST submissions [DCK+20a, BPSV19, SPK17] each describe some variation of this.[2]

However, we propose a different and much more efficient way to compute the public map only requiring $(|\mathbb{F}| - 2) \cdot m + \left(\!\binom{n}{2}\!\right)$ multiplications. This method is not mentioned in previous documents describing UOV-based MQ systems. Our modified procedure for computing $\mathcal{P}$ is illustrated in Algorithm 5. One key observation is that we do not need the verification to have a runtime that is independent of the inputs as both the signature and the public key are considered public. Therefore, we propose to use one accumulator (of m field elements) for each possible value of the monomial $z_i \cdot z_j$. The corresponding column of the matrix A is then added to the accumulator corresponding to the value of $z_i \cdot z_j$. This obviously may leak the value of $z_i \cdot z_j$ through a cache timing side-channel, but that does not need to concern us. The computation of the monomials within the loop costs $\left(\!\binom{n}{2}\!\right)$ multiplications. In the very end, we combine the accumulators by multiplying each of them with the corresponding $\mathbb{F}_{16}$ element requiring $(|\mathbb{F}| - 2) \cdot m$ multiplications as multiplications by 0 and 1 are trivial. This allows a massive speed-up at the cost of additional memory large enough to hold $|\mathbb{F}| \cdot m$ field elements (or $(|\mathbb{F}| - 1) \cdot m$ if one omits the buffer for $z_i \cdot z_j = 0$.) In the case of `rainbowI`, the additional memory of $16 \cdot 64/2 = 512$ bytes is negligible. For the larger parameter sets using $\mathbb{F}_{256}$ this approach is probably still worthwhile on some platforms. For `rainbowIII` ($m = 80$) and `rainbowV` ($m = 100$), $256 \cdot 80 = 20\,480$ bytes and $256 \cdot 100 = 25\,600$ bytes are required respectively.

One could further reduce the number of multiplications to $(\log_2(|\mathbb{F}_{16}|) - 1) \cdot m = 3 \cdot m$ by instead doing more additions. First, we sum up the accumulators corresponding to the elements that have the least significant bit is set, i.e., $1, x+1, y+1, y+x+1, yx+1, yx+x+1, yx+y+1, yx+y+x+1$. Then, we sum up the accumulators corresponding to the elements that have the second bit set $(x, x+1, y+x, y+x+1, y+x, y+x+1, yx+x, yx+x+1)$, multiply the sum by x, and then added to the first sum. Similarly, we proceed for the other two bits corresponding to y, and yx. That approach is then similar to the one by Cheng, Chou, Niederhagen, and Yang [CCNY12, Sec. 3.1]. However, we chose not to

[2] Note that the **secret** MQ evaluation we used during signing is performed like it was in [DCK+20a], which uses a separate buffer in the inner two loops and only multiply by z_i outside these loops, *because multiplying a vector is much faster than multiplying individual elements.* In addition, we accelerated multiplications as in the previous section.

Algorithm 6 Bitsliced Multiply and Accumulate for $\mathbb{F}_{16} = \mathbb{F}_2[x]/(x^4 + x + 1)$

Input: 32 $\mathbb{F}_{16}$ elements bitsliced into a_0, a_1, a_2, a_3
Input: 1 $\mathbb{F}_{16}$ element in the least significant nibble of b
Input: 32 $\mathbb{F}_{16}$ elements bitsliced in the accumulator c_0, c_1, c_2, c_3
Output: Each of the elements in a_i multiplied by b and added to c_i

```
eor tmp0, $a_0$, $a_3$
eor tmp1, $a_2$, $a_3$
eor tmp2, $a_1$, $a_2$

tst $b$, #1
itttt ne
eorne $c_0$, $c_0$, $a_0$
eorne $c_1$, $c_1$, $a_1$
eorne $c_2$, $c_2$, $a_2$
eorne $c_3$, $c_3$, $a_3$
tst $b$, #2
itttt ne
eorne $c_0$, $c_0$, $a_3$
eorne $c_1$, $c_1$, tmp0
eorne $c_2$, $c_2$, $a_1$
eorne $c_3$, $c_3$, $a_2$

tst $b$, #4
itttt ne
eorne $c_0$, $c_0$, $a_2$
eorne $c_1$, $c_1$, tmp1
eorne $c_2$, $c_2$, tmp0
eorne $c_3$, $c_3$, $a_1$

tst $b$, #8
itttt ne
eorne $c_0$, $c_0$, $a_1$
eorne $c_1$, $c_1$, tmp2
eorne $c_2$, $c_2$, tmp1
eorne $c_3$, $c_3$, tmp0
```

implement this trick as the performance gain is negligible and the final multiplications already take less than 1% of our total run-time.

Instead, as variable run-time is of no concern, we can further improve the procedure:

$\mathbb{F}_{16}$ Multiplication using LUTs. As the signature z is public, we may use look-up tables to compute the $\mathbb{F}_{16}$ multiplications. This is particularly useful when individual field elements are to be multiplied when computing the monomials $z_i \cdot z_j$ as those multiplications are tedious to bitslice. We replace those multiplications by a look-up to a 256 element look-up table. For efficiency, we do not pack the elements in the look-up table and it, consequently, occupies 256 bytes in the case of $\mathbb{F}_{16}$. For the multiplications of the accumulators in the end, we stick with bitsliced multiplications as those bulk multiplications outperform table look-ups.

Skipping parts of the public key. Whenever $z_i \cdot z_j = 0$, the corresponding entries in A have no impact on the result h. This is the case when either $z_i = 0$ or $z_j = 0$. When $z_j = 0$, the inner loop can be skipped saving load, addition, and store operations of m field elements. Even more importantly, when $z_i = 0$, both inner loops can be skipped which saves $(n - i) \cdot m$ operations. The additional cost of branching depending on the variables is by far outweighed by the savings: Processing one column takes 37 cycles (3 cycles for multiplication using a LUT, 18 cycles load of accumulator and column, 8 cycle addition, and 8 cycle store.) Checking for $z_j = 0$ in the inner loop costs two cycles (`cmp`, `beq`). As it is expected to skip the computation in $\frac{1}{16}$ of cases, implementing the check pays off slightly. For the outer loop, the speed-up is more pronounced as we skip $n/2 = 50$ columns on average. This saves more than 1850 cycles and is expected to happen for every 16th execution, i.e., saving significantly more than the 2 cycles needed for the check.

4 Alternative $\mathbb{F}_{16}$ Representation

In addition to $\mathbb{F}_{16}$ tower field representation as mandated by the Rainbow specification, we have also experimented with using the direct representation $\mathbb{F}_{16} = \mathbb{F}_2[x]/(x^4 + x + 1)$. By switching to that representation one can implement bitsliced multiplication using the instruction sequence presented in Algorithm 6. This sequence needs only 27 cycles (one cycle per instruction) compared to 32 cycles for the multiplication for the tower field representation.

Unfortunately, Rainbow keys, signatures, and all values sampled in the signing procedure are using the tower field representation and one would have to convert to and from the direct representation to make use of Algorithm 6. The conversion can only be done by multiplication by a 4×4 bit matrix while bitsliced. This conversion outweighs the performance gain from faster multiplication.

Consequently, the only way to benefit from this more efficient representation is to change the Rainbow specification to use $\mathbb{F}_2[x]/(x^4 + x + 1)$ everywhere. For a Cortex-M4 implementation, there is no benefit to use the tower field implementation and a change of the specification would only make it faster Clearly, the same is the case for other bitsliced implementations which are likely to be used on other microcontroller platforms. For AVX2 implementations (e.g., the one from [DCK+20a]) a change of the field representation does not have any impact on performance as field multiplication is implemented using constant-time table lookups. Hence, we argue that the Rainbow specification should be changed to use the direct representation for $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$.

When changing the field representation, one also has to update the lookup tables for the inverse describes in Section 3.2 and the variable-time multiplication described in Section 3.3. Besides that, all other parts of Rainbow remain the same.

5 Results

This section presents the results when applying the optimization presented in this paper to the reference implementation that is part of the Rainbow submission package [DCK+20a].

Platform. Due to Rainbow's large keys, we use the somewhat non-standard microcontroller `EFM32GG11B`[3] which is part of Silicon Labs' Giant Gecko Starter Kit. It comes with 512 kB of RAM and 2 MiB of flash memory. The core can run at a frequency of up to 72 MHz. It comes with a TRNG which we use to obtain the required randomness in Rainbow. Another feature of the `EFM32GG11B` that makes it an attractive target for post-quantum cryptography is that it comes with a cryptography accelerator supporting `AES128`, `AES256`, `SHA-1`, `SHA256`, and 256-bit multiplication. Section 5.2 presents how using the `AES256` and `SHA256` changes the performance of our implementations.

`SHA2` and `AES256`. For hashing Rainbow uses `SHA2`. We use the `SHA2` implementation from SUPERCOP[4]. Additionally, Rainbow uses `AES256` extensively for expanding matrices from a random seed. We use the bitsliced implementation[5] by Adomnicai and Peyrin [AP20].

Benchmarking. We base our benchmarking on the testing and benchmarking framework pqm4 [KRSS]. As pqm4 is built for the `STM32F407`, we adapt their hardware abstraction layer to support the Giant Gecko. We use the `arm-none-eabi-gcc` compiler version 10.2.0 and compile with `-O3`. We do not run the Giant Gecko at the maximum frequency, but instead, down-clock it to 16 MHz and configure it to have zero wait states when fetching instructions and data from flash. This ensures that the resulting cycle counts are comparable to the ones produced by pqm4 on the `STM32F407`. Similar to pqm4, we use the built-in SysTick timer to count cycles. As the `EFM32GG11B` is not commonly used in the literature, we perform experiments to confirm that the timing behavior is comparable to the `STM32F407`. We benchmarked the schemes from pqm4 [KRSS] and found a very small cycle count differences of less than 1%.

[3] The full name of core is `EFM32GG11B820F2048GL192`.

[4] `https://bench.cr.yp.to/supercop.html`

[5] `https://github.com/aadomn/aes`

5.1 RainbowI with and without precomputation

Table 2 contains the performance results obtained on the EFM32GG11B. The runtime of our implementation of verification heavily depends on the signature as explained in Section 3.3. Signing also has varying run-time depending on how many attempts are needed until the matrix inversion succeeds. Hence, we run 10 000 iterations of signing and verification (with different messages) and report the average. For comparison, we report the performance results of Moya Riera [MR19] for the round 2 parameters. Despite, the larger parameters, we achieve a reduction in cycle counts by 27%, 47%, and 85% for key generation, signing, and verification respectively. For reference for the other parameter sets, we also report the cycle counts for the C implementation that is part of the Rainbow submission package [DCK+20a].

According to the specification, the Rainbow secret key is stored in nibble-packed representation. In our implementation, for each part of the secret key, the first step is to convert it to bitsliced representation. This change of representation can also be precomputed. We include the precomputation in the key generation, but it could also be implemented differently. This saves around 187 000 cycles for signing. However, this makes the secret key representation implementation-specific and platform-specific (due to Endianness) which may not be desirable. For `rainbowIcompressed`, this approach does not work as the secret key only consists of a seed that is used to re-sample the secret key during signing. One could also consider precomputing the bitsliced representation of the public key. However, this would only result in negligible speed-up due to the optimized verification algorithm that uses very few multiplications. Additionally, having an implementation-specific public key representation appears even less enticing.

The results for the alternative $\mathbb{F}_{16}$ representation described in Section 4 is also shown in Table 2. It consistently reduces the runtime by up to 7% for signing.

Table 3 presents the stack requirement and code size of our implementations. As we do not use any dynamically allocated memory, all intermediate variables are included in the stack. It does not include keys, the message, and the signature as those are allocated by the calling code. We measure the stack consumption by writing a fixed value to each byte of the stack, running the procedure and then checking how much of the stack has been overwritten.

For obtaining the code-size, we run `arm-none-eabi-size` on then binary that includes all the code required to execute, i.e., we strip out all unused code. However, this includes the platform code and we, hence, subtract 21 kB to obtain the code-size of the Rainbow code. We additionally report the code size when only a part of the signature scheme is needed. If used by the procedure, the code size includes 5 kB for `AES256` and 8 kB for `SHA256`. `SHA256` is only used in signing and verification. `AES256` is only used in key generation, signing, and for circumzenithal verification (`rainbowIcircumzenithal` and `rainbowIcompressed`).

Optimizing RAM and code size is not the primary target for our work; we merely report them for completeness. `classic` inherently provides competitive memory consumption for signing and verification. `circumzenithal` and `compressed` require significantly more RAM. However, one needs to take into account that they also have smaller keys. For example, `circumzenithal` public keys are almost 100 kB smaller than `classic` public keys. If keys reside in RAM, `circumzenithal` outperforms `classic` in terms of RAM consumption. Clearly, more RAM efficient implementations are possible, and it is an interesting area of future work.

5.2 Hardware Acceleration for SHA2 and AES

Symmetric cryptography is at the core of virtually all post-quantum cryptography schemes often making up the majority of cycles [KRSS] (e.g., up to 80% for Kyber [ABCG20], 81%

Table 2: Performance of RainbowI parameter sets on ARM Cortex-M4. Cycle counts are obtained on the EFM32GG11B running at 16 MHz. For signing and verification, the cycle counts are the average of 10 000 executions. For rainbowIclassic and rainbowIcircumzenithal, signing can be sped up by precomputing the bitsliced secret key. We include the precomputation in the key generation.

		clock cycles	
parameter set		w/o precomp.	w/ precomp.
rainbowIclassic (Round 2)	[MR19]	**K:** 134 354k	
		S: 1 815k	
		V: 1 619k	
rainbowIclassic	ref.	**K:** 417 316k	
		S: 5 433k	
		V: 3 529k	
	This work	**K:** 98 431k	**K:** 98 691k
		S: 957k	**S:** 770k
		V: 239k	**V:** 238k
rainbowIclassic $\mathbb{F}_{16} = \mathbb{F}_2[X]/(X^4 + X + 1)$	**This work**	**K:** 94 584k	**K:** 94 845k
		S: 907k	**S:** 719k
		V: 238k	**V:** 238k
rainbowIcircumzenithal	ref.	**K:** 462 322k	
		S: 5 422k	
		V: 27 965k	
	This work	**K:** 107 639k	**K:** 107 899k
		S: 955k	**S:** 769k
		V: 12 903k	**V:** 12 903k
rainbowIcircumzenithal $\mathbb{F}_{16} = \mathbb{F}_2[X]/(X^4 + X + 1)$	**This work**	**K:** 103 343k	**K:** 103 604k
		S: 902k	**S:** 717k
		V: 12 902k	**V:** 12 902k
rainbowIcompressed	ref.	**K:** 462 387k	
		S: 217 061k	
		V: 27 968k	
	This work	**K:** 107 711k	
		S: 56 643k	
		V: 12 903k	
rainbowIcompressed $\mathbb{F}_{16} = \mathbb{F}_2[X]/(X^4 + X + 1)$	**This work**	**K:** 103 415k	
		S: 54 778k	
		V: 12 902k	

Table 3: Stack consumption and code size of our Rainbow M4 implementation.

parameter set	stack [bytes]		code size [kB]	
`rainbowIclassic`	**K:**	40 696	**K:**	36
	S:	4 052	**S:**	32
	V:	812	**V:**	12
			all:	56
`rainbowIcircumzenithal`	**K:**	142 304	**K:**	27
	S:	4 052	**S:**	32
	V:	20 156	**V:**	22
			all:	51
`rainbowIcompressed`	**K:**	245 976	**K:**	27
	S:	224 240	**S:**	43
	V:	20 156	**V:**	22
			all:	53

for Dilithium [GKS20]). We report the cycles for our Rainbow implementations (with precomputation) in Table 4. When using software implementations for `AES` and `SHA2`, we see that for `rainbowIclassic` only 10% of signing and 4% of verification are spent in hashing. This looks very differently for circumzenithal verification (`rainbowIcircumzenithal` and `rainbowIcompressed`) where 92% are spent in symmetric primitives.

Interestingly, the Giant Gecko provides hardware support for the symmetric cryptography needed by Rainbow. We, hence, also report results using the hardware accelerator. This provides a vast speed-up for verification of `rainbowIcircumzenithal` and `rainbowIcompressed` of 13×. For `rainbowIclassic` the speed-up is less notable.

5.3 Comparison to other Post-Quantum Signature Schemes

Table 5 compares our `rainbowIclassic` implementation with the other to NISTPQC signature finalists: Dilithium [BLD+20] and Falcon [FHK+20]. Both have been optimized for the Cortex-M4 [GKS20, Por19]. The results shown are taken from the corresponding publications and have been obtained by benchmarking on the `STM32F407`. However, as the `EFM32GG11B` timings are very close, the results are comparable to ours.

For our implementation, we report the one with software implementations of `AES` and `SHA256`. Interestingly, both Falcon and Dilithium signing benefit from precomputation as well. For all implementations, precomputation is included in the key generation cycles.

Our implementation of `rainbowIclassic` signing is 4× faster than the state of the art Dilithium2 implementation and 4× faster than Falcon-512. Verification is 5× than Dilithium2 and 2× faster than Falcon-512. Consequently, our implementation of Rainbow on the Cortex-M4 is by far the fastest among the finalists of the NISTPQC competition.

Acknowledgements

This work has been supported by the European Commission through the ERC Starting Grant 805031 (EPOQUE). Taiwanese authors were supported by Taiwan Ministry of Science and Technology Grant 109-2221-E-001-009-MY3, Sinica Investigator Award AS-IA-109-M01, Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

Table 4: Hashing results for our implementations with precomputation. We report the cycles spent in `AES` and `SHA256` combined. We report both software results, and results using the hardware accelerator of the Giant Gecko.

parameter set	`AES`, `SHA2`	total cc		`AES`+`SHA2` cc	
`rainbowIclassic`	sw	**K:**	98 691k	**K:**	12 451k (13%)
		S:	770k	**S:**	78k (10%)
		V:	238k	**V:**	9k (4%)
	hw	**K:**	86 490k	**K:**	679k (1%)
		S:	697k	**S:**	4k (1%)
		V:	230k	**V:**	1k (0%)
`rainbowIcircumzenithal`	sw	**K:**	107 899k	**K:**	12 466k (12%)
		S:	769k	**S:**	78k (10%)
		V:	12 903k	**V:**	12 131k (92%)
	hw	**K:**	95 683k	**K:**	680k (1%)
		S:	694k	**S:**	4k (1%)
		V:	1 027k	**V:**	664k (65%)
`rainbowIcompressed`	sw	**K:**	107 711k	**K:**	12 466k (12%)
		S:	56 643k	**S:**	12 544k (22%)
		V:	12 903k	**V:**	12 131k (92%)
	hw	**K:**	95 494k	**K:**	680k (1%)
		S:	44 355k	**S:**	684k (2%)
		V:	1 026k	**V:**	664k (65%)

Table 5: Comparison to other NIST PQC finalist signature schemes. Signing benefits from precomputation of a different representation of the secret key compared to the reference implementation. The precomputation is included in the key generation of the respective implementations.

scheme	implementation	precomp.	cycle count	
`dilithium2`	[GKS20]	no	**K:**	1 315k
			S:	3 987k
			V:	1 259k
		yes	**K:**	2 267k
			S:	3 097k
			V:	1 259k
`falcon-512`	[Por19]	no	**K:**	171 294k
			S:	43 302k
			V:	504k
		yes	**K:**	187 485k
			S:	21 156k
			V:	504k
`rainbowIclassic`	**This work**	no	**K:**	98 431k
			S:	957k
			V:	239k
		yes	**K:**	98 691k
			S:	770k
			V:	238k

References

[ABB$^+$20] Jean-Philippe Aumasson, Daniel J. Bernstein, Ward Beullens, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, and Bas Westerbaan. SPHINCS+. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-m4 optimizations for R,M lwe schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):336–357, Jun. 2020. https://eprint.iacr.org/2020/012.

[AP20] Alexandre Adomnicai and Thomas Peyrin. Fixslicing aes-like ciphers: New bitsliced aes speed records on arm-cortex m and risc-v. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):402–425, Dec. 2020. https://eprint.iacr.org/2020/1123.

[ARM18] ARM. ARMv7-M architecture reference manual, 2018. https://developer.arm.com/documentation/ddi0403/ed.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastian Coron, editors, *Cryptographic Hardware and Embedded Systems – CHES 2013*, Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2013. Document ID: e801a97c500b3ac879d77bcecf054ce5, http://cryptojedi.org/papers/#mcbits.

[Beu20] Ward Beullens. Improved cryptanalysis of uov and rainbow. Available from https://eprint.iacr.org/2020/1343, 2020.

[BFM$^+$18] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! Faster Frodo for the ARM Cortex-M4. 2018. https://eprint.iacr.org/2018/1116.

[BLD$^+$20] Shi Bai, Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-DILITHIUM. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[BMKV20] Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in toom-cook multiplication: an application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):222–244, Mar. 2020. https://eprint.iacr.org/2020/268.

[BPSV19] Ward Beullens, Bart Preneel, Alan Szepieniec, and Frederik Vercautern. LUOV, submission to the nist post-quantum cryptography project, 2019. Available from https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-2/submissions/LUOV-Round2.zip.

[CCC+08] Anna Inn-Tung Chen, Chia-Hsin Owen Chen, Ming-Shing Chen, Chen-Mou Cheng, and Bo-Yin Yang. Practical-sized instances of multivariate PKCs: Rainbow, TTS, and ℓIC-derivatives. In Johannes Buchmann and Jintai Ding, editors, *PQCrypto*, volume 5299 of *Lecture Notes in Computer Science*, pages 95–108. Springer, 2008.

[CCC+09] Anna Inn-Tung Chen, Ming-Shing Chen, Tien-Ren Chen, Chen-Mou Cheng, Jintai Ding, Eric Li-Hsiang Kuo, Frost Yu-Shuang Lee, and Bo-Yin Yang. SSE implementation of multivariate PKCs on modern x86 CPUs. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2009.

[CCNY12] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, and Bo-Yin Yang. Solving quadratic equations with xl on parallel architectures. In Emmanuel Prouff and Patrick Schaumont, editors, *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 356–373. Springer, 2012.

[CFM+20] A. Casanova, J.-C. Faugère, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem. GeMSS. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J. Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings: New speed records for Saber and NTRU on Cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021. https://eprint.iacr.org/2020/1278.

[Cho17] Tung Chou. Mcbits revisited. In *Cryptographic Hardware and Embedded Systems - CHES 2017*, pages 213–231, 2017. https://eprint.iacr.org/2017/793.

[CLP+18] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Chen-Mou Cheng, and Bo-Yin Yang. Implementing 128-bit secure mpkc signatures. *IEICE Transactions*, E101-A(3):553–569, 2018.

[DCK+20a] Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow, submission to the nist post-quantum cryptography project, 2020. Available from www.pqcrainbow.org and https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Rainbow-Round3.zip.

[DCK+20b] Jintai Ding, Ming-Shing Chen, Matthias Kannwischer, Jacques Patarin, Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Response to recent paper by ward beullens, 2020. Available from http://precision.moscito.org/by-publ/recent/response-ward.pdf.

[DS05] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *Conference on Applied Cryptography and Network Security — ACNS 2005*, volume 3531 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2005.

[DYC+08] Jintai Ding, Bo-Yin Yang, Chia-Hsin Owen Chen, Ming-Shing Chen, and Chen-Mou Cheng. New differential-algebraic attacks and reparametrization of rainbow. In *Applied Cryptography and Network Security*, volume 5037 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2008. cf. http://eprint.iacr.org/2008/108.

[FHK$^+$20] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. Submission to the NIST Post-Quantum Cryptography Standardization Project [NIS], 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact Dilithium implementations on Cortex-M3 and Cortex-M4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):1–24, 2020. https://eprint.iacr.org/2020/1278.

[KO63] Anatolii Karatsuba and Yuri Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595–596, 1963. Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962. Scanned version on http://cr.yp.to/bib/1963/karatsuba.html.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced Oil and Vinegar signature schemes. In *Advances in Cryptology — EUROCRYPT 1999*, volume 1592 of *Lecture Notes in Computer Science*, pages 206–222. Jacques Stern, ed., Springer, 1999.

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $\mathbf{Z}_{2^m}[x]$ on cortex-m4 to speed up NIST PQC candidates. In *Applied Cryptography and Network Security*, pages 281–301, 2019. https://eprint.iacr.org/2018/1018.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[MR19] Joan Moya Riera. *Performance Analysis of Rainbow on ARM Cortex-M4*. Bachelor's thesis, Technische Universität München, 2019. http://hdl.handle.net/2117/169145.

[NIS] NIST, the US National Institute of Standards and Technology. Post-quantum cryptography standardization project. https://csrc.nist.gov/Projects/post-quantum-cryptography.

[PBB10] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. Cyclicrainbow - a multivariate signature scheme with a partially cyclic public key. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT*, volume 6498 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2010.

[Por19] Thomas Pornin. New efficient, constant-time implementations of falcon. Cryptology ePrint Archive, Report 2019/893, 2019. https://eprint.iacr.org/2019/893.

[Sho94] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *FOCS 1994*, pages 124–134. IEEE, 1994. https://ieeexplore.ieee.org/abstract/document/365700.

[SJA19] Hwajeong Seo, Amir Jalali, and Reza Azarderakhsh. SIKE round 2 speed record on ARM cortex-m4. In *Cryptology and Network Security - CANS*, pages 39–60, 2019. https://eprint.iacr.org/2019/535.

[SPK17] Kyung-Ah Shim, Cheol-Min Park, and Aeyoung Kim. Himq-3, submission to the nist post-quantum cryptography project, 2017. Available from `https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-1/submissions/HiMQ_3.zip`.

[Wol04] Christopher Wolf. Efficient public key generation for HFE and variations. In Ed Dawson and Wolfgang Klemm, editors, *Cryptographic Algorithms and their Uses - 2004, International Workshop, Gold Coast, Australia, July 5-6, 2004, Proceedings*, pages 78–93. Queensland University of Technology, 2004.

[YC05] Bo-Yin Yang and Jiun-Ming Chen. Building secure tame-like multivariate public-key cryptosystems: The new TTS. In *ACISP 2005*, volume 3574 of *Lecture Notes in Computer Science*, pages 518–531. Springer, July 2005.

A Toy Example of the Central Map of Rainbow

- $\mathbb{K} = \mathrm{GF}(7)$, $(v_1, o_1, o_2) = (2, 2, 2)$
- central map $\mathcal{Q} = (q^{(3)}, q^{(4)}, q^{(5)}, q^{(6)})$ with

$$\begin{aligned}
q^{(3)} &= x_1^2 + 3x_1x_2 + 5x_1x_3 + 6x_1x_4 + 2x_2^2 + 6x_2x_3 + 4x_2x_4 + 2x_2 + 6x_3 + 2x_4 + 5,\\
q^{(4)} &= 2x_1^2 + x_1x_2 + x_1x_3 + 3x_1x_4 + 4x_1 + x_2^2 + x_2x_3 + 4x_2x_4 + 6x_2 + x_4,\\
q^{(5)} &= 2x_1^2 + 3x_1x_2 + 3x_1x_3 + 3x_1x_4 + x_1x_5 + 3x_1x_6 + 6x_1 + 4x_2^2 + x_2x_3 + 4x_2x_4\\
&+ x_2x_5 + 3x_2x_6 + 3x_2 + 3x_3x_4 + x_3x_5 + 2x_3x_6 + 2x_3 + 3x_4x_5 + x_5 + 6x_6,\\
q^{(6)} &= 2x_1^2 + 5x_1x_2 + x_1x_3 + 5x_1x_4 + 5x_1x_6 + 6x_1 + 5x_2^2 + 3x_2x_3 + 5x_2x_5 + 4x_2x_6\\
&+ x_2 + 3x_3^2 + 5x_3x_4 + 4x_3x_5 + 2x_3x_6 + 4x_3 + x_4^2 + 6x_4x_5 + 3x_4x_6\\
&+ 4x_4 + 4x_5 + x_6 + 2.
\end{aligned}$$

- Goal: Find pre image $\mathbf{x} \in \mathbb{K}^6$ of $\mathbf{y} = (6, 2, 0, 5)$ under the map $\mathcal{Q}$
- Choose random values for the Vinegar variables x_1 and x_2, e.g. $(x_1, x_2) = (0, 1)$ and substitute them into the polynomials $q^{(3)}, \ldots, q^{(6)}$.

$$\begin{aligned}
\tilde{q}^{(3)} &= 5x_3 + 6x_4 + 2, \tilde{q}^{(4)} = x_3 + 5x_4,\\
\tilde{q}^{(5)} &= 3x_3x_4 + x_3x_5 + 2x_3x_6 + 3x_3 + 3x_4x_5 + 4x_4 + 2x_5 + 2x_6,\\
\tilde{q}^{(6)} &= 3x_3^2 + 5x_3x_4 + 4x_3x_5 + 2x_3x_6 + x_4^2 + 6x_4x_5 + 3x_4x_6 + 4x_4 + 2x_5 + 5x_6 + 1.
\end{aligned}$$

- Set $\tilde{q}^{(3)} = y_1 = 6$ and $\tilde{q}^{(4)} = y_2 = 2$ and solve for $x_3, x_4 \Rightarrow (x_3, x_4) = (3, 4)$
- Substitute into $\tilde{q}^{(5)}$ and $\tilde{q}^{(6)} \Rightarrow \tilde{\tilde{q}}^{(5)} = 3x_5 + x_6 + 5, \tilde{\tilde{q}}^{(6)} = 3x_5 + 2x_6 + 1$
- Set $\tilde{\tilde{q}}^{(5)} = y_3 = 0$ and $\tilde{\tilde{q}}^{(6)} = y_4 = 5$, solve for x_5 and $x_6 \Rightarrow (x_5, x_6) = (0, 2)$

A pre image of $\mathbf{y} = (6, 2, 0, 5)$ is given by $\mathbf{x} = (0, 1, 3, 4, 0, 2)$.

B Conversion to bitsliced representation

Algorithm 7 Conversion of $\mathbb{F}_{16}$ elements from normal to bitsliced representation and vice versa.

Input: 32 $\mathbb{F}_{16}$ elements in a_0, a_1, a_2, a_3

Output: Bitsliced $\mathbb{F}_{16}$ elements in b_0 (LSB), b_1, b_2, b_3 (MSB)

```
and $b_0$, $a_0$, #0x11111111
and $t$, $a_1$, #0x11111111
orr $b_0$, $b_0$, $t$, lsl#1
and $t$, $a_2$, #0x11111111
orr $b_0$, $b_0$, $t$, lsl#2
and $t$, $a_3$, #0x11111111
orr $b_0$, $b_0$, $t$, lsl#3

and $b_1$, $a_1$, #0x22222222
and $t$, $a_0$, #0x22222222
orr $b_1$, $b_1$, $t$, lsr#1
and $t$, $a_2$, #0x22222222
orr $b_1$, $b_1$, $t$, lsl#1
and $t$, $a_3$, #0x222222221
orr $b_1$, $b_1$, $t$, lsl#2

and $b_2$, $a_2$, #0x44444444
and $t$, $a_0$, #0x44444444
orr $b_2$, $b_2$, $t$, lsr#2
and $t$, $a_1$, #0x44444444
orr $b_2$, $b_2$, $t$, lsr#1
and $t$, $a_3$, #0x44444444
orr $b_2$, $b_2$, $t$, lsl#1

and $b_3$, $a_3$, #0x88888888
and $t$, $a_0$, #0x88888888
orr $b_3$, $b_3$, $t$, lsr#3
and $t$, $a_1$, #0x88888888
orr $b_3$, $b_3$, $t$, lsr#2
and $t$, $a_2$, #0x88888888
orr $b_3$, $b_3$, $t$, lsr#1
```