# Oil and Vinegar: Modern Parameters and Implementations

Ward Beullens[1], Ming-Shing Chen[2], Shih-Hao Hung[3], Matthias J. Kannwischer[2], Bo-Yuan Peng[2,3], Cheng-Jhih Shih[3] and Bo-Yin Yang[2]

[1] IBM Research Zurich, Switzerland

[2] Academia Sinica, Taipei, Taiwan

[3] National Taiwan University, Taipei, Taiwan

**Abstract.** Two multivariate digital signature schemes, Rainbow and GeMSS, made it into the third round of the NIST PQC competition. However, neither made its way to being a standard due to devastating attacks (in one case by Beullens, the other by Tao, Petzoldt, and Ding). How should multivariate cryptography recover from this blow? We propose that, rather than trying to fix Rainbow and HFEv- by introducing countermeasures, the better approach is to return to the classical Oil and Vinegar scheme. We show that, if parametrized appropriately, Oil and Vinegar still provides competitive performance compared to the new NIST standards by most measures (except for key size). At NIST security level 1, this results in either 128-byte signatures with 44 kB public keys or 96-byte signatures with 67 kB public keys. We revamp the state-of-the-art of Oil and Vinegar implementations for the Intel/AMD AVX2, the Arm Cortex-M4 microprocessor, the Xilinx Artix-7 FPGA, and the Armv8-A microarchitecture with the Neon vector instructions set.

**Keywords:** Oil and Vinegar, Intel AVX2, Arm Neon, Arm Cortex-M4, Xilinx Artix-7

## 1 Introduction

The Oil and Vinegar (OV) signature scheme was invented by Patarin in 1997 [43]. It was inspired by the famous bilinearization attack against $C^*$. Because OV parameters were changed [33] in response to the attack in [34] to make the Vinegar subspace larger than the Oil subspace, it is sometimes referred to as "Unbalanced Oil and Vinegar" in the literature. OV has a large public key. This is partially mitigated by the compressed key generation method introduced in [44] and [45] which compressed the public key by around an order of magnitude. With this modification, OV by itself becomes usable. OV is mostly useful for cases where very small signatures and/or fast verification are crucial, but large public keys are tolerable. For example, root and intermediate certificates in public-key infrastructures can benefit from OV as public keys have to be transmitted rarely (e.g., embedded into the OS or browser) while the signatures have to be transmitted for each connection to a new host. Another promising application are certificate transparency logs that have to store a large number signatures and only a very limited number of signers exist.

Despite this, people still tried out many OV variants, aiming to do better than plain OV in terms of speed, key size, or both. The best-known of these variants is multi-layered OV, or Rainbow [23], which was one of the three digital signature finalists in the NIST PQC competition. Unfortunately, almost all of these descendant constructions have pre-deceased OV itself, including Rainbow whose chosen parameters were broken by a devastating attack in early 2022 [12]. After [49] broke HFEv-, the venerable OV is the sole survivor of the earlier generations of multivariate cryptography.

While NIST has selected Dilithium, Falcon, and SPHINCS+ as winners from its post-quantum competition for the category of digital signatures [3], NIST has also made a supplementary call for digital signatures in which they asked for additional constructions (preferably not based on lattices) [42]. The long history of OV inspires confidence in its security, so it is natural to consider Oil and Vinegar in this context. In this work, we study how well OV performs compared to the other candidates and (soon-to-be) standards.

## 1.1 Our Contributions

Firstly, we propose modern instantiations of the OV signature scheme and present a justification for the proposed parameter sets. Secondly, we have compiled a set of high-performance constant-time OV implementations, using all known techniques from the multivariate cryptography literature. These comprise implementations for the Intel Haswell/AVX2, the Arm Cortex-M4 microprocessor, the Xilinx Artix-7 FPGA, and the Armv8-A microarchitecture with the Neon vector instructions set. Note that the first three platforms were specifically targeted by NIST as reference comparisons [4][1] while most handheld devices plus all newer Apple computers use Arm CPUs with Neon. Since NIST never specified an Armv8-A standard platform to benchmark on, we are using the popular, if somewhat dated, Raspberry Pi4b which uses an Arm Cortex-A72.

To the best of our knowledge, we integrated all the best techniques for optimizing OV in all categories, and we benchmarked against existing implementations where available, concluding that OV is competitive by all measures except public key size. For all software platforms, we show that blocked inversion as proposed for OV by Shim, Lee, and Koo [48] is inferior to Gaussian elimination by a large margin.

For each of these platforms, our implementations include novel tricks not yet described in the literature:

**Intel AVX2.** We present new techniques for generating multiplication tables, which are critical intermediate steps for constant-time SIMD multiplication in AVX2 implementations.

**Arm Neon.** We present Neon implementation for vector-vector field multiplication. Based on the vector-vector multiplication and a "lazy reduction" technique, we achieve a faster matrix-vector multiplication than the previous vector-scalar-based implementations.

**Arm Cortex-M4.** We introduce novel bit-sliced and byte-sliced $\mathbb{F}_{256}$ arithmetic with the latter outperforming the former. We present memory-efficient verification for compressed OV public keys using "lazy sampling" of the public key.

**Artix-7 FPGA.** We designed a coprocessor with a customized instruction set, with which we constructed the function of key generation, signing, and signature verification schemes. The coprocessor was tested on two different Xilinx Artix-7 FPGAs and the performance with some of the selected parameters are given, showing the power and the limits of Artix-7 FPGAs on the implementation of OV.

**Source code.** The source code for all our implementations is available under CC0

[1] While most papers today focus on the "Skylake" or later Intel models, NIST had actually specified the Intel "Haswell" platform, which is quite old, for baseline comparisons. We present results for both Skylake and Haswell.

copyright-waiver at https://github.com/pqov/pqov-paper. The code passes constant-time and memory leakage tests using `Valgrind`.[2]

### 1.2 Related Work

The most relevant related literature deal with Rainbow, for which the most up-to-date description (NIST round 3) can be found in [22]. Much code and many ideas from Rainbow over the years can be recycled for OV. For example, many multivariate techniques were summarized in [18, 48, 20], while one main attack against OV, the Intersection attack for OV can be found given in [11].

**Side-channel attacks.** In this work we focus on constant-time implementations achieving high-performance while we consider more advanced side-channel attacks (e.g., power-analysis attacks) out of scope. However, side-channel attacks against our implementations and defenses thereof have recently been studied by Aulbach, Campos, Krämer, Samardjiska, and Stöttinger [6]. We refer to [6, Sec. 5] for an overview on protecting UOV against side-channel attacks. The authors conclude that OV signing is relatively easy to protect against first-order attacks using multiplicative masking of the vinegar variables. Additionally, shuffling can be implemented cheaply to further reduce the signal-to-noise ratio and prevent profiled attacks such as the one proposed in [6]. Finally, as noted in [48] it may be useful to generate vinegar variables message-independent as their insertion into the central map then may be part of a pre-computation step disabling attacks on that computation. We leave the study of fully-protected OV implementations to future work.

### 1.3 Organization of this paper

We introduce Oil and Vinegar in Section 2, explaining how we selected our parameters. Section 3 describes some implementation techniques common to our software implementations. Sections 4–7 describes in order, implementations for AVX2, Armv8-A Neon, Arm Cortex-M4, and Artix-7 FPGA.

## 2 Oil and Vinegar

The Oil and Vinegar signature scheme is based on a trapdoored multivariate quadratic map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$. The trapdoor is an $m$-dimensional subspace $O \subset \mathbb{F}_q^n$ on which $\mathcal{P}$ vanishes, i.e., $\mathcal{P}(\mathbf{o}) = 0$ for all $\mathbf{o} \in O$.

Given $O$ and $\mathbf{y}$, one can efficiently sample preimages $\mathbf{x}$ such that $\mathcal{P}(\mathbf{x}) = \mathbf{y}$, by first sampling $\mathbf{v} \in \mathbb{F}_q^n$, and then solving for a vector $\mathbf{o} \in O$ such that $\mathcal{P}(\mathbf{v} + \mathbf{o}) = \mathbf{y}$. For any quadratic map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ and any $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{F}_q^n$ we have $\mathcal{P}(\mathbf{x}_1 + \mathbf{x}_2) = \mathcal{P}(\mathbf{x}_1) + \mathcal{P}(\mathbf{x}_2) + \mathcal{P}'(\mathbf{x}_1, \mathbf{x}_2)$, for some bilinear map $\mathcal{P}' : \mathbb{F}_q^n \times \mathbb{F}_q^n \to \mathbb{F}_q^m$, which is called the differential of $\mathcal{P}$. Therefore $\mathcal{P}(\mathbf{v} + \mathbf{o}) = \mathbf{y}$ simplifies to

$$\mathcal{P}'(\mathbf{v}, \mathbf{o}) + \underbrace{\cancel{\mathcal{P}(\mathbf{o})}}_{\mathcal{P} \text{ vanishes on } O} = \mathbf{y} - \mathcal{P}(\mathbf{v}),$$

which is a system of linear equations in $\mathbf{o}$, so it can be solved efficiently. We now describe the key generation, signing, and verification algorithms in more detail.

**Key generation.** To allow for a simple implementation, we make the restriction that the trapdoor space $O$ (the secret key) has a basis given by the rows of the matrix $(\mathbf{O}^\mathsf{T} \ \mathbf{1}_m)$, where $\mathbf{O} \in \mathbb{F}_q^{(n-m)\times m}$. We derive $\mathbf{O} = \mathsf{Expand}_{\mathsf{sk}}(\mathsf{seed}_{\mathsf{sk}})$ deterministically from a short seed $\mathsf{seed}_{\mathsf{sk}}$ of length sk_seed_len, chosen uniformly at random. Most spaces of dimension

[2] https://valgrind.org/

$m$ have a basis of this form, so this restriction does not decrease the key space much. To sample the trapdoor, one first samples $\mathbf{O}$, and then chooses at random a sequence of $m$ multivariate quadratic polynomials that vanish on the space spanned by rows of $(\mathbf{O}^\mathsf{T}\ \mathbf{1}_m)$. Each multivariate quadratic polynomial $p_i$ can be uniquely represented by an upper diagonal matrix

$$\mathbf{P}_i = \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix},$$

such that $p_i(\mathbf{x}) = \mathbf{x}^\mathsf{T}\mathbf{P}_i\mathbf{x}$. The quadratic polynomial $p_i$ vanishes on $O$ exactly if

$$\begin{pmatrix} \mathbf{O}^\mathsf{T} & \mathbf{1}_m \end{pmatrix} \mathbf{P}_i \begin{pmatrix} \mathbf{O} \\ \mathbf{1}_m \end{pmatrix} = \mathbf{O}^\mathsf{T}\mathbf{P}_i^{(1)}\mathbf{O} + \mathbf{O}^\mathsf{T}\mathbf{P}_i^{(2)} + \mathbf{P}_3^{(i)}$$

is skew-symmetric, so one can simply pick $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-m)\times(n-m)}$ (upper diagonal) and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-m)\times m}$ uniformly at random, and put

$$\mathbf{P}_i^{(3)} = \mathsf{Upper}(-\mathbf{O}^\mathsf{T}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^\mathsf{T}\mathbf{P}_i^{(2)}),$$

where $\mathsf{Upper}(\mathbf{M})$ is the unique upper diagonal matrix that is equal to $\mathbf{M}$ up to the addition of a skew-symmetric matrix. Since the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ are chosen uniformly at random, we expand them from a short seed $\mathsf{seed}_{\mathsf{pk}}$ of length $\mathsf{pk\_seed\_len}$, so that we have the option to expand $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ from the short seed instead of storing all their coefficients.

**Signing.** To sign a message $M \in \{0,1\}^*$, the signer hashes $M$, together with a salt $\mathsf{salt} \in \{0,1\}^{\mathsf{salt\_len}}$ (whose purpose is to protect against side-channel and fault injection attacks), to get a target vector $\mathbf{t} = \mathsf{Hash}(M||\mathsf{salt}) \in \mathbb{F}_q^m$. The signature comprises the $\mathsf{salt}$ and a preimage $\mathbf{s} \in \mathbb{F}_q^n$ for $\mathbf{t}$. To compute the preimage $\mathbf{s}$, the signer deterministically generates a so-called vinegar vector $\mathbf{v} = \mathsf{Expand}_{\mathbf{v}}(M||\mathsf{salt}||\mathsf{seed}_{\mathsf{sk}}||\mathsf{ctr}) \in \mathbb{F}_q^{n-m}$, where $\mathsf{ctr}$ is a byte-sized counter, initialized at `0x0`. Then the signer solves a system of linear equations to find a vector $\mathbf{x} \in \mathbb{F}_q^m$ such that $\mathbf{s} = (\mathbf{v} + \mathbf{O}\mathbf{x})||\mathbf{x} \in \mathbb{F}_q^n$ is the desired preimage for $\mathbf{t}$.

This system of linear equations is of the form $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$, where the vector $\mathbf{y}$ is the evaluation of $\mathcal{P}$ at $\mathbf{v}||\mathbf{0}_m$. More concretely, the $i$-th component of $\mathbf{y}$ is $y_i = \mathbf{v}^\mathsf{T}\mathbf{P}_i^{(1)}\mathbf{v}$. The $i$-th row of $\mathbf{L}$ is equal to $\mathbf{v}^\mathsf{T}\mathbf{S}_i$, where $\mathbf{S}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O} + \mathbf{P}_i^{(2)}$. The $\mathbf{S}_i$ matrices are relatively expensive to compute, but they are independent of the message we are signing, so we choose to compute them only once during key generation, and store them as part of the secret key.

The signer solves the linear system $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$ for $\mathbf{x}$ with a linear algebra method of choice, and then outputs the signature $(\mathsf{salt}, \mathbf{s})$, where $\mathbf{s} = (\mathbf{v} + \mathbf{O}\mathbf{x})||\mathbf{x}$. If the linear system is singular, then $\mathsf{ctr}$ is incremented by one, and the signing starts again with the new $\mathbf{v} = \mathsf{Expand}_{\mathbf{v}}(M||\mathsf{salt}||\mathsf{seed}_{\mathsf{sk}}||\mathsf{ctr})$. It is noteworthy that the signer does not have to keep secret how many attempts were required to find a solution. It is, thus, not an issue to leak (e.g., via timing side-channels) that the solving failed. After 256 failed attempts the signer aborts, but in honest executions this happens only with an extremely small probability ($\leq 2^{-786}$ for our parameters).

**Verification.** The verifier accepts if $\mathcal{P}(\mathbf{s}) = \mathsf{Hash}(M||\mathsf{salt})$. Concretely, he recomputes $\mathbf{t} = \mathsf{Hash}(M||\mathsf{salt})$, and checks that the $i$-th component of $\mathbf{t}$ is equal to

$$\mathbf{s}^\mathsf{T} \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix} \mathbf{s}, \tag{1}$$

for all $i$ from 1 to $m$. If the verifier has enough memory to store the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices he can keep them in memory, otherwise he can expand $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ from $\mathsf{seed}_{\mathsf{pk}}$ on the fly.

In any case the $\mathbf{P}_i^{(3)}$ matrices need to be stored in memory because they are not expanded from a seed.

**Key expansion algorithms.** The high-level structure of the key-generation, signing, and verification algorithms are given in Figure 1. There is some freedom to choose if the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices are communicated as part of the public key, or recomputed on the verifying device, and similarly, there is the freedom to either compute the $\mathbf{S}_i$ matrices during keygen or at signing time. Therefore, we split up the key generation algorithm into three parts KeyGen, ExpandPK, and ExpandSK. The KeyGen algorithm outputs (cpk, csk), compact representations of the public and secret keys. The ExpandSK and ExpandPK algorithms take csk and cpk as input respectively, and output expanded keys pk, and sk respectively, which can be used to run the signing and verification algorithms. This API gives the user freedom to run the key expansion algorithms where and when they want. For example, an application might communicate a compact cpk to the verifier, who expands it only once and verifies a large number of signatures with the expanded public key.

To compare Oil and Vinegar with other signature schemes, we need to fit the usual 3-part API for signature schemes. Therefore we define and implement three variants of the Oil and Vinegar scheme:

- `classic`: In this variant, the ExpandPK and ExpandSK operations are considered to be part of the KeyGen algorithm. This means the key sizes are larger, but signing and verification are faster.
- `pkc`: In this pk-compressed variant, ExpandSK is considered part of the KeyGen algorithm, but ExpandPK is considered part of the verification algorithm. This makes the public key much smaller (by a factor between 6 and 7), but makes verification slower.
- `pkc+skc`: In this doubly-compressed variant, ExpandSK is part of the Signing algorithm, and ExpandPK is part of the verification algorithm. Compared to the compressed pk variant the KeyGen algorithm is faster, and the secret key becomes tiny (only pk_seed_len + sk_seed_len bits), but the Signing algorithm becomes much slower.

Note that these are in line with previous variants of the Rainbow signature scheme (classic, circumzenithal, and compressed). However, we find our names more intuitive.

## Parameters and Security Analysis.

Table 1 contains the parameter sets we propose and implement. For NIST security level 1 we propose two parameter sets: `ov-Ip`, which works over $\mathbb{F}_{256}$ to get slightly smaller keys, and `ov-Is`, which works over $\mathbb{F}_{16}$, and has shorter signatures. For security levels 3 and 5, we propose one parameter set each. We use $\mathbb{F}_{16} := \mathbb{F}_2[x]/(x^4 + x + 1)$ and $\mathbb{F}_{256} := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$ as the field representations. One $\mathbb{F}_{256}$ element is stored in one byte as its coefficient array with the most significant bit corresponding to $x^7$. For $\mathbb{F}_{16}$, we pack two field elements into one byte with the first element in the least significant nibble. The most significant bit of each nibble corresponds to $x^3$. Regardless of the security level we use pk_seed_len $= 128$, sk_seed_len $= 256$, and salt_len $= 128$. In our implementation, we use randomly chosen 128-bit salts, whose size is included in the signature sizes we report.

Table 2 contains lower bounds for the bit-complexity of the state-of-the-art attacks against UOV. In the remainder of this section, we discuss the state-of-the-art attacks and we clarify how the complexities in Table 2 are obtained.

**KeyGen():**

$\mathsf{seed}_{\mathsf{sk}} \leftarrow \{0,1\}^{\mathsf{sk\_seed\_len}}$
$\mathsf{seed}_{\mathsf{pk}} \leftarrow \{0,1\}^{\mathsf{pk\_seed\_len}}$
$\mathbf{O} \leftarrow \mathsf{Expand}_{\mathsf{sk}}(\mathsf{seed}_{\mathsf{sk}})$ ▷ $\mathbf{O} \in \mathbb{F}_q^{(n-m)\times m}$
$\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$ ▷ $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-m)\times(n-m)}$ upper triangular
**for** $i$ from 1 to $m$ **do** ▷ $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-m)\times m}$
  $\mathbf{P}_i^{(3)} \leftarrow \mathsf{Upper}(-\mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(2)})$
$\mathsf{pk} \leftarrow (\mathsf{seed}_{\mathsf{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in\{i,\ldots,m\}})$
**return** $(\mathsf{cpk}, \mathsf{csk} = (\mathsf{seed}_{\mathsf{pk}}, \mathsf{seed}_{\mathsf{sk}}))$.

**ExpandSK**$(\mathsf{csk} = (\mathsf{seed}_{\mathsf{sk}}, \mathsf{seed}_{\mathsf{pk}}))$**:**

$\mathbf{O} \leftarrow \mathsf{Expand}_{\mathsf{sk}}(\mathsf{seed}_{\mathsf{sk}})$
$\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$
**for** $i$ from 1 to $m$ **do**
  $\mathbf{S}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O} + \mathbf{P}_i^{(2)}$
**return** $\mathsf{sk} = (\mathsf{seed}_{\mathsf{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]})$.

**Sign**$(\mathsf{sk}, M, \mathsf{salt})$**:**

$\mathsf{seed}_{\mathsf{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]} \leftarrow \mathsf{sk}$
$\mathbf{t} \leftarrow \mathsf{Hash}(M||\mathsf{salt})$ ▷ $\mathbf{t} \in \mathbb{F}_q^m$.
**for** $\mathsf{ctr}$ from 0 to 255 **do**
  $\mathbf{v} \leftarrow \mathsf{Expand}_{\mathbf{v}}(M||\mathsf{salt}||\mathsf{seed}_{\mathsf{sk}}||\mathsf{ctr})$ ▷ $\mathbf{v} \in \mathbb{F}_q^{n-m}$.
  $\mathbf{L} \leftarrow \mathbf{0}_{m\times m}$
  **for** $i$ from 1 to $m$ **do**
    Set $i$-th row of $\mathbf{L}$ to $\mathbf{v}^{\mathsf{T}}\mathbf{S}_i$.
  **if** $\mathbf{L}$ is invertible **then**
    $\mathbf{y} = \{\mathbf{v}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{v}\}_{i\in[m]}$
    Solve $\mathbf{Lx} = \mathbf{t} - \mathbf{y}$ for $\mathbf{x}$
    $\mathbf{s} \leftarrow (\mathbf{v} + \mathbf{Ox})||\mathbf{x}$ ▷ $\mathbf{s} \in \mathbb{F}_q^n$.
    **return** $\mathbf{s}$
**return Fail**

**ExpandPK**$(\mathsf{cpk})$**:**

$\mathsf{seed}_{\mathsf{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]} \leftarrow \mathsf{pk}$
$\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} \leftarrow \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$
**for** $i$ from 1 to $m$ **do**
  $\mathbf{P}_i = \begin{pmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ 0 & \mathbf{P}_i^{(3)} \end{pmatrix}$
**return** $\mathsf{pk} = \{\mathbf{P}_i\}_{i\in[m]}$.

**Verify**$(\mathsf{pk}, M, \mathbf{s}, \mathsf{salt})$**:**

$\{\mathbf{P}_i\}_{i\in[m]} \leftarrow \mathsf{pk}$
$\mathbf{t} \leftarrow \mathsf{Hash}(M||\mathsf{salt})$
**return** accept if $\mathbf{s}^{\mathsf{T}}\mathbf{P}_i\mathbf{s} = t_i$ for all $i \in [m]$.

Figure 1: The key generation, signing, and verification algorithms of the Oil and Vinegar signature scheme.

Table 1: Parameter sets and corresponding key and signature sizes for the Oil and Vinegar signature scheme. The size of the compressed secret key csk is 48 bytes for all parameter sets.

| | NIST SL | $n$ | $m$ | $\mathbb{F}_q$ | $\|$pk$\|$ (bytes) | $\|$sk$\|$ (bytes) | $\|$cpk$\|$ (bytes) | $\|$sig+salt$\|$ (bytes) |
|---|---|---|---|---|---|---|---|---|
| ov-Ip | 1 | 112 | 44 | $\mathbb{F}_{256}$ | 278 432 | 237 896 | 43 576 | 128 |
| ov-Is | 1 | 160 | 64 | $\mathbb{F}_{16}$ | 412 160 | 348 704 | 66 576 | 96 |
| ov-III | 3 | 184 | 72 | $\mathbb{F}_{256}$ | 1 225 440 | 1 044 320 | 189 232 | 200 |
| ov-V | 5 | 244 | 96 | $\mathbb{F}_{256}$ | 2 869 440 | 2 436 704 | 446 992 | 260 |

Table 2: Bit-complexity estimates (lower bound for the base-2 logarithm of the number of binary gates required to perform an attack) of state-of-the-art attacks against our proposed parameter sets. The KS and Intersection attacks are key-recovery attacks, and the claw-finding and Direct attacks are universal forgery attacks.

| Parameter set | Claw-Finding | Direct | | KS | Intersection | |
|---|---|---|---|---|---|---|
| $(n, m, q)$ | $\log_2$ | $k$ | $\log_2$ | $\log_2$ | $k$ | $\log_2$ |
| ov-Ip $(112, 44, 256)$ | 192 | 2 | **145** | 218 | 2 | 166 |
| ov-Is $(160, 64, 16)$ | **143** | 12 | 165 | 154 | 3 | 176 |
| ov-III $(184, 72, 256)$ | 304 | 4 | **218** | 348 | 2 | 250 |
| ov-V $(244, 96, 256)$ | 400 | 6 | **278** | 445 | 2 | 312 |

**Claw-finding attack.** The first attack we consider is a simple claw-finding attack on $\mathcal{P}(\mathbf{s}) = \mathsf{Hash}(M||\mathsf{salt})$. An attacker can compute $\mathcal{P}(\mathbf{s}_i)$ for $X$ inputs $\{\mathbf{s}_i\}_{i\in[X]}$ and compute $\mathsf{Hash}(M||\mathsf{salt}_j)$ for $Y$ salts $\{\mathsf{salt}_j\}_{j\in[Y]}$. If $XY = q^m$, then there is a collision $\mathcal{P}(\mathbf{s}_i) = \mathsf{Hash}(M||\mathsf{salt}_j)$ with probability $\approx 1 - e^{-1}$, and the attacker can output the signature $(\mathsf{salt}_j, \mathbf{s}_i)$ for the message $M$. For the sake of concreteness, we estimate for $r \in \{4, 8\}$, that the cost of multiplication in $\mathbb{F}_{2^r}$ is $2r^2$ bit operations, and that of addition is $r$ bit operations, and that the cost of a Kecack-f 1600 permutation is $2^{17}$ bit operations. The bit-cost of the attack is then

$$Xm(2r^2 + r) + Y2^{17}\,,$$

which is equal to $2\sqrt{q^m m(2r + r)2^{17}}$ for optimally chosen $X, Y$ such that $XY = q^m$. This is the formula we use in Table 2. We have used gray-code enumeration [14] to evaluate $\mathcal{P}$ at $X$ inputs. Realistically, an attacker would use a memoryless claw finding algorithm [51], where it might not be possible to take full advantage of gray-code enumeration.

**Direct attack.** In this attack, the attacker computes $\mathsf{Hash}(M||\mathsf{salt})$, and then use system-solving techniques to solve for $\mathbf{s}$ such that $\mathcal{P}(\mathbf{s}) = \mathsf{Hash}(M||\mathsf{salt})$. A priori, the attacker might compute $\mathsf{Hash}(M||\mathsf{salt})$ for a large number of salts, and then solve a multi-target version of the system-solving problem. But there are no known algorithms that can take advantage of a large number of targets (beyond the naive claw-finding algorithm from the previous section). So, we estimate the complexity of this attack as the complexity of solving a random system of $m$ quadratic equations in $n$ variables. The state-of-the-art approach is to first take advantage of the underdeterminedness of the system by reducing to the problem of solving a system of $m' = m - 1$ equations in $n' = m - 1$ variables with the approach of Thomae and Wolf [50], and then using the hybrid WiedemannXL algorithm to solve the new system. This has an estimated bit complexity of

$$\min_k q^k \cdot 3\binom{n' - k + d_{n'-k,m'}}{d_{n'-k,m'}}^2 \binom{n' - k + 2}{2}(2r^2 + r)\,,$$

where $d_{N,M}$ is the *operating degree* of XL, which is the first $d > 0$ such that the the coefficient of $t^d$ in the power series expansion of $(1-t^2)^M(1-t)^{-(N+1)}$ is non-positive.

**Kipnis-Shamir attack.** The Kipnis-Shamir attack [34] tries to recover the secret key $O$ from the public map $\mathcal{P}$. The attack was first proposed for the case $n = 2m$, where it runs in polynomial time. Later, it was generalized to $n > 2m$, and it runs in time $\mathcal{O}(q^{n-2m}n^4)$ according to the literature if $n$ is even or $q$ is odd. However, this is an overestimate of the cost of the attack. The cost of finding a single vector in $O$ is dominated by the cost of computing on average $q^{n-2m}$ characteristic polynomials of $n$-by-$n$ matrices, and solving the same number of linear systems in $n$ variables. This can be done in time $\mathcal{O}(q^{n-2m}n^{\omega}\log(n))$ field multiplications, where $\omega$ is the exponent of matrix multiplication. The $n^4$ factor in the literature was obtained by putting $\omega = 3$, and repeating the attack $m = O(n)$ times to get a basis for $O$. Repeating the attack is wasteful because once a first vector in $O$ is found, the other vectors in $O$ can be found more efficiently with other methods (e.g., see [11]). For Table 2, we estimate the bit complexity of the attack as

$$q^{n-2m}n^{2.8}(2r^2+r)\,,$$

where we have used $\omega = 2.8$ from Strassen's algorithm. We believe this is an underestimate of the cost of the attack for our proposed parameters.

**Intersection attack.** The intersection attack tries to simultaneously find $k$ vectors in $O$, by solving a system of quadratic equations for some vector in the intersection $\cap_{i=1}^{k} M_i O$, for some matrices $M_i$. The attack only works if the intersection is nonempty, which is guaranteed if $n < \frac{2k-1}{k-1}m$. For details, we refer to [11]. The cost of the attack is dominated by the cost of solving a random system of $M = \binom{k+1}{2}m - 2\binom{k}{2}$ equations in $N = kn - (2k-1)m$ variables. For the `ov-Ip` parameter set, we use $k = 3$, even though $n = \frac{2k-1}{k-1}m$. This means that the intersection is not guaranteed to be nontrivial, and the attack is likely to fail. However, one can check that for these parameters the intersection is non-trivial with probability $1/(q-1)$, so on average we only need to repeat the attack $q - 1 = 15$ times, which is still cheaper than running a single attack with $k = 2$.

**Quantum attacks.** All the known quantum attacks against Oil and Vinegar are obtained by speeding some part of a classical attack up with Grover's algorithm. Therefore, they outperform the classical attacks by at most a square root factor, and they do not threaten our security claims. Indeed, the NIST security levels 1,3, and 5 are defined with respect to the hardness of a key search against a block cipher such as the AES with $128, 192$, or 256-bit keys respectively. Grover speeds up a key search by almost a square root factor, so, for a quantum attack to break the NIST security targets it needs to improve on classical attacks by more than a square root factor, which is not possible by relying on Grover's algorithm alone. Collision attacks are not believed to benefit from quantum computing. The system-solving attacks benefit relatively little from Grover's algorithm, because only a small part of the cost comes from guessing some of the variables, and only this part can be sped up with Grover's algorithm. Almost all of the cost of the Kipnis-Shamir attack comes from guessing a good matrix, so here Grover can almost fully achieve a quadratic speedup. However, the Kipnis-Shamir attack is less efficient than a key search classically, and checking if a guess is correct has a higher circuit depth, so a Groverized Kipnis-Shamir attack is still more costly than a Groverized key search.

## Symmetric primitives

Throughout OV, we require various hash functions and pseudo-random functions. We instantiate the performance uncritical functions $\mathsf{Hash}$, $\mathsf{Expand}_{\mathbf{v}}$, and $\mathsf{Expand}_{\mathsf{sk}}$ processing either public or secret data using instances of `shake256` [41]. For the performance critical

$\mathsf{Expand_{pk}}$ processing only public inputs, we use `aes128` [40] as it results in much faster implementations.

**Hash($M$||salt) :** $\{0,1\}^* \times \{0,1\}^{128} \rightarrow \mathbb{F}_q^m$

Maps a message $M$ and a 16-byte salt to the target vector $\mathbf{t}$. The size of the target vector is $m \cdot \log_2 |\mathbb{F}_q|$ bits, i.e., 32, 44, 72, and 96 bytes for `ov-Is`, `ov-Ip`, `ov-III`, and `ov-V`, respectively. We implement it as `shake256(`$M$||salt`)`.

**Expand$_\mathbf{v}$($M$||salt||seed$_\mathsf{sk}$||ctr) :** $\{0,1\}^{\mathsf{sk\_seed\_len}} \times \{0,1\}^* \times \{0,1\}^{128} \times \{0,1\}^8 \rightarrow \mathbb{F}_q^{n-m}$

Samples a vinegar vector $\mathbf{v}$ given the message $M$, a 16 byte salt, the secret seed seed$_\mathsf{sk}$, and a 1-byte counter. The output size is $(n-m) \cdot \log_2 |\mathbb{F}_q|$ bits, i.e., 48, 68, 112, and 148 bytes for `ov-Is`, `ov-Ip`, `ov-III`, and `ov-V`, respectively. We implement it as `shake256(`$M$||salt||seed$_\mathsf{sk}$||ctr`)`.

**Expand$_\mathsf{sk}$(seed$_\mathsf{sk}$) :** $\{0,1\}^{\mathsf{sk\_seed\_len}} \rightarrow \mathbb{F}_q^{m\cdot(n-m)}$

Expands the secret key to the matrix $\mathbf{O}$. The output size is $(n-m)\cdot m \cdot \log_2 |\mathbb{F}_q|$ bits, i.e., 3 072 (`ov-Is`), 2 992 (`ov-Ip`), 8 064 (`ov-III`), 14 208 (`ov-V`) bytes. We sample the matrix in column-major order as it is required in key generation and signing. We implement it using `shake256(`seed$_\mathsf{sk}$`)`.

**Expand$_\mathbf{P}$(seed$_\mathsf{pk}$) :** $\{0,1\}^{\mathsf{pk\_seed\_len}} \rightarrow \mathbb{F}_q^{m\cdot((n-m)(n-m+1)/2+m\cdot(n-m))}$

Expands the 16 byte public seed to the matrices $\mathbf{P}^{(1)} = \{\mathbf{P}_i^{(1)}\}_{i\in[m]}$ and $\mathbf{P}^{(2)} = \{\mathbf{P}_i^{(2)}\}_{i\in[m]}$. We first sample the $\mathbf{P}^{(1)}$ matrices, and then the $\mathbf{P}^{(2)}$ matrices. The $m$ matrices are expanded in an interleaved fashion, in column-major order. That is, we start by sampling the (0,0) entry of $\mathbf{P}_1^{(1)}$, followed by the (0,0) entry of $\mathbf{P}_2^{(1)}$, etc. After sampling the (0,0) entry of the last matrix $\mathbf{P}_m^{(1)}$ we continue with the (1,0) entries, followed by the (1,1) entries and proceeding column by column, i.e., in lexicographic order. The size of $\mathbf{P}^{(1)}$ is $m \cdot \frac{(n-m)(n-m+1)}{2} \cdot \log_2 |\mathbb{F}_q|$ bits, i.e., 148 992 (`ov-Is`), 103 224 (`ov-Ip`), 455 616 (`ov-III`), 1 058 496 (`ov-V`) bytes. The size of $\mathbf{P}^{(2)}$ is $m \cdot m \cdot (n-m) \cdot \log_2 |\mathbb{F}_q|$ bits, i.e., 196 608 (`ov-Is`), 131 648 (`ov-Ip`), 580 608 (`ov-III`), 1 363 968 (`ov-V`) bytes. We choose $\mathsf{Expand}_\mathbf{P}$ as `aes128ctr` using seed$_\mathsf{pk}$ as the key and a zero nonce. If the `aes128ctr` API allows passing a custom counter value, this allows sampling at arbitrary output positions which is tremendously useful for memory-constrained devices. As the columns of the public key corresponding to zero variables in the signature are not required for verification, this also allows to omit the sampling of those columns altogether. This is particularly useful for $\mathbb{F}_{16}$ as approximately 1/16 of the variables in each signature are zero.

Note that we do not require $\mathsf{Expand}_\mathbf{P}$ to be a cryptographically secure stream cipher. We (optionally) propose to use `aes128ctr` reduced to 4 (instead of 10) rounds. 4-round `aes128` has been proven to have a maximal differential probability of $2^{-114}$ [32] which we deem sufficient for public-key expansion.

## 3 Implementation Techniques

In this section, we describe our implementation techniques that are shared among platforms for linear equation solving (Subsection 3.1) and verification (Subsection 3.2).

**Notation.** Our implementations represent $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ as binary polynomials packed into bytes as specified in Section 2. In this paper, however, we sometimes write the polynomials as decimal numbers for a more compact presentation. We use the straightforward conversion, i.e., 1 corresponds to 1, 2 corresponds to $x^1$, 4 corresponds to $x^2$, and so forth.

**Algorithm 1** Constant-time linear equation solving using matrix inversion

**Input:** Linear equation $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$
**Output:** Solution $\mathbf{x} \in \mathbb{F}^m$ or $\perp$

1: $\mathbf{L}' \leftarrow (\mathbf{L}|I_m) \in \mathbb{F}^{m \times 2m}$
2: **for** $i \leftarrow 0, \ldots, m-1$ **do**
3:   **for** $j \leftarrow i+1, \ldots, m-1$ **do**
4:     **if** $\mathbf{L}'_{i,i} = 0$ **then**
5:       **for** $k \leftarrow i, \ldots, 2m-1$ **do**
6:         $\mathbf{L}'_{i,k} \leftarrow \mathbf{L}'_{i,k} + \mathbf{L}'_{j,k}$
7:   **if** $\mathbf{L}'_{i,i} = 0$ **then return** $\perp$
8:   $p^{-1} \leftarrow \mathbf{L}'^{-1}_{i,i}$
9:   **for** $k \leftarrow i, \ldots, 2m-1$ **do**
10:     $\mathbf{L}'_{i,k} \leftarrow p^{-1} \cdot \mathbf{L}'_{i,k}$
11:   **for** $j \leftarrow 0, \ldots, m-1$ **do**
12:     **if** $j \neq i$ **then**
13:       **for** $k \leftarrow i, \ldots, 2m-1$ **do**
14:         $\mathbf{L}'_{j,k} \leftarrow \mathbf{L}'_{j,k} + \mathbf{L}'_{j,i} \cdot \mathbf{L}'_{i,k}$
15: $\mathbf{L}^{-1} \leftarrow$ right half of $\mathbf{L}'$
16: **return** $\mathbf{x} = \mathbf{L}^{-1}(\mathbf{t} - \mathbf{y})$

**Algorithm 2** Constant-time linear equation solving using Gaussian elimination directly

**Input:** Linear equation $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$
**Output:** Solution $\mathbf{x} \in \mathbb{F}^m$ or $\perp$

1: $\mathbf{L}' \leftarrow (\mathbf{L}|\mathbf{t} - \mathbf{y}) \in \mathbb{F}^{m \times (m+1)}$
2: **for** $i \leftarrow 0, \ldots, m-1$ **do**
3:   **for** $j \leftarrow i+1, \ldots, m-1$ **do**
4:     **if** $\mathbf{L}'_{i,i} = 0$ **then**
5:       **for** $k \leftarrow i, \ldots, m$ **do**
6:         $\mathbf{L}'_{i,k} \leftarrow \mathbf{L}'_{i,k} + \mathbf{L}'_{j,k}$
7:   **if** $\mathbf{L}'_{i,i} = 0$ **then return** $\perp$
8:   $p^{-1} \leftarrow \mathbf{L}'^{-1}_{i,i}$
9:   **for** $k \leftarrow i, \ldots, m$ **do**
10:     $\mathbf{L}'_{i,k} \leftarrow p^{-1} \cdot \mathbf{L}'_{i,k}$
11:   **for** $j \leftarrow 0, \ldots, m-1$ **do**
12:     **if** $j \neq i$ **then**
13:       **for** $k \leftarrow i, \ldots, m$ **do**
14:         $\mathbf{L}'_{j,k} \leftarrow \mathbf{L}'_{j,k} + \mathbf{L}'_{j,i} \cdot \mathbf{L}'_{i,k}$
15: **for** $i \leftarrow m-1, \ldots, 1$ **do**
16:   **for** $j \leftarrow 0, \ldots, i-1$ **do**
17:     $\mathbf{L}'_{j,m} \leftarrow \mathbf{L}'_{j,m} + \mathbf{L}'_{i,j}\mathbf{L}'_{i,m}$
18: **return** last column of $\mathbf{L}'$

## 3.1 Solving linear equations

OV signing requires solving the system of linear equations $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$ for the $m$ variables $\mathbf{x}$. It is commonly implemented in either of two ways: Either one directly computes the solution using constant-time Gaussian elimination, or one first computes the inverse of $\mathbf{L}$ and multiplies it by the right side of the equation. We outline both approaches in Algorithm 1 and Algorithm 2. Both algorithms proceed in a similar way: As the first step (line 3) in the outer loop, we conditionally add all following rows to make sure the pivoting element $\mathbf{L}'_{i,i}$ is non-zero. This has to be performed in constant time, i.e., the addition has to be performed for all following rows. In case it is still zero, we return $\perp$ (line 7) as the matrix is not invertible or the system of linear equations has no unique solution. Leaking that the matrix is not invertible via a timing side-channel is not an issue as the matrix is discarded anyway. Then, we invert the pivoting element (line 8) and multiply the current row by the inverse (line 9). We then add multiples of that row to the remainder of the matrix (line 11). In the case of matrix inversion, we take the right half of the resulting matrix $\mathbf{L}'$ and multiply it by the vector $\mathbf{t} - \mathbf{y}$ to obtain the solution $\mathbf{x}$ (line 16). In the case of solving the linear equations directly, we back-substitute the variables into the system of equations to obtain the solutions (line 15).

Previous work by Shim, Lee, and Koo [48] used the approach of Algorithm 1 for AVX2 implementations of Rainbow and OV as it can be significantly sped-up by using blocked matrix inversion which allows replacing a $m \times m$ matrix inversion by two $m/2 \times m/2$ matrix inversions, two matrix-matrix multiplications and various matrix-vector multiplications. While Rainbow explicitly computed the inverse matrix for signing in its specification [22, algorithm 7], however, Shim, Lee, and Koo did not study if the blocked matrix inversion can be outperformed by directly solving the system of equations (Algorithm 2). By counting the number of multiplications involved, we can estimate the approximate cost:

The inversion of a $m \times m$ matrix requires $3/2 \cdot m^3$ field multiplications, while a $m \times m$ matrix product requires $m^3$ multiplications. Hence, a blocked matrix inversion costs at least $2 \cdot (3/2 \cdot (m/2)^3) + 2 \cdot (m/2)^3 = 5/8m^3$ multiplications. On the other hand, solving a system of $m$ equations in $m$ variables directly costs $m^3/3$ multiplications. Hence, from the number of multiplications, it does not appear promising to use blocked matrix inversion. We will show in later sections that it indeed is not worthwhile for any of our software implementations.

**Reducing the number of conditional additions.** For both Algorithm 1 and Algorithm 2, we have to perform a large number of conditional additions in lines 3-6 to achieve constant-time behavior. In practice, most of these additions will not actually be performed as the pivoting element is already nonzero. We instead propose to limit the additions to a small number of rows. We propose to add at most 15 rows for $\mathbb{F}_{16}$ and at most 7 rows for $\mathbb{F}_{256}$. This results in a probability of at most $m \cdot 16^{-16} = 2^{-58}$ and $m \cdot 256^{-8} \leq 2^{-57.4}$ to wrongly abort for the $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ parameters respectively, which we deem sufficiently small.

## 3.2 Verification

For OV verification, we evaluate the public map (Equation 1) represented by a Macaulay matrix at the variables given by the signature $\mathbf{s}$ and verify that the output equals the hash of the message. Note that OV verification is exactly the same as the one of Rainbow and, thus, the same techniques apply. We make use of a technique first introduced by Chou, Kannwischer, and Yang [20]: Instead of multiplying the monomials $s_i s_j$ by the corresponding column of the Macaulay matrix and accumulating it into a single accumulator, we use multiple accumulators and do not perform any multiplication while passing through the matrix. At the end of verification, each accumulator is multiplied by the corresponding field element to obtain the final result. This allows for delaying all multiplications to the end and, hence, vastly reducing the number of required multiplications. This results in a substantial speed-up. In the case of $\mathbb{F}_{16}$, we use 15 accumulators: One for each possible value of $s_i s_j$ except for zero as those columns can be discarded straight away. In the case of $\mathbb{F}_{256}$, we use $2 \times 15$ accumulators: One set for the four least significant bits, and one set for the four most significant bits. Each column gets added to the corresponding accumulator of each set. By using different accumulators for the high and low bits, we keep the memory requirements for this approach reasonable while still vastly reducing the number of required costly field multiplications. Note that this approach results in signature-dependent memory access patterns which may be problematic in case signatures are secret and if the targeted device leaks memory addresses, e.g., through cache timing side channels. For the majority of cases, however, the signature is public and this approach should be used for signing speed.

**Skipping parts of the public key.** As already pointed out by Chou, Kannwischer, and Yang [20], the verification can be further speed-up by exploiting that in case a monomial $s_i s_j$ is zero, the corresponding columns in the Macaulay do not affect the result as they are multiplied by zero. We, hence, skip ahead in case either of the variables is zero. This is particularly significant when working with $\mathbb{F}_{16}$ as 1/16 of variables are expected to be zero, which means 31/256 of the products $s_i s_j$ is expected to be zero.

**"Lazy sampling".** When using compressed public keys, the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices are sampled pseudo-randomly from a public seed by computing $\mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$. Straightforward implementations first sample the entire pseudo-random part and then call the classic verification routine. However, if some variables in the signature are zero, then this is wasteful as some parts of the public key are multiplied by zero, i.e., not used. We can simply advance the state of the PRNG (through a function `prng_skip`) by increasing the counter of `aes128ctr` state. We refer to this technique as "lazy sampling". Note that this optimization is made possible by choosing a PRNG construction that allows sampling output at arbitrary positions. This was not possible with previous constructions, e.g., used within Rainbow which requires sampling all the output sequentially. It would also not be possible when using a sponge-based extendable-output function (XOF) like `shake256` which may have appeared to be a natural choice for seed expansion. "Lazy sampling" results in a significant speed-up especially for $\mathbb{F}_{16}$.

# 4 X86 AVX2 Implementation

In this section, we present our optimization for x86-64 platforms, which is designated as the reference platform in NIST PQC standardization [42]. More precisely, we focus on the optimization for the AVX2 instruction set, which is arguably the most useful instruction set for its availability on modern x86 platforms. While NIST is requiring code primarily for the Intel Haswell microarchitecture, we additionally study the Intel Skylake microarchitecture as it is easily available more than Haswell and results in better performance.

## 4.1 AVX2 Instruction Set

Advanced Vector Extensions (AVX) are instruction extensions to the x86 architecture and Advanced Vector Extension 2 (AVX2) is an AVX extensions that supports most integer operations with 256-bit vector registers. AVX2 was introduced in the Intel Haswell architecture in 2013 and is commonly supported in x86 CPUs today. Newer CPUs also support AVX-512 with 512-bit vector registers. However, as AVX2 is much more widely adopted, we focus on AVX2 implementations in this paper. AVX2 provides single-instruction-multiple-data (SIMD) instructions, which treats its 256-bit registers as vectors of 8-, 16-, 32-, or 64-bit vector elements and operates on the vector elements simultaneously. The available instructions implement most of the common logic, arithmetic, data movement, and memory access operations. There are 16 vector registers provided in AVX2.

By far the most relevant AVX2 instruction for OV implementation is `vpshufb`. It operates as

$$\texttt{vpshufb( ymm\_t , ymm\_i )} \rightarrow \texttt{ymm\_d}$$

where `ymm_t` and `ymm_i` are two 256-bit input registers and `ymm_d` represents its 256-bit destination registers. Among the two inputs, $\texttt{ymm\_t} = (t_0, \ldots, t_{15}, q_0, \ldots, q_{15})$ stores two 128-bit tables of 16 8-bit entries and $\texttt{ymm\_i} = (i_0, \ldots, i_{15}, j_0, \ldots, j_{15})$ stores 4-bit indices pointing to particular entries of the tables in `ymm_t`. Its output $\texttt{ymm\_d} = (t_{i_0}, \ldots, t_{i_{15}}, q_{j_0}, \ldots, q_{j_{15}})$ provides the results of 32 table lookup operations. When the indices are negative numbers, it sets zero to the results. The operation can also be seen as shuffling byte data in the `ymm_t`, as suggested by its name. Note that `vpshufb` executes in constant time.

## 4.2 Finite Field Arithmetics

In this section, we discuss AVX2 implementations of field multiplication on $\mathbb{F}_{16} := \mathbb{F}_2[x]/(x^4 + x + 1)$ and $\mathbb{F}_{256} := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$. Since AVX2 contains no instruction tailored to implementing binary field arithmetic, we resort to a table-based implementation using `vpshufb`. Note that with AVX-512, Intel introduced the Galois Field New Instructions (GF-NI) dedicated to 8-bit binary GF arithmetic. We expect these instructions to benefit OV implementations significantly in a similar way as shown in [24] for Rainbow implementations.

**Field multiplication with table lookup instructions.** Since the SSE instruction set (the predecessor of AVX), the field multiplication within multivariate cryptography heavily relies on table lookup operations as proposed in [16]. Although the vector width has grown to 256 bits with AVX2 instructions, the same techniques are used for field multiplication.

In AVX2 OV implementation, we rely heavily on the `vpshufb` instruction for performing vector-scalar multiplication, which multiplies a vector of field elements by a scalar. Specifically for $\mathbb{F}_{16}$, to multiply the vector $(a_0, a_1, \ldots, a_{31})$ by a scalar $b \in \mathbb{F}_{16}$, we prepare two copies of a pre-computed multiplication table $b \cdot (0, 1, \ldots, 15)$ in the register `tab_b` and load our data $(a_0, a_1, \ldots, a_{31})$ in `ymm_a`. Then we perform the 32 multiplications on $\mathbb{F}_{16}$ with one `vpshufb` operation

$$\texttt{vpshufb}(\texttt{tab\_b}, \texttt{ymm\_a}) \to b \cdot (a_0, a_1, \ldots, a_{31}) \ . \tag{2}$$

For $\mathbb{F}_{256}$ multiplication, we compute vector-scalar multiplication using 2 `vpshufb` instructions. Given a vector $\mathbf{a} = (a_0, a_1, \ldots, a_{31})$ to be multiplied by a scalar $b \in \mathbb{F}_{256}$, we first compute 2 intermediate vectors $\texttt{lownib}(\mathbf{a})$ and $\texttt{highnib}(\mathbf{a})$, where $\texttt{highnib}(\cdot)$ and $\texttt{lownib}(\cdot)$ refer to the higher 4 bits (higher degrees) and lower 4 bits, respectively. Those can be obtained using 2 `AND` with masks for fetching 4-bit data and one logic shift for shifting high degree bits to index range of `vpshufb`. Then we need 2 pre-computed multiplication tables $b \cdot (0, 1, \ldots, 15)$ and $b \cdot (0 \ll 4, 1 \ll 4, \ldots, 15 \ll 4)$ storing the products of all possible lower and higher 4-bit values in $\mathbb{F}_{256}$ multiplied by $b$. Again, we have two copies of the two 16-byte tables stored in two 256-bit registers `tab_bl` and `tab_bh`. We can produce the 32 products $(a_0, a_1, \ldots, a_{31}) \cdot b$ with two `vpshufb` operations

$$\texttt{vpshufb}(\texttt{tab\_bl}, \texttt{lownib}(\mathbf{a})) \oplus \texttt{vpshufb}(\texttt{tab\_bh}, \texttt{highnib}(\mathbf{a})) \ . \tag{3}$$

Preparing the multiplication tables becomes an important issue when applying the table lookup multiplication. When working on non-secret data, a typical implementation stores tables of all possible values in memory. When computing $(a_0, a_1, \ldots, a_{31}) \cdot b$, it loads the table of $b$ from memory indexed by the value of $b$. However, when $b$ is secret this approach cannot be used as it would result in a timing side-channel leakage through cache attacks. To solve the constant-time issue, Chen, Li, Peng, Yang, and Cheng [18] proposed to calculate the required tables on demand. They batched the computation of multiplication tables for multiple scalars. For example of $\mathbb{F}_{16}$, to compute 16 multiplication tables for 16 multiplicands $\mathbf{b} = (b_0, \ldots, b_{15}) \in \mathbb{F}_{16}^{16}$, They first calculated 16 vector-scalar multiplications with 16 *known* multiplicands, i.e., $0 \cdot \mathbf{b}$, $1 \cdot \mathbf{b}$, $2 \cdot \mathbf{b}$, $3 \cdot \mathbf{b}$, and so on. Then, they collected 16 multiplication tables of $\mathbf{b}$ by a $16 \times 16$ matrix transpose on the previous 16 products.

In the following, we present two new methods for generating multiplication tables. One for single multiplicands and the other for multiple multiplicands, which improves the method in [18].

**Fast generation of multiplication tables for individual elements.** Our methods rely on exploiting the basis elements of underlying fields, which are $\{1, 2, 4, 8\}$ in $\mathbb{F}_{16}$ and $\{1, 2, \ldots, 128\}$ in $\mathbb{F}_{256}$. To generate the multiplication table of one secret element, we conditionally accumulate pre-defined multiplication tables of basis values based on corresponding secret bits. Hence there are 4 and 8 conditional additions for the 4 and 8 basis values in $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ respectively.

Figure 2 shows the `C` code for generating the table of one element $\texttt{b} \in \mathbb{F}_{256}$. It fetches particular bits of `b` by performing an `AND` with the corresponding constants of basis values and gets masks by comparing the previous results with zero. It tests the bits on the original value and shifted value for using fewer basis constants and thus reducing the

```
static inline
__m256i gf256_generate_multab_avx2( uint8_t b ) {
    __m256i bb     = _mm256_set1_epi16( b );
    __m256i bb_sr1 = _mm256_srli_epi16( bb, 1 );              // shift right 1 bit
    return ( _multab_x1_x10  & _mm256_cmpgt_epi16(bb     & _const_1  , _const_0) )
         ^ ( _multab_x2_x20  & _mm256_cmpgt_epi16(bb_sr1 & _const_1  , _const_0) )
         ^ ( _multab_x4_x40  & _mm256_cmpgt_epi16(bb     & _const_4  , _const_0) )
         ^ ( _multab_x8_x80  & _mm256_cmpgt_epi16(bb_sr1 & _const_4  , _const_0) )
         ^ ( _multab_x10_x1b & _mm256_cmpgt_epi16(bb     & _const_16 , _const_0) )
         ^ ( _multab_x20_x36 & _mm256_cmpgt_epi16(bb_sr1 & _const_16 , _const_0) )
         ^ ( _multab_x40_x6c & _mm256_cmpgt_epi16(bb     & _const_64 , _const_0) )
         ^ ( _multab_x80_xd8 & _mm256_cmpgt_epi16(bb_sr1 & _const_64 , _const_0) );
}
```

Figure 2: Generating multiplication table for one element `b` $\in \mathbb{F}_{256}$ in `C` code with Intel intrinsics [29].

register pressure. By masking (`AND`) pre-defined multiplication tables of basis values with previous masks, it conditionally accumulates the contributions of particular bits into the result table. The main part of the function clearly computes 8 conditional additions. For generating the table of one multiplicand in $\mathbb{F}_{16}$, we have a similar code except for different pre-defined multiplication tables and only 4 conditional additions for its return value.

There are other efficient implementations of the conditional addition depending on the underlying hardware architecture. For example, on the Intel Skylake architecture, the 16-bit vector multiplication (`vpmullw`) has higher throughput than on the Haswell architecture. We achieve a faster implementation by replacing the compare-and-mask operations with multiplication by 1 or 0 (i.e., shifting the desired bit to the least significant bit).

**Fast batched generation of multiplication tables.** We achieve fewer instruction counts per element than the previous method for generating tables of many elements. We first multiply all elements by all possible *basis* values of the underlying field and then rearrange the products to the proper positions to obtain the multiplication tables. For example, when generating tables of a $\mathbb{F}_{16}$ vector $\mathbf{b} = (b_0, \ldots, b_{31})$, we first calculate their products with basis values $\{1, 2, 4, 8\}$, i.e., $1 \cdot \mathbf{b}$, $2 \cdot \mathbf{b}$, $4 \cdot \mathbf{b}$, and $8 \cdot \mathbf{b}$. For collecting the table of $b_0$, we broadcast the product of $1 \cdot b_0$ to positions $\{1, 3, 5, \ldots, 15\}$, which are the indices with value 1 at its first bit, resulting in $b_0 \cdot (0,1,0,1,\ 0,1,0,1,\ 0,1,0,1,\ 0,1,0,1)$. The product of $2 \cdot b_0$ is broadcasted to positions $\{2, 3, 6, 7, 10, 11, 14, 15\}$, which are the indices with value 1 at its second bit, resulting $2b_0 \cdot (0,0,1,1,\ 0,0,1,1,\ 0,0,1,1,\ 0,0,1,1)$. The broadcasted results for $4 \cdot b_0$ and $8 \cdot b_0$ are $4b_0 \cdot (0,0,0,0,\ 1,1,1,1,\ 0,0,0,0,\ 1,1,1,1)$ and $8b_0 \cdot (0,0,0,0,\ 0,0,0,0,\ 1,1,1,1,\ 1,1,1,1)$ respectively. Then, we sum up (`XOR`) the 4 broadcast results for the table $b_0 \cdot (0, 1, 2, 3, \ldots, 15)$.

Figure 3 shows the `C` code for generating multiplication tables of 16 $\mathbb{F}_{256}$ elements. Given a 128-bit input variable `a` storing the 16 $\mathbb{F}_{256}$ elements, it first computes 4 256-bit variables `a_x1_x10`, `a_x2_x20`, `a_x4_x40`, and `a_x8_x80` for the products of all basis values. Then, with the `vpshufb` instruction, it broadcasts the products to the proper positions. The multiplication tables for all 16 elements are generated in a loop. In our actual implementation, we compute 2 tables in one loop iteration by broadcasting the 0-th and 1-st positions for reducing the number of shift-right instructions as the comments in the figure. Compared to the method in [18], there are two differences. First, our method multiplies only 4 basis values instead of all possible 16 values in the multiplication tables. Second, we broadcast the product with `vpshufb` instruction avoiding the matrix transpose operation.

In summary, we spend 16 `AND`, 8 `vpcmpgtw`, and 7 `XOR` instructions for generating one multiplication table in Figure 2; and 4 `vpshufb`, 2 `vpsrldq`, and 3 `XOR` for one table on

```
static inline void gf256v_generate_16_multab_avx2( __m256i multabs[16] , __m128i a ) {
    __m256i aa  = _mm256_setr_m128i( a , a );
    __m256i a_l = aa & _const_0x0f;                                      // low  4 bits
    __m256i a_h = _mm256_srli_epi16(aa,4) & _const_0x0f;                 // high 4 bits
    __m256i a_x1_x10 = _mm256_shuffle_epi8( _multab_x01_x10 , a_l )      // vpshufb
                     ^_mm256_shuffle_epi8( _multab_x10_x1b , a_h );      // vpshufb, xor
    __m256i a_x2_x20 = _mm256_shuffle_epi8( _multab_x02_x20 , a_l )      // vpshufb
                     ^_mm256_shuffle_epi8( _multab_x20_x36 , a_h );      // vpshufb, xor
    __m256i a_x4_x40 = _mm256_shuffle_epi8( _multab_x04_x40 , a_l )      // vpshufb
                     ^_mm256_shuffle_epi8( _multab_x40_x6c , a_h );      // vpshufb, xor
    __m256i a_x8_x80 = _mm256_shuffle_epi8( _multab_x08_x80 , a_l )      // vpshufb
                     ^_mm256_shuffle_epi8( _multab_x80_xd8 , a_h );      // vpshufb, xor
    multabs[0] =  _mm256_shuffle_epi8(a_x1_x10,_broadcast_x1_0)          // vpshufb
                ^ _mm256_shuffle_epi8(a_x2_x20,_broadcast_x2_0)          // vpshufb, xor
                ^ _mm256_shuffle_epi8(a_x4_x40,_broadcast_x4_0)          // vpshufb, xor
                ^ _mm256_shuffle_epi8(a_x8_x80,_broadcast_x8_0);         // vpshufb, xor
    for(int i=1;i<16;i++) {          // a loop unrolling here can save the shift operations.
        a_x1_x10 = _mm256_srli_si256( a_x1_x10 , 1 );                  // shift right 1 byte
        a_x2_x20 = _mm256_srli_si256( a_x2_x20 , 1 );                  // shift right 1 byte
        a_x4_x40 = _mm256_srli_si256( a_x4_x40 , 1 );                  // shift right 1 byte
        a_x8_x80 = _mm256_srli_si256( a_x8_x80 , 1 );                  // shift right 1 byte
        multabs[i] =  _mm256_shuffle_epi8(a_x1_x10,_broadcast_x1_0)    // vpshufb
                    ^ _mm256_shuffle_epi8(a_x2_x20,_broadcast_x2_0)    // vpshufb, xor
                    ^ _mm256_shuffle_epi8(a_x4_x40,_broadcast_x4_0)    // vpshufb, xor
                    ^ _mm256_shuffle_epi8(a_x8_x80,_broadcast_x8_0);   // vpshufb, xor
    }
}
```

Figure 3: Generating multiplication tables for 16 elements in `C` code with Intel intrinsics [29].

average in Figure 3. From our benchmarking results on the Intel Haswell architecture, it costs on average 23.0 and 10.3 cycles for computing one multiplication table with Figure 2 and Figure 3 (with a loop unrolling by 2) respectively.

## 4.3 Data Alignment for Gaussian Elimination on SIMD Architectures

While implementing Gaussian elimination (Algorithm 2) on SIMD architectures, we perform row operations on matrices of dimension $m \times (m+1)$, usually resulting in row vectors of unfriendly lengths for SIMD architectures. In OV, the row vectors are of length 65 over $\mathbb{F}_{16}$ and 45, 73, and 97 over $\mathbb{F}_{256}$. In the case of $\mathbb{F}_{16}$, a 256-bit AVX2 register is capable of storing 64 elements. Thus vectors of 65 elements are stored and processed in 2 registers. A naive implementation would store the starting 64 elements in one register and 65-th element in another. Then it always operates on 2 registers while performing row operations.

Since the lengths of row vectors shorten during the elimination (see indices of loops at line 5, 9, and 13 in Algorithm 2), a better data alignment of vectors in SIMD registers improves the performance over the naive implementation. For storing a row vector of 65 $\mathbb{F}_{16}$ elements, we can store the first element in one register and the remaining 64 elements in a second register. Then we process a row vector of 2 registers only for eliminating the first column of matrices while performing row operations in Algorithm 2. After the first column, we always process vectors of one register for row operations. This saves roughly half of the operations compared to the naive approach. The same principle applies to all matrices of $\mathbb{F}_{256}$ in OV implementations. However, to save the cost for moving $\mathbb{F}_{16}$ data (4-bit units), we adopt the alignment of naive implementation for $\mathbb{F}_{16}$ matrices and swap the last column and first column after the first column is eliminated. This results in the same effect of processing vectors with fewer registers.

Table 3: Benchmarking results of AVX2 implementations. Numbers are median CPU cycles of 1000 executions.

| | Haswell | | | Skylake | | |
|---|---|---|---|---|---|---|
| | KeyGen | Sign | Verify | KeyGen | Sign | Verify |
| `ov-Ip` | 3 311 188 | 116 624 | 82 668 | 2 903 434 | 105 324 | 90 336 |
| `ov-Ip-pkc` | 3 393 872 | | 311 720 | 2 858 724 | | 224 006 |
| `ov-Ip-pkc+skc` | 3 287 336 | 2 251 440 | | 2 848 774 | 1 876 442 | |
| `ov-Is` | 4 945 376 | 123 376 | 60 832 | 4 332 050 | 109 314 | 58 274 |
| `ov-Is-pkc` | 5 002 756 | | 398 596 | 4 376 338 | | 276 520 |
| `ov-Is-pkc+skc` | 5 448 272 | 3 042 756 | | 4 450 838 | 2 473 254 | |
| Dilithium 2† [38] | 97 621* | 281 078* | 108 711* | 70 548 | 194 892 | 72 633 |
| Falcon-512 [46] | 19 189 801* | 792 360* | 103 281* | 26 604 000 | 948 132 | 81 036 |
| SPHINCS+‡ [28] | 1 334 220 | 33 651 546 | 2 150 290 | 1 510 712* | 50 084 397* | 2 254 495* |
| `ov-III` | 22 046 680 | 346 424 | 275 216 | 17 603 360 | 299 316 | 241 588 |
| `ov-III-pkc` | 22 389 144 | | 1 280 160 | 17 534 058 | | 917 402 |
| `ov-III-pkc+skc` | 21 779 704 | 11 381 092 | | 17 157 802 | 9 965 110 | |
| `ov-V` | 58 162 124 | 690 752 | 514 100 | 48 480 444 | 591 812 | 470 886 |
| `ov-V-pkc` | 57 315 504 | | 2 842 416 | 46 656 796 | | 2 032 992 |
| `ov-V-pkc+skc` | 57 306 980 | 26 021 784 | | 45 492 216 | 22 992 816 | |

† Security level II. ‡ Sphincs+-SHA2-128f-simple. * Numbers from SUPERCOP [8].

## 4.4 Symmetric Cryptography

For implementing the four symmetric primitives ($\mathsf{Hash}$, $\mathsf{Expand}_{\mathbf{v}}$, $\mathsf{Expand}_{\mathsf{sk}}$, and $\mathsf{Expand}_{\mathbf{P}}$), we call the OpenSSL library when relating to standard cryptographic primitives, e.g., `shake256` and `aes128`. For $\mathsf{Expand}_{\mathbf{P}}$ using round-reduced AES, we adapt the `aes128ctr` implementation in [25], which utilizes x86 AES instructions, to implement only 4 AES rounds.

## 4.5 Results

We benchmark our AVX2 optimization of OV on the Intel Haswell and the Intel Skylake architectures. The `C` source code is compiled with `clang` version `14.0.0-1ubuntu1` and the performance numbers are measured on Intel Xeon E3-1230L v3 1.80GHz (Haswell) and Intel Xeon CPU E3-1275 v5 3.60GHz (Skylake) with turbo boost and hyper-threading disabled.

Table 3 reports the performance of our AVX2 implementations and comparisons to other standard PQC schemes. In the table, we merge the numbers for **Sign()** from `classic` and `pkc` versions and **Verify()** from `pkc` and `pkc+skc` to indicate that they use the same implementations. Among all comparisons, Table 3 shows that 1) `ov-Ip` has the fastest signing while `ov-Is` signing is only 2% slower. 2) `ov-Is` has the fastest verification although its public key is larger than `ov-Ip`. This reflects the fact that `ov-Ip` uses more `XOR` operations for the 2 accumulators while evaluating $\mathbb{F}_{256}$ public polynomials (see Subsection 3.2). 3) For verification with compressed keys, the computation of $\mathsf{Expand}_{\mathbf{P}}$, i.e., `aes128ctr`, dominates the execution time, which can be seen by comparing with the results of 4-round AES in Table 4. The round-reduced AES improves the verification time by around 40%. 4) For signing with compressed secret keys, the main computation spends on expanding the compressed keys.

Table 4: Benchmarking results of AVX2 implementations using 4-round AES for public-key expansion. Numbers are median CPU cycles of 1000 executions.

| | Haswell | | | Skylake | | |
|---|---|---|---|---|---|---|
| | KeyGen | Sign | Verify | KeyGen | Sign | Verify |
| `ov-Ip-pkc` | 3 130 128 | 114 012 | 182 100 | 2 815 902 | 106 336 | 150 902 |
| `ov-Ip-pkc+skc` | 3 154 404 | 2 113 924 | | 2 861 082 | 1 818 690 | |
| `ov-Is-pkc` | 4 799 564 | 117 948 | 205 504 | 4 337 958 | 110 602 | 167 886 |
| `ov-Is-pkc+skc` | 4 810 612 | 2 755 060 | | 4 252 570 | 2 366 766 | |
| `ov-III-pkc` | 21 419 104 | 348 756 | 714 252 | 17 441 792 | 300 716 | 589 846 |
| `ov-III-pkc+skc` | 21 203 604 | 11 222 092 | | 16 909 288 | 9 603 518 | |
| `ov-V-pkc` | 55 983 388 | 723 628 | 1 516 652 | 45 508 552 | 624 774 | 1 268 998 |
| `ov-V-pkc+skc` | 56 136 556 | 24 824 672 | | 44 792 434 | 21 823 506 | |

# 5 Arm Neon Implementation

In this section, we present our optimization of OV for the Armv8-A architecture. We briefly introduce the Armv8-A architecture and highlight some useful instructions. Subsection 5.2 and Subsection 5.3 present our optimizations for field multiplication and matrix-vector multiplications, respectively. We conclude the section with our performance results in Subsection 5.5.

## 5.1 Neon Instruction Set

Armv8-A is a 64-bit Arm architecture that is part of Arm's application (A) profile targeting high-performance computing, for example, for PCs, smartphones, and servers. It is an important platform in addition to x86, yet attracting relatively few studies of NIST PQC candidates. For optimizing OV, we focus on the Advanced SIMD (Neon) instructions that are part of the Armv8-A architecture. While the name "Neon" and many of its functionalities are shared among many Arm architectures, we denote in this paper with "Neon" the Armv8-A (AArch64) instruction architecture specifically. We benchmark our Neon implementation on the Arm Cortex-A72 [37] processor, which is commonly available on a Raspberry Pi4b. We also provide benchmarks for the Apple M1.

Armv8-A Neon provides 32 vector registers of 128-bit each. Neon instruction interprets these as vectors of 8-, 16-, 32-, or 64-bit elements. As suggested in its name, these elements are processed in SIMD manners. Most commonly used instructions, such as memory access, logic, and arithmetics, can be processed in the Neon instruction set. Unlike AVX2, Neon provides dedicated instructions for (binary) polynomial multiplication which are useful for binary field multiplication. We list the heavily used instructions in our Neon implementation:

**`TBL/TBX`:** These table lookup instructions are similar to `vpshufb` on AVX2. However, they are capable of larger tables up to 64-byte entries (in 4 vector registers) which are indexed using 6-bit indices. When the indices are out of range, `TBL` set results to 0 and `TBX` keeps the destinations unchanged. In our implementation, we use `TBL` in almost the same way as the `vpshufb` instruction.

**`PMUL`:** The instruction multiplies 16 8-bit $\mathbb{F}_2$ polynomials, producing 16 *8-bit* products of $\mathbb{F}_2$ polynomials.

**`PMULL/PMULL2`:** These are the "widening" versions of the `PMUL` instruction. The `PMULL` instruction multiplies two $8 \times 8$-bit sources to one $8 \times 16$-bit destination. Since one 128-bit Neon register contains $16 \times 8$-bit data, the other instruction `PMULL2` performs the same computation as `PMULL` on the most significant $8 \times 8$-bit of the sources.

```
static inline
uint8x16_t _gf256v_reduce_neon( uint8x16_t ab0, uint8x16_t ab1 ) {
  uint8x16_t abh = vuzp2q_u8( ab0 , ab1 );                                   // UZP1
  uint8x16_t abl = vuzp1q_u8( ab0 , ab1 );                                   // UZP2
  return abl ^ vqtbl1q_u8( table_deg8to11  , abh & const_0x0f )        // EOR, TBL, AND
             ^ vqtbl1q_u8( table_deg12to15 , vshrq_n_u8( abh , 4 ));   // EOR, TBL, USHR
}
static inline
uint8x16_t _gf256v_mul_neon( uint8x16_t a , uint8x16_t b ) {
  uint8x16_t ab0 = vmull_p8( vget_low_u8(a) , vget_low_u8(b) );             // PMULL
  uint8x16_t ab1 = vmull_high_p8( a , b );                                  // PMULL2
  return _gf256v_reduce_neon( ab0 , ab1 );
}
```

Figure 4: $\mathbb{F}_{256}$ vector-vector multiplication in `C` code with Neon intrinsics [5].

## 5.2 Finite Field Arithmetics

Since `TBL` in Neon performs similar functionalities to `vpshufb` in AVX2, we immediately have a `TBL` based vector-scalar multiplication by replacing the `vpshufb` instructions with `TBL` in Equation 2 and Equation 3. We treat it as a baseline for our Neon OV implementation and discuss other optimizations based on Neon instructions.

In this section, we present a NEON implementation for vector-vector componentwise multiplication by applying polynomial multiplication instructions `PMULL` and `PMULL2`. It is a more general multiplication, since vector-scalar multiplication can be seen as a special case of vector-vector multiplication. With the vector-vector multiplication, we first develop a method for generating multiplication tables with less instruction counts than methods in Subsection 4.2. We also compare the efficiency between the `TBL` based vector-scalar and `PMULL` based vector-vector multiplications to conclude this section. The benchmark shows `PMULL` multiplication performs better than `TBL` multiplication for short vectors when the cost of generating multiplication is included. Based on the `PMULL` multiplication, we develop a fast matrix-vector multiplication in Subsection 5.3.

**Vector-vector componentwise multiplications.** For multiplication on $\mathbb{F}_{16} = \mathbb{F}_2[x]/(x^4 + x + 1)$, we use one `PMUL` to multiply degree-3 source polynomials for products of degree-6 $\mathbb{F}_2$ polynomials. A reduction step then reduces the degree-6 polynomials to degree-3. It first shifts (`USHR`) the parts of degree-4 to 6 to degree-0 to 2, multiplies the shifted parts by the polynomial $x + 1$ by another `PMUL`, and finally accumulates (`EOR`) the second products to the parts of degree-0 to 2 of the first products. The `PMUL` instruction in the reduction step can be replaced by a `TBL` instruction with an extra load operation for a "reduction" table.

Figure 4 implements vector-vector multiplication on $\mathbb{F}_{256}$ with Arm Neon intrinsics [5]. We apply one `PMULL` and one `PMULL2` to perform two $8 \times 8 \to 16$ multiplication on the sources of lower and higher 64-bit of Neon registers. They produce 16 16-bit products in two Neon registers. For reducing the 16-bit polynomials to 8-bit forms, we first apply one `UZP1` and one `UZP2` to split the 16-bit polynomials into two 8-bit polynomials in two registers, containing the parts of degree-0 to 8 and degree-8 to 15 respectively. We then apply two `TBL` instructions for reducing the polynomials of degree-8 to 11 and degree-12 to 15, respectively. The results are accumulated (`EOR`) into the register of degree-0 to 8 to finish the multiplication. Note that, after splitting products into two polynomials of high and low degrees, the remaining reduction step uses the same operations as Equation 3 except the tables are different.

Comparing to the vector-scalar multiplication described in Subsection 4.2, the vector-scalar costs fewer instructions for performing field multiplication if excluding the cost of

Table 5: Average CPU cycles of 1024 tests for $\mathbb{F}_{256}$ vector-scalar multiplication.

| | Vector length | 16 | 32 | 48 | 64 | 80 |
|---|---|---|---|---|---|---|
| Cortex-A72 | `TBL` (incl. generating multab) | 33.90 | 42.62 | **50.02** | **71.33** | **81.31** |
| | `PMULL` (Figure 4) | **19.25** | **35.49** | 76.33 | 79.15 | 97.78 |
| Apple M1 | `TBL` (incl. generating multab) | 5.01 | 6.32 | **7.57** | **9.01** | **10.06** |
| | `PMULL` (Figure 4) | **3.06** | **5.50** | 8.03 | 10.54 | 13.03 |

generating multiplication tables. It uses only one and two `TBL` instructions for multiplying vectors of $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$, respectively, which is roughly the same cost of the reduction operations in the vector-vector multiplication. To figure out the exact cost of multiplying longer vectors, we need to look into the cost of generating multiplication tables.

**Generating multiplication tables.** The vector-vector multiplication provides a simpler way to generate multiplication tables in contrast to methods in Figure 2 and Figure 3. For generating the multiplication table of a multiplicand $v \in \mathbb{F}_{16}$, we first duplicate $v$ into a vector $\mathbf{v} = (v, \ldots, v)$ in a Neon register and multiply $\mathbf{v}$ by a vector $\mathbf{c} = (0, 1, \ldots, 15)$ with the vector-vector multiplication. The products are the multiplication table of $v$.

For $v \in \mathbb{F}_{256}$, we need $v \cdot \mathbf{c}$ and $v \cdot (\mathbf{c} \cdot x^4)$ for the tables of multiplying low and high nibbles. To reduce the common computation while computing the 2 tables, we first divide $v$ into 2 4-bit nibbles $(v_l, v_h) = (\texttt{lownib}(v), \texttt{highnib}(v))$ and then compute $v_l \cdot \mathbf{c}$ and $v_h \cdot \mathbf{c}$ with 2 `PMUL` instructions. Then we have the first table $v \cdot \mathbf{c} = v_l \cdot \mathbf{c} + x^4 \cdot v_h \cdot \mathbf{c}$. To raise the degree by 4, we use a shift-left operation and one `TBL` for reducing the coefficients with degrees 8 to 11.

We compute the second table $x^4 \cdot v \cdot \mathbf{c}$ depending on architectures. In general, we raise the degree of the first table by 4 for the second table. However, since the latency of `TBL` instruction is $3 \times 1 + 3$ on Cortex-A72 [37, Page 30] (comparing to 2 on Apple M1 [30]), there will be a serious data hazard if we wait for the result of the first table on Cortex-A72. Thus, on Cortex-A72, we compute the second table $x^4 \cdot v \cdot \mathbf{c} = x^4 \cdot v_l \cdot \mathbf{c} + x^8 \cdot v_h \cdot \mathbf{c}$, where we reuse the `PMULL` results and raise their degrees by 4 and 8. Raising the degree by 8 is the same as the reduce operation in Figure 4.

Table 5 compares `TBL`-based and `PMULL`-based multiplications by measuring the average time for multiplying vectors of variable lengths by a scalar. For `TBL` multiplication, we include the time for generating multiplication tables of the scalar. The difference between adjacent lengths of vectors shows the actual cost of `TBL` multiplication, which is, for instance, $6.32 - 5.01 = 1.31$ for 16 multiplications on Apple M1. We can infer the cost of generating multiplication tables is $5.01 - 1.31 = 3.7$, which is slightly larger than the cost for one `PMULL` multiplication for 16 elements. Based on the results, we apply `PMULL` multiplication for vectors of length $\leq 32$ and `TBL` multiplication for longer vectors in our Neon optimization.

## 5.3 Matrix-Vector Multiplication

Considering $c = A \cdot b$ where $A \in \mathbb{F}_{256}^{m \times n}$, $b \in \mathbb{F}_{256}^{n}$ and $c \in \mathbb{F}_{256}^{m}$, we perform the computation as $c = \sum_{i=0}^{n-1} A_i \cdot b_i$ where $A_i \in \mathbb{F}_{256}^{m}$, which is $n$ accumulation of the results of vector-scalar multiplication $A_i \cdot b_i$. We have two options for the accumulation. First, we can apply `TBL` vector-scalar multiplication for computing $A_i \cdot b_i$. With this option, we omit the cost of generating multiplication tables of $n$ $b_i$s by assuming the multiplication tables are used many times, e.g., during signing when $b_i$ are the vinegar variables. The other option applies lazy reduction with `PMULL` multiplication, which accumulates the results of `PMULL`

Table 6: Average CPU cycles of 1024 tests for $\mathbb{F}_{256}$ matrix-vector multiplication.

| Matrix dimension | | 48 × 8 | 48 × 16 | 48 × 24 | 48 × 32 | 48 × 40 |
|---|---|---|---|---|---|---|
| Cortex-A72 | `TBL` | **182** | 358 | 530 | 702 | 894 |
| | `PMULL` w/ lazy reduction | 199 | **348** | **490** | **634** | **791** |
| Apple M1 | `TBL` | **155** | **197** | **236** | 266 | 296 |
| | `PMULL` w/ lazy reduction | 161 | 201 | 237 | **262** | **289** |

and `PMULL2` in Figure 4 and applies only one reduction after accumulating all intermediate results.

We compare the main computations of the accumulation operations in the two methods since the cost for generating a table of $b_i$ amortizes with growing $m$ for `TBL` multiplication and the cost for reduction operation amortizes with increasing $n$ for `PMULL` multiplication. `TBL` multiplication uses 6 instructions (2 `TBL`, 1 `AND`, 1 `USHR`, 2 `EOR`)[3]. On the other hand, `PMULL` multiplication uses 4 instructions (1 `PMULL`, 1 `PMULL2`, and 2 `EOR`)[4]. `PMULL` multiplication with lazy reduction uses fewer instructions for the main computation.

Table 6 compares the two methods for matrix-vector multiplication with various lengths of vectors. We choose 48 as the height of the matrix since 48 is the most common length for processing vectors in `ov-Ip`, where the size of oil variables is $m = 44$. The result shows `PMULL` multiplication with lazy reduction outperforms `TBL` vector-scalar multiplication when $n$ is larger than 16 and 32 on Cortex-A72 and Apple M1 respectively.

## 5.4 Symmetric Cryptography

For symmetric primitives relating to `shake256` function, i.e., Hash, $\text{Expand}_{\mathbf{v}}$, and $\text{Expand}_{\text{sk}}$, we call the OpenSSL library since it is generally available on most platforms.

We have two different Neon implementations for `aes128ctr` depending on the availability of Arm AES instructions. On platforms supporting AES instructions, e.g., Apple M1, we implement the standard and round reduced `aes128ctr` with AES instructions. On platforms without AES instructions, e.g., Raspberry Pi4b, we port the bitsliced implementation for 32-bit platforms in [2], which runs four parallelized 32-bit bitsliced instances, to the Neon instruction set, since Biesheuvel [13] reported bitsliced implementations outperform `TBL`-based implementations in the Linux kernel setting.

## 5.5 Results

We benchmark our Neon implementations of OV on Raspberry Pi4b and Apple's 2020 MacBook Air, both supporting 64-bit Armv8-A instruction set. The Raspberry Pi4b equips a Broadcom BCM2711 CPU (Arm Cortex-A72 CPU [37]) running at 1.8 GHz without Arm AES instructions. The source code is compiled with Debian `clang` version `version 11.0.1-2`. The Macbook has an Apple M1 CPU running at 3.2 GHz with Arm AES instruction support. Its compiler is Apple `clang` version `14.0.0 (clang-1400.0.29.202)`.

Table 7 reports the results of Neon OV implementation and comparison with other PQC signatures on the two Armv8-A platforms. Table 8 shows results with the 4-round AES option of $\text{Expand}_{\mathbf{P}}$. The results show that:

1) `ov-Ip` has the best signing time which is consistent with the results of AVX2 implementation (Table 3). However, `ov-Ip` outperforms `ov-Is` by a margin on

[3] On Apple M1, we use `EOR3` instruction from the SHA3 extension instead of 2 `EOR`.
[4] On Apple M1, `PMULL`+`EOR` costs the same as only `PMULL` [30].

NEON while, on AVX2, `ov-Ip` leads `ov-Is` by $< 10\%$. This is caused by the mismatch between the sizes of registers and vectors. When processing line 9 and 10 of **Sign()** in Figure 1, the vectors are of legnth 44 or 45 bytes for `ov-Ip`. These vectors are actually processed as $16 \times 3$ bytes on NEON but $32 \times 2$ bytes on AVX2 due to their 128-bit or 256-bit registers. It is clear that the AVX2 implementation wastes more computations than NEON.

2) For verification, due to the fewer accumulators on $\mathbb{F}_{16}$ (see Subsection 3.2), `ov-Is` outperforms `ov-Ip` although its larger size of public keys. On the other hand, the verification time is proportional to the public key sizes for the `pkc` and `pkc+skc` variants, where $\mathsf{Expand}_{\mathbf{P}}$ dominates the computation time.

3) For `pkc` and `pkc+skc` variants, the symmetric primitives play an important role in the performance. By comparing the performance impact of key compressed variants to the `classic` variant, the impact is significantly smaller on the Apple M1 than the Raspberry Pi4b, since the native AES (and SHA3) instructions on M1 result in faster symmetric primitives than the bit-sliced ones on the Raspberry Pi4b.

4) The 4-round AES makes for an efficient $\mathsf{Expand}_{\mathbf{P}}$ function such that the verification time of `pkc` variants is of the same order as other PQC schemes on Apple M1 CPU.

Table 7: Benchmarking results of our Neon implementations. Numbers are median CPU cycles of 10 000 executions.

<table>
<tr><th></th><th colspan="3">Cortex-A72</th><th colspan="3">Apple M1</th></tr>
<tr><th></th><th>KeyGen</th><th>Sign</th><th>Verify</th><th>KeyGen</th><th>Sign</th><th>Verify</th></tr>
<tr><td>ov-Ip</td><td>11 172 204</td><td rowspan="2">245 095</td><td>142 868</td><td>1 793 119</td><td rowspan="2">55 289</td><td>49 719</td></tr>
<tr><td>ov-Ip-pkc</td><td>11 193 794</td><td rowspan="2">3 677 844</td><td>1 775 826</td><td rowspan="2">112 934</td></tr>
<tr><td>ov-Ip-pkc+skc</td><td>11 229 231</td><td>7 617 137</td><td>1 774 748</td><td>1 056 617</td></tr>
<tr><td>ov-Is</td><td>29 269 925</td><td rowspan="2">460 655</td><td>141 528</td><td>3 391 967</td><td rowspan="2">74 633</td><td>45 908</td></tr>
<tr><td>ov-Is-pkc</td><td>28 906 183</td><td rowspan="2">5 070 253</td><td>3 360 648</td><td rowspan="2">138 496</td></tr>
<tr><td>ov-Is-pkc+skc</td><td>29 467 684</td><td>16 413 501</td><td>3 393 812</td><td>2 089 131</td></tr>
<tr><td>Dilithium 2† [7]</td><td>269 724</td><td>649 230</td><td>272 824</td><td>71 061</td><td>224 125</td><td>69 792</td></tr>
<tr><td>Falcon-512 [39]</td><td>—</td><td>1 044 600</td><td>59 900</td><td>—</td><td>459 200</td><td>22 700</td></tr>
<tr><td>ov-III</td><td>66 871 027</td><td rowspan="2">1 542 143</td><td>574 080</td><td>9 836 359</td><td rowspan="2">147 564</td><td>189 837</td></tr>
<tr><td>ov-III-pkc</td><td>66 554 826</td><td rowspan="2">17 161 246</td><td>9 803 637</td><td rowspan="2">461 896</td></tr>
<tr><td>ov-III-pkc+skc</td><td>64 147 364</td><td>42 794 977</td><td>9 751 198</td><td>6 353 401</td></tr>
<tr><td>ov-V</td><td>313 814 250</td><td rowspan="2">3 316 413</td><td>1 319 092</td><td>28 286 979</td><td rowspan="2">293 826</td><td>376 000</td></tr>
<tr><td>ov-V-pkc</td><td>305 700 907</td><td rowspan="2">39 337 795</td><td>26 743 866</td><td rowspan="2">1 011 331</td></tr>
<tr><td>ov-V-pkc+skc</td><td>312 729 427</td><td>107 305 680</td><td>26 663 940</td><td>15 830 169</td></tr>
</table>

† Security level II.

# 6 Arm Cortex-M4 Implementation

This section covers our implementations of OV for the Arm Cortex-M4. We base our implementation on the Rainbow implementation by Chou, Kannwischer, and Yang [20]. Subsection 6.1 introduces the features of the Arm Cortex-M4 which prove useful for OV implementations. Subsection 6.2 describes the characteristic two finite field multiplication which is used throughout the key generation, signing, and verification algorithms. Subsection 6.3 presents our implementations for solving linear equations which is essential for signing. Subsection 6.4 covers signature verification and its memory-efficient implementation on the Cortex-M4. In Subsection 6.6, we present the resulting performance of our

Table 8: Benchmarking results of Neon implementations using 4-round AES for public-key expansion. Numbers are median CPU cycles of 10 000 executions.

| | Cortex-A72 | | | Apple M1 | | |
|---|---|---|---|---|---|---|
| | KeyGen | Sign | Verify | KeyGen | Sign | Verify |
| ov-Ip-pkc | 9 191 247 | 249 910 | 1 672 544 | 1 746 623 | 55 175 | 83 021 |
| ov-Ip-pkc+skc | 9 473 513 | 5 627 393 | | 1 748 646 | 1 026 701 | |
| ov-Is-pkc | 25 698 880 | 448 188 | 2 266 233 | 3 324 331 | 74 503 | 97 325 |
| ov-Is-pkc+skc | 28 324 760 | 13 333 557 | | 3 349 000 | 2 045 042 | |
| ov-III-pkc | 56 890 636 | 1 569 429 | 8 318 527 | 9 640 984 | 147 524 | 330 463 |
| ov-III-pkc+skc | 56 815 652 | 34 533 235 | | 9 645 510 | 6 221 280 | |
| ov-V-pkc | 282 742 682 | 3 339 648 | 18 602 008 | 26 305 292 | 293 117 | 704 986 |
| ov-V-pkc+skc | 291 438 637 | 86 727 909 | | 26 298 657 | 15 522 513 | |

implementations. Due to the stack limitations of available Cortex-M4 cores, we restrict this section to the security level 1 parameter sets of OV, i.e., `ov-Is` and `ov-Ip`.

### 6.1 Armv7E-M Instruction Set and the Arm Cortex-M4

The Arm Cortex-M4 has been designated the primary microcontroller optimization target for the NIST PQC standardization project and Cortex-M4 implementations of post-quantum cryptography have received by far the most attention in the embedded cryptography literature. The Arm Cortex-M4 implements the Armv7E-M instruction set which provides several features proving useful for implementing binary finite field arithmetic:

**Floating-point registers.** Processors implementing the Armv7E-M architecture can optionally implement a single-precision floating-point unit. When available, the floating-point unit comes with 32 32-bit floating-point register `s0` to s31 that can be used for performing floating-point arithmetic. While the arithmetic is usually not useful for implementing cryptography, it is noteworthy that it is possible to move data between floating-registers and general-purpose registers using `vmov` in a single cycle per word. This is faster than spilling registers to memory which requires $n+1$ cycles for spilling $n$ words.

**Flexible second operand (barrel shifter).** A distinguishing feature of the Arm architecture is the flexible second operand which allows to shift or rotate the second operand of most data-processing (but not multiplication instructions) instructions. For example, `eor Rd, Rn, Rm, lsl#7` shifts `Rm` to the left by 7 bits before performing the `eor` operation with `Rn`. Using the barrel shifter on the Cortex-M4 does not increase the latency or throughput of instructions.

**Conditional execution.** Conditional execution allows to execute up to four Thumb instructions within a `IT` block conditionally on a flag. The `IT` instruction is used to encode the condition and the number of instructions in the "then"-branch and the "else" branch. For example,

```
tst r0, #1
itt ne
eorne r1, r1, r2
eorne r1, r1, r3
```

performs an `eor` of `r1, r2, r3` and conditionally writes it to `r1` if the least significant bit of `r0` is set. Note that each instruction within the `IT` block will take one clock cycle irrespective of conditions which makes the use of conditional execution suitable for constant-time code. The instruction sequence above will always take 4 clock cycles.

**Algorithm 3** $\mathbb{F}_{256} = \mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$ multiply-accumulate on 4 elements packed into one register. Bold instructions are only needed once in case more elements are multiplied by $b$. We unroll the loops in the actual code. If all inputs fit in registers, this code requires $44 + 24n$ clock cycles to process $n$ words ($4n$ field elements).

**Input:** First multiplicand $a$ (4 field elements packed into one register)
**Input:** Second multiplicand $b$ (1 field element in the least significant byte)
**Input:** `pconst` = 0x1b (corresponding to $x^8 + x^4 + x^3 + x + 1$); `mconst` = 0x01010101
**Input/Output:** accumulator $c$ (4 field elements packed into one register)

1: **vmov** $b'_0$, $b$ ▷ precomputation of $b' = b, bx, bx^2, ..., bx^7$ (36 cycles)
2: **for** bit k=1,...,7 **do**
3:     **and** $t_1$, **mconst**, $b$, **lsr#7** ▷ multiply by $x$ and reduce
4:     **eor** $b$, $b$, $t_1$, **lsl#7**
5:     **mul** $t_1$, $t_1$, **pconst**
6:     **eor** $b$, $t_1$, $b$, **lsl#1**
7:     **vmov** $b'_k$, $b$
8: **for** bit k=0,...,7 **do** ▷ multiplication c = c + ab (32 cycles)
9:     **vmov** $t_1$, $b'_k$
10:     `and` $t_0$, `mconst`, $a$, `lsr#`$k$
11:     `mul` $t_0$, $t_1$, $t_0$
12:     `eor` $c$, $c$, $t_0$

## 6.2 $\mathbb{F}_{256}$ and $\mathbb{F}_{16}$ Arithmetic

The basic core arithmetic operation within OV is finite field multiplication in the fields $\mathbb{F}_{16} = \mathbb{F}_2[x]/\langle x^4 + x + 1\rangle$ and $\mathbb{F}_{256} = \mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$. In particular, OV requires a multiply-accumulate operation multiplying a vector of field elements by a single field element. For $\mathbb{F}_{16}$, we make use of the bitsliced arithmetic proposed by Chou, Kannwischer, and Yang [20] for (a tweaked version) of Rainbow. It bitslices the vector while keeping the single field element in a single register and accessing individual bits. For $\mathbb{F}_{256} = \mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$, we are not aware of any Cortex-M4 implementation supporting the required vector by scalar multiply-accumulate operation. We, hence, write our own. We present two implementations: One operating on four field elements packed into one 32-bit register (i.e., byte-sliced) and one operating on 32 field elements bit-sliced into eight registers. The former turns out to be superior.

Algorithm 3 presents our byte-sliced implementation of the multiplication of four field elements packed into the register $a$ by one field element in the least significant byte of the register $b$ and then adds the result to the four elements in $c$. It works by first pre-computing $b, bx, bx^2, ..., bx^7$ and storing the result in eight floating-point registers. It then goes through the bits of the four field elements stored in $a$, masks them out (line 10), multiplies the individual bits by the corresponding pre-computed multiple of $b$, and then adds the result to the accumulator. For a single input register, the instruction sequence takes 36+32=68 instructions, i.e., 68 clock cycles (17 cycles/element). However, if more than four elements have to be multiplied by $b$, one can easily extend the sequence to multiple inputs. In that case, the instructions printed in boldface are only required once. We can, hence, implement a multiply-accumulate for 8, 12, and 16 field elements in 92, 116, and 140 clock cycles (11.5 cycles/element, 9.7 cycles/element, and 8.75 cycles/element). For larger vectors, the inputs do not fit in registers anymore, however, one can still re-use the precomputation. For example, multiplying 32 field elements takes $36 + 104 + 104 = 244$ clock cycles (7.6 cycles/element).

Our second multiply-accumulate implementation is bitsliced implementation multiplying

**Algorithm 4** $\mathbb{F}_{256} = \mathbb{F}_2[x]/\langle x^8 + x^4 + x^3 + x + 1\rangle$ multiply-accumulate on 32 bitsliced elements. As there are not enough registers available for all inputs in outputs, we split the computation into the lower and upper bits of the product. We cache the other values in floating-point registers (not shown here). Requires 162 clock cycles ($2 \times 65$ cycles for arithmetic plus 32 cycles `vmov`) for 32 field multiplications excluding bitslicing.

**Input:** Bitsliced first multiplicand $a_0, ..., a_7$ (32 elements)
**Input:** Second multiplicand $b$ (1 field element in the least significant byte)
**Input/Output:** accumulator $c$ (4 field elements packed into one register)

```
tst b, #1
itttt ne
eorne c0, c0, a0
eorne c1, c1, a1
eorne c2, c2, a2
eorne c3, c3, a3
eor a0, a0, a7
eor a2, a2, a7
eor a3, a3, a7
tst b, #2
itttt ne
eorne c0, c0, a7
eorne c1, c1, a0
eorne c2, c2, a1
eorne c3, c3, a2
eor a7, a7, a6
eor a1, a1, a6
eor a2, a2, a6
tst b, #4
itttt ne
eorne c0, c0, a6
eorne c1, c1, a7
eorne c2, c2, a0
eorne c3, c3, a1
eor a6, a6, a5
eor a0, a0, a5
eor a1, a1, a5
tst b, #8
itttt ne
eorne c0, c0, a5
eorne c1, c1, a6
eorne c2, c2, a7
eorne c3, c3, a0
eor a5, a5, a4
eor a7, a7, a4
tst b, #16
itttt ne
eorne c0, c0, a4
eorne c1, c1, a5
eorne c2, c2, a6
eorne c3, c3, a7
eor a4, a4, a3
eor a6, a6, a3
tst b, #32
itttt ne
eorne c0, c0, a3
eorne c1, c1, a4
eorne c2, c2, a5
eorne c3, c3, a6
eor a3, a3, a2
eor a5, a5, a2
tst b, #64
itttt ne
eorne c0, c0, a2
eorne c1, c1, a3
eorne c2, c2, a4
eorne c3, c3, a5
eor a2, a2, a1
eor a4, a4, a1
tst b, #128
itttt ne
eorne c0, c0, a1t
eorne c1, c1, a2
eorne c2, c2, a3
eorne c3, c3, a4
//continue with 4 most significant bits
```

a vector of 32 elements bitsliced into 8 registers $a_0, \ldots, a_7$ by a single element in the least significant byte of $b$. We first require an efficient transformation of byte-sliced field elements into bit-sliced representation (and the inverse transformation). We implement a straightforward adaptation of [20, Algorithm 7] requiring 128 clock cycles. Note that the register pressure is very high (8 inputs and 8 outputs) in this case and we, hence, resort to storing the result in floating-point registers. Algorithm 4 shows (part of) our bit-sliced multiply-accumulate implementation. An essential difference to $\mathbb{F}_{16}$ is that the register pressure is much higher as we require 17 registers to keep $a, b$, and $c$ which is more than available on the Cortex-M4. We work around this by keeping inputs and outputs in floating-point registers and by sequentially computing the two halves (4 bits) of the output. Using this trick, we require $8 + 1 + 4 = 13$ general-purpose registers during the computation. The bitsliced computation proceeds by repeatedly multiplying $a$ by $x$ and, then conditionally on the bits of $b$ adding to the accumulator. The multiplication by $x$ is implemented using three `eor` instructions and implicit variable renaming implementing the shifts. After the first four bits are computed, the results are stored in floating-point registers, and the original inputs $a_0, \ldots, a_7$, and the upper four bits of the accumulator $c_4, \ldots, c_7$ are fetched from floating-point registers. The second half then proceeds in the same way as the first half. In the last multiplications by $x$, not all bits of the outputs are being used and we, hence, eliminate all instructions computing unused bits. A full multiply-accumulate operation with byte-sliced inputs and outputs requires $128 + 162 + 128 = 418$ clock cycles (13.1 cycles/element). Note that this is slower than calling byte-sliced implementation for 32 field elements from Algorithm 3 twice (244 clock cycles ). Even when taking into account that in all cases within OV, either $a$ or $c$ can be kept in bitsliced representation, we require $128 + 162 = 290$ clock cycles (9.1 cycles/element) for a multiply-accumulate operation for 32 field elements. Therefore, the bitsliced implementation does not appear promising.

## 6.3 Solving Linear Equations

As described in Subsection 3.1, there are two approaches for solving linear equations in signing: matrix inversion followed by matrix-vector multiplication, and direct equation solving using Gaussian elimination. While the number of multiplications clearly favours the latter, we implement both for the Arm Cortex-M4 to compare their actual performance.

### 6.3.1 (Blocked) Matrix inversion

We implement matrix inversion both with and without the blocked matrix inversion approach. We require the inversion of a $64 \times 64$ matrix for the $\mathbb{F}_{16}$ parameter set and the inversion of a $44 \times 44$ matrix for the $\mathbb{F}_{256}$ parameter set. Hence, when using blocked matrix inversion, we need a $32 \times 32$ $\mathbb{F}_{16}$ matrix inversion and a $22 \times 22$ $\mathbb{F}_{256}$ matrix inversion. For $\mathbb{F}_{16}$, our implementation is very close to the implementation of Chou, Kannwischer, and Yang [20]. We adapt the dimensions to 64 and 32. For $\mathbb{F}_{256}$, we implement the same algorithm and make use of the field multiplication from Subsection 6.2.

In addition, as a part of constant-time Gaussian elimination, we need to invert individual $\mathbb{F}_{256}$ field elements. We make use of the constant-time extended Euclidean algorithm as proposed by Bernstein and Yang [9]. For $\mathbb{F}_{256}$, our implementation consists of 230 instructions, i.e., 230 cycles. We believe that there is likely a better way to implement the inversion as previous work on the bitsliced AES SBox which includes an $\mathbb{F}_{256}$ inversion requires only 113 logic gates [15]. However, since the field inversion only accounts for a negligible share of the matrix inversion cycles, we do not further investigate faster approaches.

### 6.3.2 Directly solving the linear system of equations

In addition to the (blocked) inversion, we also implement the linear equations solving using constant-time Gaussian elimination as shown in Algorithm 2 to study if the tricks introduced in [48] are worthwhile on the Arm Cortex-M4. The implementation proceeds similarly to the matrix inversion but requires much fewer multiplications.

### 6.3.3 Comparison

The upper part of Table 9 presents the results for the matrix inversion. We report the cycles both with and without using blocked inversion. For the blocked inversion, we also report the cycle counts for the smaller (half-sized) inversion. Blocked inversion provides a 2.1× speed-up for the $\mathbb{F}_{16}$ parameter set and a 1.5× speed-up for the $\mathbb{F}_{256}$ parameter set over the non-blocked inversion. However, when looking at the results for solving the linear system of equations (lower part of Table 9), we see that the blocked inversion is slower than directly solving the equations using constant-time Gaussian elimination. The gap is particularly large for $\mathbb{F}_{256}$ as $\mathbb{F}_{256}$ multiplications are particularly costly which makes it more important to minimize the number of multiplications. We conclude that (blocked) matrix inversion is not worthwhile for the Cortex-M4.

Table 9: Cycles counts on the Arm Cortex-M4 for a matrix inversion with and without blocked inversion as well as linear equation solving. Blocked inversion is significantly faster (2.1× for $\mathbb{F}_{16}$ and 1.5× for $\mathbb{F}_{256}$) than non-blocked inversion. However, directly using Gaussian elimination is even faster (1.1× for $\mathbb{F}_{16}$ and 2.5× for $\mathbb{F}_{256}$.)

| (Blocked) matrix inversion | | | | | |
|---|---|---|---|---|---|
| $\mathbb{F}_{16}$ | | | $\mathbb{F}_{256}$ | | |
| $d$ | | cycles | $d$ | | cycles |
| 64 | — | 1 499 802 | 44 | — | 1 645 998 |
| 64 | blocked | 720 904 | 44 | blocked | 1 086 057 |
| 32 | — | 189 289 | 22 | — | 207 427 |

Solving $\mathbf{Lx} = \mathbf{t} - \mathbf{y}$

| | $\mathbb{F}_{16}$ | $\mathbb{F}_{256}$ |
|---|---|---|
| Using blocked inversion | 742 956 | 1 194 424 |
| Using Gaussian elimination | 636 453 | 438 891 |

## 6.4 Verification

We implement the verification using the techniques for reducing the number of multiplications as described in Subsection 3.2. In addition, we implement the following target-specific optimizations:

**"Lazy sampling" for memory-efficient implementations.** Note the lazy sampling technique from Subsection 3.2 has the additional benefit of not requiring the store the expanded $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ matrices in RAM which is desirable for microcontroller implementations. To keep the code simple, we always sample one contiguous block of the public key corresponding to one variable $s_i$, and then loop over the remaining variables $s_j$, i.e., we sample in the outer loop of verification. If $s_i = 0$, then we don't require this part of the public key at all. We, hence, don't have to sample it. This way, we require at most $m \times n$ field elements in memory, keep the sampling overhead small, and the code remains easy to read.

**T-table AES implementation for sampling the expanded public key.** As some Cortex-M4 platforms contain a data cache [36], it is important to consider cache-timing attacks for the implementation of the pseudo-random sampling using AES. However, in the case of the public-key expansion, this is of no concern. We, hence, make use of the fast t-table implementation by Schwabe and Stoffelen [47].

## 6.5 Symmetric cryptography

For implementing $\mathsf{Hash}$, $\mathsf{Expand}_{\mathbf{v}}$, and $\mathsf{Expand}_{\mathsf{sk}}$, we use `shake256` as implemented in pqm4 [31] which integrates the Keccak permutation in `Armv7-M` assembly from the XKCP [21]. For implementing the sampling of the public key ($\mathsf{Expand}_{\mathbf{P}}$), we use the t-table AES implementation by Schwabe and Stoffelen [47]. We also modify said implementation to implement a round-reduced AES with only 4 rounds. We present results both for the 10-round and 4-round AES.

## 6.6 Results

In the following, we present the performance of the Cortex-M4 implementation described above.

Table 10: Cortex-M4F cycle counts for our M4 implementations in comparison to the fastest implementations of the winners of the NIST PQC competition and Rainbow. For signing and verification we report the average of 10 000 executions.

| | | speed (clock cycles) | | |
|---|---|---|---|---|
| | variant | KeyGen | Sign | Verify |
| `ov-Ip` (This work) $\mathbb{F}_{256}$ | classic | 138 833k | 2 482k | 995k |
| | pkc | 175 020k | | 11 551k (10 717k) |
| | pkc+skc | 175 021k | 88 757k | |
| `ov-Is` (This work) $\mathbb{F}_{16}$ | classic | 195 744k | 2 374k | 616k |
| | pkc | 203 321k | | 16 045k (15 175k) |
| | pkc+skc | 296 161k | 113 446k | |
| RainbowI [20] | classic | 98 431k | 957k | 239k |
| | CZ | 107 639k | | 12 903k |
| | comp. | 107 711k | 56 643k | |
| Dilithium 2 [1] | | 1 598k | 4 083k | 1 572k |
| Falcon-512 [46, 31] | | 163 994k | 39 014k | 473k |
| sphincs-sha256-128f-simple [31] | | 16 112k | 400 443k | 22 548k |
| sphincs-sha256-128s-simple [31] | | 1 031 755k | 7 848 131k | 7 711k |

**Target platform.** We use the ST NUCLEO-L4R5ZI development board featuring a STM32L4R5ZI ultra-low-power Arm Cortex-M4F core with 640 KB of RAM, and 2048 KB of flash memory. It runs at a frequency of up to 120 MHz. However, we clock the device at 16 MHz allowing for zero wait-states when fetching instructions and data from flash. For benchmarking, we use the pqm4 [31] benchmarking framework.

**Keys exceeding RAM size.** For the `ov-Is` ($\mathbb{F}_{16}$) parameter sets, the combined size of the expanded secret key and the expanded public key is 743 KB which exceeds the RAM of our target platform. To still be able to benchmark all primitives, we split up key generation into secret key and public key computation. We then write the keys to flash memory as was previously proposed by Chen and Chou for Classic McEliece [17]. This requires minimal code modification while still being able to provide benchmarks for all parts of the scheme. Higher security levels, however, are out of reach for running on the Cortex-M4.

Table 10 contains the performance benchmarks for Arm Cortex-M4. We present cycle counts for all six variants of the level one parameter sets. Due to timing variations (depending only on public data) in signing and verification, we perform 10 000 measurements and report the average. Note that public key compression does not affect signing performance, while secret key compression does not affect verification performance. For the `ov-Is`, the key generation cycles exclude the writing of keys to flash. We report the flashing cycles separately in Table 12.

For verification with compressed public keys, there are two approaches available: Either expanding the public key first and calling the classic verification, or inlining the expansion as described in Subsection 6.4. The former approach has a much larger memory footprint, but has slightly better speed.

Table 11 contains the memory utilization of our implementation excluding the key material. The parameter sets using secret key compression are currently performing signing by first

Table 11: Cortex-M4F memory utilization (excluding keys) for our OV implementation in comparison to the fastest implementations of the winners of the NIST PQC competition and Rainbow. Code size excludes 3.5 KiB of platform code and includes the code required for SHAKE (7.5 KiB) and AES (4.6 KiB).

| | | memory consumption (bytes) | | | code size (KiB) |
|---|---|---|---|---|---|
| | variant | KeyGen | Sign | Verify | |
| `ov-Ip` (This work) $\mathbb{F}_{256}$ | classic | 15 744 | 5 268 | 2 548 | 72.4 |
| | pkc | 142 312 | 5 268 | 6 592 (280 980) | 75.3 |
| | pkc+skc | 380 248 | 243 204 | 6 592 (280 980) | 75.5 |
| `ov-Is` (This work) $\mathbb{F}_{16}$ | classic | 613 056 | 5 468 | 1 024 | 31.6 |
| | pkc | 350 072 | 5 468 | 5 248 (413 632) | 33.2 |
| | pkc+skc | 416 636 | 354 216 | 5 248 (413 632) | 33.6 |
| RainbowI [20] | classic | 40 696 | 4 052 | 812 | 56.0 |
| | CZ | 142 304 | 4 052 | 20 156 | 51.0 |
| | comp. | 245 976 | 224 240 | 20 156 | 53.0 |
| Dilithium 2 [1] | | 38 000 | 49 000 | 36 000 | 26.0 |
| Falcon-512 [46, 31] | | 18 384 | 42 528 | 4 484 | 79.9 |
| sphincs-128f† [31] | | 2 104 | 2 168 | 2 656 | 13.3 |
| sphincs-128s‡ [31] | | 2 432 | 2 392 | 1 960 | 13.6 |

†sphincs-sha256-128f-simple. ‡ sphincs-sha256-128s-simple.

Table 12: For the `Is` parameter sets the keys are too large to fully fit in RAM, we, hence, write them to flash during key generation. Cycles in Table 10 exclude the cycles required for flashing. This table contains the cycles required for flashing as well as the total key generation cycles.

| | | key generation w/o flashing (cc) | flashing (cc) | key generation w/ flashing (cc) |
|---|---|---|---|---|
| `ov-Is` $\mathbb{F}_{16}$ | classic | 195 744k | 202 296k | 398 040k |
| | pkc | 203 321k | 110 744k | 314 065k |
| | pkc+skc | 296 161k | 18 287k | 314 447k |

expanding the secret key and then invoking the classic signing and, hence, require an expanded secret key in additional memory. Key generation of `ov-Is` requires much more memory than `ov-Ip`. This is due to having to cache the keys in RAM before writing them to flash.

Table 13 presents the cycle counts when using a round-reduced AES (4 rounds instead of 10 rounds) for expanding the public key. It results in significantly faster verification (2.0× for `ov-Ip` and 2.1× for `ov-Is`).

# 7 FPGA Implementation

In this section, we present our field-programmable gate array (FPGA) design for OV signatures and report the performance of the design on popular platforms. Since our design supports multiple parameters and variants of OV, we adopt a processor design that provides a custom instruction set dedicated for the computation of OV functions. This

Table 13: Cortex-M4F cycle counts when using 4-round AES for expanding the public key. This change primarily affects the verification procedure providing a 2.0× speed-up for `ov-Ip` and a 2.1× speed-up for `ov-Is`.

| | | speed (clock cycles) | | |
|---|---|---|---|---|
| | variant | KeyGen | Sign | Verify |
| `ov-Ip` (This work) $\mathbb{F}_{256}$ | pkc | 169 280k | 2 502k | 5 804k |
| | pkc+skc | 169 281k | 83 018k | |
| `ov-Is` (This work) $\mathbb{F}_{16}$ | pkc | 194 875k | 2 390k | 7 594k |
| | pkc+skc | 287 715k | 105 004k | |

way, we support the key generation, signing, and verification functions in Figure 1 with pre-loaded firmware using the proposed instructions.

The FPGA presents a good platform to design, simulate, and test customized hardware implementations performing specific algorithms. Although state-of-the-art FPGAs provide a large number of programmable resources to the designers, in many practical deployments programmers still need to adapt their design to particular FPGAs with limited resources. In the paper, we test our design on two Xilinx Artix-7 platforms: Zynq-7000™ Z-7020 and Artix-7 XC7A200T. We target Artix-7 as it is the hardware target platform recommended by NIST [4] for the PQC standardization effort. Consequently, other PQC schemes have also been implemented on Artix-7 allowing comparison to our implementation. Z-7020 is the core chip of several popular development boards for educational purposes due to the relatively low cost and the easy-to-use toolchain for testing and verification. It provides an integrated SoC platform, including a processing system (PS) component consisting of a Arm Cortex-A9 processor and a programmable logic (PL) component of the Artix-7 architecture. We use only the PL component in the work. On the other hand, XC7A200T is the largest Artix-7 platform we are aware of, which provides abundant hardware resources [54]. We have chosen this platform to validate that, even for the largest parameter of OV, Artix-7 is capable to run the variants of compressed keys. Our design is fully parameterized. Although we report our results with a setting tailoring for the Artix-7 platforms, it can be easily adapted to other parameter sets and ported to other FPGAs.

Since we use a processor design for performing OV in hardware, our hardware modules can be roughly divided into 3 categories according to their functionalities. These 3 categories are (1) an instruction memory for storing firmware and a decoder for decoding user code and sending control signals to other hardware modules for computation, (2) data memory responsible for storing OV keys and data movement from/to the computation modules, and (3) the modules for performing actual computations. We describe the topics relating the 3 categories in Subsection 7.1, Subsection 7.2, and Subsection 7.3, respectively. We report the implementation results of our design in Subsection 7.4.

## 7.1 Instruction Set Architecture

We describe our customized instruction set for OV in this section. Since we adapt a processor design, a complete OV implementation includes not only hardware modules for actual computation but also an instruction set for controlling the hardware modules and the firmware performing functionalities of signature systems with the customized instructions. This design aims to simplify the hardware implementation of the OV scheme and provide multiple functionalities in one design with the customized instruction set. The instructions hence provide basic flow control and operations that are commonly used in key generation, signing, and verification. In this way, we only need to focus on designing modules for the critical operations in the scheme (e.g., Gaussian elimination, polynomial evaluation) and utilize these modules repeatedly to carry out the computation. Besides, code maintenance becomes much more manageable, as it is simple to insert or remove an instruction without touching the existing instructions.

Instructions can be divided into two categories: (1) control instructions and (2) function instructions. Control instructions are meant to control the program flow, while function instructions perform the actual computation accounting for the vast majority of run time. We provide 16 control registers `r0-r15` and one program counter to control the program. These control registers also serve as indices or counters of loops for a complex function instruction. In our case, 16 registers are enough to construct nested loops and hold the temporary values in key generation, signing, and verification.

Function instructions can be further divided into three parts: (1) core instructions, (2) AES-related instructions, and (3) SHAKE-related instructions. Table 14 and Table 15 list the function instructions and describe their functionalities in detail. Note that we have dedicated hardware modules for the 3 parts of function instructions so each part operates independently. Therefore, we can perform polynomial evaluations (core) and public key sampling (AES) at the same time.

Function instructions have different numbers of inputs, resulting in different instruction encodings. We optimize the instruction encoding and aim to use as few bits as possible. In our implementation, we use 32 bits to encode the instructions. The reason for this choice is that when we use the instruction `addi(r2, r1, imm)` to set the AES counter to register `r2`, the `imm` bit field requires 18 bits to cover the range of AES counter values to be able to sample the public key of the largest parameter set (151 404 AES blocks for `ov-V-pkc`/`ov-V-pkc+skc`). Additionally, we use 16 registers, so the `r1` and `r2` bit-fields require 4 bits each. Lastly, we use 6 bits for the opcode. Therefore, the total number of bits needed is $18 + 4 + 4 + 6 = 32$. We also support up to 1024 instructions, which is enough for the instructions used in key generation, signing, and verification, and this is achievable by using 1 BRAM36K.

By using the customized instructions, we perform key generation, signing, and verification with their firmware on an FPGA. Algorithm 5 shows an example of the firmware performing classic verification in `ov-Ip`. It uses only a few instructions to perform the verification function. Instruction `load_keys(ZERO, r0, r0)` loads zeroes to the data register, and the instruction `eval` does the polynomial evaluation in hardware. Here the `eval` instruction takes an immediate value specifying the part of the public key. It will further be decoded into a serial control signals for accessing key data and performing computations for the particular part of the key in a matrix processing module described in Subsubsection 7.3.1. Since we use only one processing module due to the area limit of our FPGAs, the next `eval` instruction won't be dispatched until its previous `eval` finishes. Finally, we shift the data out and compare it with the hashed message using `unload_check`.

## 7.2 Memory Management

Due to the large key size of OV, memory management becomes a crucial aspect for mapping our design to an FPGA device and achieving high performance. The on-chip storage of FPGAs is composed of multiple stripes of BRAMs, with each stripe containing multiple BRAM36Ks, as depicted in Figure 5a. In our implementation, we use the Zynq-7000 and XC7A200T FPGA boards which have 140 and 365 BRAMs, respectively. We implement `ov-Ip`, `ov-Is` on the Zynq-7000 and `ov-III`, `ov-V` on XC7A200T. However, the key size for certain parameter sets and variants still exceeds 70% of the available BRAMs leaving little room for other logic requiring on-chip memory resources. As an example, in Table 16, the

Table 14: Function instructions: core instructions.

| Instruction | Description |
|---|---|
| `store_keys(imm,r1,r2)` | Store values in data registers into the column of Macaulay matrix $(\mathbf{P}^{(1)}, \mathbf{P}^{(2)}, \mathbf{P}^{(3)}, \mathbf{S})$. The address (`imm`, `control_reg[r1]`, `control_reg[r2]`) is translated into the represented column of the Macaulay matrix. |
| `load_keys(imm,r1,r2)` | Load data in address (`imm`, `control_reg[r1]`, `control_reg[r2]`) of the Macaulay matrix to data registers. |
| `mul_key_o(imm1, r1, r2, X, r3, r4)` | Multiply data in (`imm1`, `control_reg[r1]`, `control_reg[r2]`) of Macaulay matrix with `control_reg[r3]` row, `control_reg[r4]` column of matrix $\mathbf{O}$, and accumulate the results in data registers. |
| `mul_key_sig(imm, r1, r2)` | Shift $m \cdot \log_2 \lvert\mathbb{F}_q\rvert^\dagger$ bits from the AES buffer to the random registers in the systolic array. Then, multiply data in the random registers with `s[control_reg[r1]]`·`s[control_reg[r2]]` (`s` is signature), and accumulate the results with values in data registers. |
| `eval(imm)` | Perform polynomial evaluations on the Macaulay matrix. It calculates $\mathbf{t}$ in signing and verification. |
| `unload_add_y()` | Shift data register results out, and perform $(\mathbf{t} - \mathbf{y})$ in signing. |
| `calc_l(imm, X, r2)` | Calculate the column `control_reg[r2]` of $\mathbf{L}$ in signing. |
| `store_l(r1)` | Shift data register results out and prepare for the column `control_reg[r1]` of the matrix $\mathbf{L}$ in signing. |
| `gauss_elim(imm)` | Perform Gaussian elimination to solve $\mathbf{Lx} = \mathbf{t} - \mathbf{y}$ in signing. Program counter jumps to `imm` if it fails. |
| `mul_o(X, r1, r2)` | Perform matrix-vector multiplication $\mathbf{O} \cdot \mathbf{x}$ in signing. Register `r1`, `r2` specify the submatrix of $\mathbf{O}$ that is being multiplied. |
| `add_to_sig_v(r1)` | Shift data register results out, and perform addition $\mathbf{v} + \mathbf{Ox}$ in signing. Register `r1` specifies the subvector that is being processed. |
| `unload_check(r1)` | Shift data register results out and compare them with $\mathbf{t}$ in verification. Register `r1` specifies the parameter $m$ in the OV scheme. |

$^\dagger$ $\log_2 \lvert\mathbb{F}_q\rvert$ is 4 in `ov-Is`, and 8 for other parameter sets.

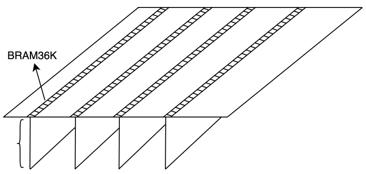
`ov-Ip` and `ov-Is` in classic mode use 72% (101/140) and 106% (149/140) of the BRAMs, respectively, while `ov-III-pkc` and `ov-V-pkc+skc` use 82% (300/365) and 105% (382/365) of the BRAMs, respectively. This results in a more complex place-and-route as the limited resources require careful allocation and management. Therefore, we have to allocate keys so that computing units can fetch them from local BRAMs, to avoid complicating the routing and ultimately making the design placeable and routable on FPGA.

Figure 5b shows the keys in the Macaulay matrix and how we map the key data into BRAMs in an FPGA. In the top figure, we divide the key matrix into 3 submatrices (denoted by $a, b$, and $c$) and store the 3 submatrices in 3 memory units of different colors in the bottom left figure. This way, the 3 memory units are capable of providing one column of the matrix to computation modules in the same cycle. The “key depth” of the memory units represents the width of the submatrices, which can be divided into several parts $\mathbf{P}^{(1)}, \mathbf{P}^{(2)}, \mathbf{S}$, and $\mathbf{P}^{(3)}$ with width of $v \cdot (v+1)/2, v \cdot m, v \cdot m$, and $m \cdot (m+1)/2$, where $v = n - m$ ,respectively. Since the key depth is larger than the capacity of a typical BRAM, each memory unit requires multiple BRAMs to store all required data in the bottom right figure. In reality, the BRAMs for one memory unit may spread across the FPGA. When accessing a particular column of a matrix, we use a multiplexer to select the required data from multiple BRAMs. To address the potential timing issues that can arise when working with such multi-port memories, we introduce a delay of a few clock cycles, which allows the selected data to fully propagate through the multiplexer and be properly read from the BRAMs, thus ensuring correct and reliable output.

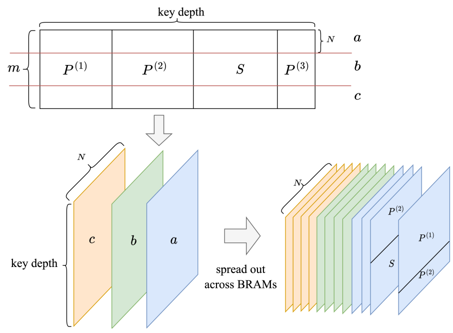
Table 15: Function instructions: AES and SHAKE instructions.

| Instruction | Description |
|---|---|
| `aes_init_key()` | Sample the $\mathsf{seed}_{\mathsf{pk}}$ and set it as the AES key. |
| `aes_set_round(imm)` | Initialize the AES round to `imm`. |
| `aes_set_ctr(X,r1,imm)` | Set the AES counter to `imm` + `control_reg[r1]`. |
| `aes_update_ctr(imm)` | AES encrypt the current plaintext, store the ciphertext to the AES buffer and add `imm` to the current AES counter. |
| `send()` | Shift $m \cdot \log_2 \|\mathbb{F}_q\|^\dagger$ bits from the AES buffer to the data registers in the systolic array. |
| `shake_hash_sk(imm)` | Sample the $\mathsf{seed}_{\mathsf{sk}}$ to `imm`, generate 8× `imm` bytes digest and store it to the SHA buffer. |
| `shake_squeeze_sk(imm)` | Squeeze out 8×`imm` bytes digest and store it to the SHA buffer. |
| `shake_hash_v()` | Perform $\mathsf{Expand}_{\mathsf{v}}(M\|\|\mathsf{salt}\|\|\mathsf{seed}_{\mathsf{sk}}\|\|\mathsf{ctr})$. |
| `shake_hash_m()` | Perform $\mathsf{Hash}(M\|\|\mathsf{salt})$. |
| `store_o(r1)` | Shift $(n - m) \cdot \log_2 \|\mathbb{F}_q\|^\dagger$ bits from the SHAKE buffer and store it to `control_reg[r1]` column of matrix $\mathbf{O}$. |

$^\dagger$ $\log_2 |\mathbb{F}_q|$ is 4 in `ov-Is`, and 8 for other parameter sets.




(a) BRAMs on FPGA.




(b) The memory layout on FPGA in `ov-Ip`.

The key depth for different OV variants decides the number of BRAMs used in the design. Table 16 summarizes the total key depth required for different variants and numbers of BRAMs for different parameter sets. The total depth is composed of two different parts: the storage for keys and temporary space during the computation. The number of BRAMs required for different variants and parameter sets cannot exceed that on Zynq-7000 (140) or XC7A200T (365) by too much. For the case that the number of BRAMs is slightly larger than the capacity of a board (e.g., `ov-V` in pkc-skc for XC7A200T), we use LUTRAMs to fill the gap. However, `ov-III` classic, `ov-V` classic, and `ov-V-pkc` are over the capacities of our target FPGAs.

Here, we describe the additional depth for temporary storage in detail:

- For OV classic, we store $\mathbf{P}^{(1)}, \mathbf{P}^{(2)}, \mathbf{P}^{(3)}$, and $\mathbf{S}$ in the BRAM. During the computation, we do not use additional temporary storage since we use key storage as temporary storage to allow doing all the computation in-place. For example, in **ExpandSK()**, we store $\mathbf{P}^{(2)}$ first and re-use the same memory for $\mathbf{S}$.
- For OV pkc and OV pkc-skc, we include $v(v+1)/2$ depth for temporary storage to store $\mathbf{P}^{(1)}$. The reason is that during key generation, the computation involving $\mathbf{P}^{(1)}$ requires specific AES counter indices in $\mathbf{Expand}_P$ to obtain the corresponding column of $\mathbf{P}^{(1)}$ in the Macaulay matrix. For example, in `ov-Ip`, we need counters to

**Algorithm 5** The firmware performing verification in `ov-Ip`. P1, P2, and P3 are immediate values that represent the Macaulay matrix $\mathbf{P}^{(1)}$, $\mathbf{P}^{(2)}$, and $\mathbf{P}^{(3)}$, respectively, which are being evaluated. The values of P1, P2, and P3 may vary depending on the variant of the design. The ZERO also represents an immediate value, which corresponds to the BRAM address storing zeros.

addi(r15, r0, 44) ▷ Initialize r15
addi(r14, r0, 68) ▷ Initialize r14
shake_hash_m() ▷ Perform $\mathsf{Hash}(M||\mathsf{salt})$
stall(24+3) ▷ Wait for the hash
load_keys(ZERO, r0, r0) ▷ Clean up the data register.
eval(P1) ▷ Perform $\{\mathbf{s}^\mathsf{T}\mathbf{P}_i^{(1)}\mathbf{s}\}_{i\in m}$.
eval(P2) ▷ Perform $\{\mathbf{s}^\mathsf{T}\mathbf{P}_i^{(2)}\mathbf{s}\}_{i\in m}$.
eval(P3) ▷ Perform $\{\mathbf{s}^\mathsf{T}\mathbf{P}_i^{(3)}\mathbf{s}\}_{i\in m}$.
unload_check(r15) ▷ Shift out the result and check
stall(5) ▷ Wait for the check. Return Reject if it fails.
finish() ▷ Return Accept

Table 16: BRAM36K utilization in different OV variants. Note that $v = n - m$.

| OV variants | classic | pkc | pkc-skc |
|---|---|---|---|
| Keys stored in design | $\mathbf{P}^{(1)}, \mathbf{P}^{(2)}, \mathbf{P}^{(3)}, \mathbf{S}$ | $\mathbf{P}^{(3)}, \mathbf{S}$ | $\mathbf{P}^{(3)}$ |
| Key depth for key storage | $2 \cdot v \cdot m + v \cdot (v+1)/2$ $+ m \cdot (m+1)/2$ | $v \cdot m + m \cdot (m+1)/2$ | $m \cdot (m+1)/2$ |
| Additional depth for temporary storage | 0 | $v \cdot (v+1)/2$ | $4 \cdot v + v \cdot (v+1)/2$ |
| # BRAMs for `ov-Ip` | 101 | 68 | 39 |
| # BRAMs for `ov-Is` | 149 | 101 | 56 |
| # BRAMs for `ov-III` | 441 | 300 | 165 |
| # BRAMs for `ov-V` | 1 066 | 724 | 382 |

be set to $0, 1$, and $2$ to obtain the first column of $\mathbf{P}^{(1)}$, and to $2, 3, 4$, and $5$ to obtain the second column of $\mathbf{P}^{(1)}$. This requirement complicates the hardware design as it requires the addition of logic to calculate the mapping between counter indices and column indices of $\mathbf{P}^{(1)}$. Furthermore, additional buffers and shifters are needed to transform the output of `aes128ctr` to the column format. To avoid this issue, we pre-expand $\mathbf{P}^{(1)}$ at the start of key generation and signing, and store it for later use.

- Finally, in OV pkc-skc, we observe that for the computation of $\{\mathbf{v}^\mathsf{T}\mathbf{S}_i\}_{i\in m}$ in signing, we can calculate few columns of $\mathbf{S}_i$ on the fly and multiply them with $\mathbf{v}^\mathsf{T}$ without preparing a whole $\mathbf{S}_i$ matrix in the beginning. In addition, the calculation of columns of $\mathbf{S}_i$ can also be done using columns of $\mathbf{P}_i^{(2)}$, whose AES counter indices are easier to infer than $\mathbf{P}^{(1)}$. In this way, we can allocate only additional $4v$ depth to hold the columns of $\{\mathbf{S}_i\}_{i\in m}$ or $\{\mathbf{P}_i^{(2)}\}_{i\in m}$.

## 7.3 OV Processor Design

In this section, we outline our hardware design for key generation, signing, and verification. The block diagram of the hardware design is depicted in Figure 6. Its main components comprise a systolic array in the right, a microcontroller in the top left, and AES and SHAKE units in the bottom left. We detail the systolic array, the core component of the design, in Subsubsection 7.3.1. After that, we describe the SHAKE-256 and AES-128 units

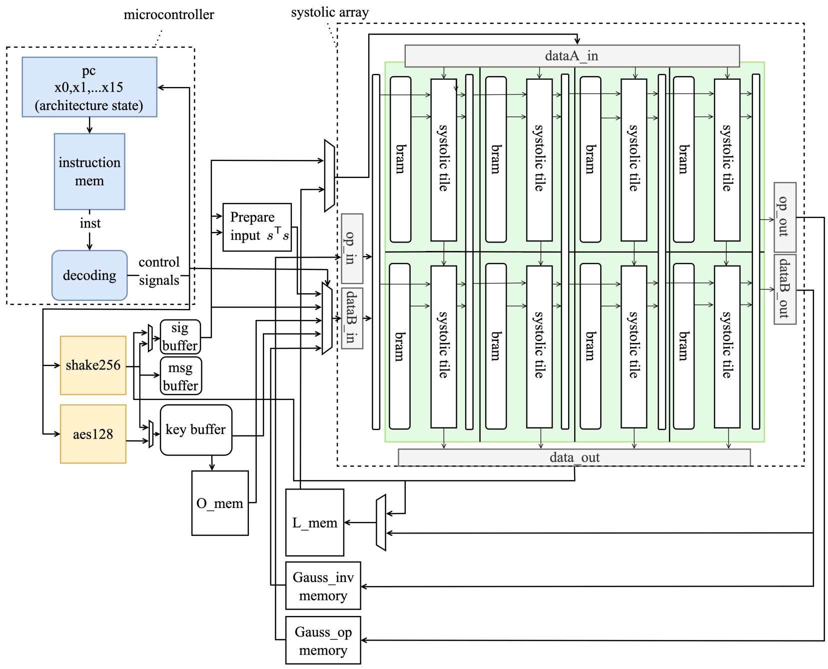



Figure 6: The block diagram of the OV processor. At the top left corner, it shows a microcontroller that fetches instructions and controls other modules. The AES and SHAKE modules generate data for the systolic array in the top right, which reads inputs from the top and left and writes outputs to the bottom and right. The outputs are stored in memories or buffers located outside of the systolic array, which subsequently serve as inputs for the systolic array.

in Subsubsection 7.3.2. Lastly, we present the microcontroller and instruction decoders in Subsubsection 7.3.3.

### 7.3.1 Systolic Array

Systolic arrays were introduced by Kung and Leiserson in 1978 [35], consisting of many small processor units whose functions are specialized and which are connected in a specified manner to achieve good performance of a pre-determined task to be done. Hochet, Quinton, and Robert [26] proposed a systolic array approach to solve large dense linear systems of equations. Later, Wang, Szefer, and Niederhagen [52] proposed the modular approach on FPGAs to construct a systolic array solving linear systems of equations on $\mathbb{F}_2$. The follow-up work by Wang, Szefer, and Niederhagen [53] further extended $\mathbb{F}_2$ to $\mathbb{F}_{2^m}$ to build a key-generator for the Niederreiter cryptosystem using binary Goppa codes. It includes an additional inverter to perform systemization of matrices in $\mathbb{F}_{2^m}$. Moreover, Chen, Chou, Deshpande, Lahr, Niederhagen, Szefer, and Wang [19] focused on the early-abort function of the systolic array when detecting a non-invertible matrix over $\mathbb{F}_2$, which is necessary to accelerate the speed in the Classic McEliece cryptosystem. As the probability of the non-invertible matrix over $\mathbb{F}_{16}$ and/or $\mathbb{F}_{256}$ occurring is much smaller than over $\mathbb{F}_2$, we still adopt the approach of [52], and modify the processor units making our systolic array suitable for the equation-solving in the OV scheme by extending them to $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$, and re-utilize the processor units to complete various expensive functions.

**Previous design for Gaussian elimination.** The authors in [52] presented a design to compute the row echelon form of a matrix $M$ in a module `comb_SA` with storage only capable of a small portion of elements in $M$. In the solution, they split $M$ into several block columns, perform elimination on one block column with pivots in `comb_SA`, store the row operations for the pivot column, and repeat the row operations on other columns.

The module `comb_SA` takes a row vector of size $w$ from a column block of $M$ as its input outputs an eliminated row vector in every cycle. It contains $w \times w$ small processors. These processors are organized as $w$ connected row units. Each unit comprises $w$ processors and thus is capable of storing a row vector and performing row operations. A row unit takes a row vector as input and outputs a processed row vector as well as the row operation it has performed. An input vector of `comb_SA` may travel through all row units and be output a eliminated vector after $n$ cycles if it is not kept in any row unit. Each row unit has a special processor (denoted as `A`) in the pivot position of the processed matrix $M$. It is responsible for finding a non-zero pivot and sending commands of row operations to other processors (denoted as `B`) in the same unit. When a row unit has stored a pivot row, it simply eliminates all input vectors and outputs them. On the other hand, if the row unit has not yet found a vector with non-zero pivot, it either stores an input vector with a non-zero pivot element or passes the input to other row units.

As an additional note, it is known that for an $n \times n$ square matrix over $\mathbb{F}_q$, the probabilty it is invertible is $\frac{1}{q^{n^2}} \cdot \prod_{j=1}^{n}(q^n - q^{j-1})$. The authors in [19] are dealing with the systemization of a $768 \times 3488$ matrix over $\mathbb{F}_2$, and the probability it is "systemizable" (similar to "invertible" but for matrices that are not square, as is defined in [19]) is only 0.2888. Meanwhile, we are dealing with the matrices which are "systemizable" with probability 0.9336 over $\mathbb{F}_{16}$ and 0.9961 over $\mathbb{F}_{256}$. Since we are not facing the problem of frequent non-systemizable matrices, we decide to omit the design with the early-abort function support and adopt the design of [52] directly.

**Longer row vectors.** We reference the design in [52] as our base design and build other functional units on it. [52] uses a systolic line architecture to solve the system of linear equations. A systolic line is an architecture that allows signals to propagate through rows in a single clock cycle. To ensure the width of the systolic line does not become the critical path of the design, we implement a partially pipelined approach as illustrated in Figure 7. As shown in the figure, we divide the entire systolic array of processors into smaller, individual tiles. Each tile can be treated as a separate systolic line, allowing signals within a tile to propagate quickly and efficiently from left to right in one clock cycle. However, to avoid critical path issues, signals between tiles are pipelined. In conclusion, by combining elements of both the systolic array and systolic line, we can achieve a reduction in the number of clock cycles required for computation compared to a full systolic array, while also achieving higher frequency compared to using only a systolic line.

**Details of processors for Gaussian elimination over $\mathbb{F}_q$.** As shown in Figure 7, the systolic array is constructed using processors `processor_AB`, `processor_B`, `processor_ABC`, `processor_BC`, `processor_ABCD`, and `processor_BCD`. The suffixes A, B, C, and D denote the specific functionality. Functionality A and B are used in the Gaussian elimination process. Processors with functionality A are allocated in the diagonal line of the systolic array to find the pivot and triangularize the matrix. Functionality B, which receives signals from the left, helps in eliminating rows or swapping rows. Functionality C provides Single Instruction Multiple Data (SIMD) multiplication on columns of the Macaulay matrix. And functionality D is responsible for providing matrix-vector multiplication used in $\mathbf{O}x$. The processor with less functionality uses subsets of the signals of `processor_ABCD` or `processor_BCD`, and therefore requires fewer resources. By utilizing this method, the

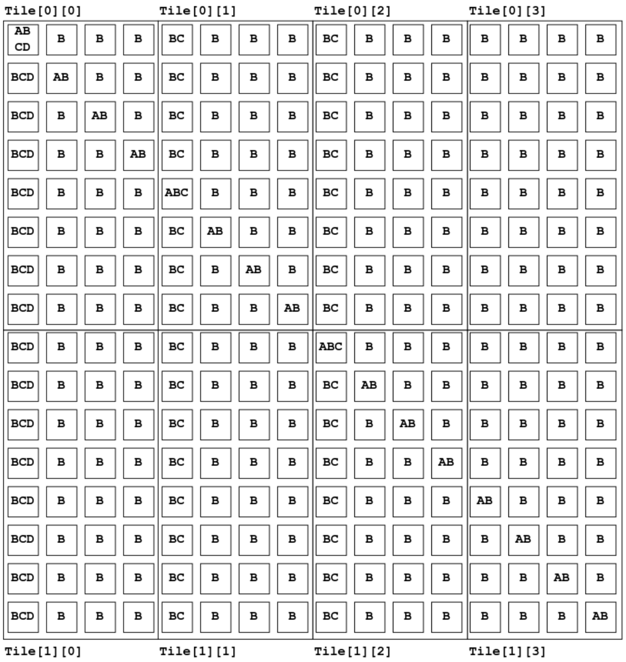



Figure 7: The Systolic Array performing Gaussian Elimination, SIMD multiplication, and/or matrix-vector multiplication. The figure shows the reference design for `ov-Ip`.

systolic array can reuse the multiplier and datapath, further reducing resource consumption.

Figure 8 shows the inputs and output signals of `processor_ABCD` and `processor_BCD`. We start by focusing on the signals related to functionality A and B. Since [52] only implements $\mathbb{F}_2$, we show the truth table for $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ in Table 17 and Table 18. We describe the two tables in the following:

- `start_in` is high at the beginning of the computation to find the pivots and triangularize the submatrix.
- For processors with functionality A, if `data_in` is 0 when `start_in` is high, `dataB_out` will be 1, so the processors to the right of it will keep the `data_in` in `data_r`.
- For processor with functionality A, if `data_in` is not 0 and `data_r` is 0, it finds

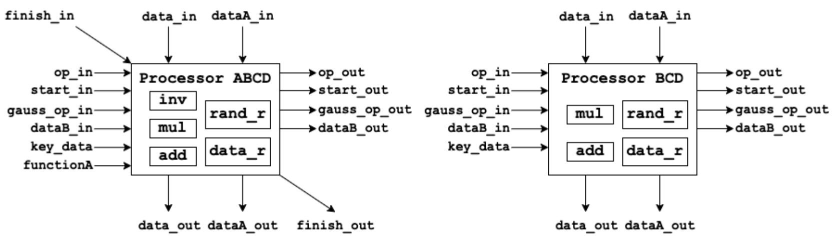



Figure 8: Input and output signals of `processor_ABCD` and `processor_BCD`. The processors multiplex inputs from the top and left, generate outputs to the bottom and right, and update their register states. Both processors possess `mul` and `add` units, which perform multiplication and addition in the fields $\mathbb{F}_{16}$ or $\mathbb{F}_{256}$. `processor_ABCD` has an additional `inv` unit, which provides the inverse of the data. Both `rand_r` and `data_r` are registers holding temporary data for the computation.

Table 17: Truth table for processor AB/ABC/ABCD.

| Input | | | State | | Output | | |
|---|---|---|---|---|---|---|---|
| `start_in` | `finish_in` | `data_in` | `data_r` | `data_r`$^+$ | `gauss_op_out` | `data_out` | `dataB_out` |
| 1 | 0 | $d$ | x | $d \cdot d^{-1}$ | start | 0 | $d^{-1}$ |
| 0 | 0 | 0 | `data_r` | `data_r` | pass | 0 | x |
| 0 | 0 | $d \neq 0$ | 0 | $d \cdot d^{-1}$ | swap | 0 | $d^{-1}$ |
| 0 | 0 | $d \neq 0$ | 1 | 1 | add | 0 | $d$ |
| 0 | 1 | x | x | x | swap | `data_r` | x |

Table 18: Truth table for processor B/BC/BCD.

| Input | | | | State | | Output |
|---|---|---|---|---|---|---|
| `start_in` | `finish_in` | `data_in` | `gauss_op_in` | `data_r` | `data_r`$^+$ | `data_out` |
| 1 | 0 | $d$ | x | x | $d \cdot$ `dataB_in` | 0 |
| 0 | 0 | $d$ | pass | `data_r` | `data_r` | $d$ |
| 0 | 0 | $d$ | swap | `data_r` | $d \cdot$ `dataB_in` | `data_r` |
| 0 | 0 | $d$ | add | `data_r` | `data_r` | $d +$ `data_r` $\cdot$ `dataB_in` |
| 0 | 1 | $d$ | swap | `data_r` | $d$ | `data_r` |

the pivot. It then issues `swap` and sets `dataB_out` to the inverse of `data_in`. For the processors to the right receiving operation `swap`, they will pass `data_r` out and update $r$ with $d \cdot$ `dataB_in`, which normalizes the row.

- For processor with functionality A, if `data_in` is not 0 and `data_r` is not 0, it will start forward elimination. It issues `add` and set `dataB_out` to `data_in`. For the processors to the right receiving operation `add`, they will do the elimination $d +$ `data_r` $\cdot$ `dataB_in`.
- For processor with functionality A, if `data_in` is 0, `data_in` is not pivot. It will issue `pass` to make the processors to the right fall through `data_in`.
- `finish_in` is high when the computation is finished. Processors with functionality A will issue `swap` to swap the result out.

We also follow the approach outlined in [52] to add additional control logic, allowing us to perform Gaussian elimination by eliminating submatrices.

**Field arithmetic.** The modules of addition, multiplication, and inversion of elements in $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ are often used in the design. The addition operation is simply implemented using a XOR circuit. For the multiplication, we use schoolbook multiplication and reduce the result using a combination of AND and XOR circuits. The synthesized result for $\mathbb{F}_{256}$ (AES field representation) requires 32 LUTs, which is only slightly more than the 29 LUTs required for multiplication in the tower field representation as it was used by Rainbow [22]. For the $\mathbb{F}_{16}$ field representation, the number of required LUTs is the same as that of the tower field representation, which is 7 LUTs. The logic delay in both addition and multiplication is typically negligible, and these operations are often combined to perform a multiply-and-accumulate operation, as illustrated in Figure 8. The inversion operation is implemented using a large look-up table. For $\mathbb{F}_{256}$, the LUT is 8-in-8-out and requires 40 LUTs in the synthesis, while for $\mathbb{F}_{16}$, it requires 2 LUTs.

**Supporting other functionalites with the same array.** Table 19 list the instructions related to the systolic array and their corresponding `op_in`. Both functionality C and D require an additional input `op_in`, indicating the operation it performs. Instructions with the same data flow have the same `op_in`. The processors `processor_A` and `processor_AB` do not use the `op_in` input, while processors with additional `C` or `CD` functionalities require

Table 19: `op_in` in systolic array.

| `op_in` | Instructions |
|---|---|
| 0 | Do nothing. |
| 1 | `gauss_elim` |
| 2 | `store_keys` |
| 3 | `load_keys` |
| 4 | `eval, calc_l, mul_key_o` |
| 5 | `send, unload_add_y, unload_check, store_l` |
| 6,7 | `mul_key_sig` |
| 8 | `add_to_sig_v` |
| 9 | `mul_o` |

3-bit or 4-bit input, respectively. Encoding `op_in` in this manner allows us to save resources on LUT.

Functionality C implements instructions operating on the column of the Macaulay matrix. For example, the instructions `eval`, `calc_l`, and `mul_key_o` multiply keys with multiplicands $\mathbf{s}_i \cdot \mathbf{s}_j$, $\mathbf{v}_i$, and element in $\mathbf{O}$, respectively. We use `dataB_in` to pass the multiplicands into processors. This allows us to reuse the datapath of `dataB_in` and multipliers in the processor, further saving resources. For the cases when `op_in` is 5, 6, or 7, these instructions also use `dataB_in` to shift in and out the data. For `load_keys` and `store_keys`, they load and store keys between local BRAMs and data registers in the systolic array.

Functionality D is activated when `op_in` is 8 or 9. It implementes instructions performing matrix-vector multiplication and vector-vector addition and uses an additional input `dataA_in`. We reuse only one column of multipliers here.

Lastly, we take `ov-Ip` in Figure 7 as an example. There are $16 \cdot 3 = 48$ processors with functionality C to perform SIMD multiplication on 44 rows of the Macaulay matrix. We can see that one column of multipliers, consisting of `processor_ABCD` and `processor_BCD`, is used to perform matrix-vector multiplication. For other parameter sets, the systolic arrays vary. In all parameters, there is one column of processors with functionality D. In particular, `ov-Is` features $32 \cdot 32$ systolic processors, as $32 \cdot 4 = 128$, and it is designed to seamlessly interface with AES-128. Consequently, `ov-III` and `ov-V` feature $16 \times 16$ systolic processors. Their differences are the number of columns with functionality C: `ov-Is` has $64/32 = 2$, `ov-III` has $\lceil 72/16 \rceil = 5$, and `ov-V` has $96/16 = 6$.

### 7.3.2 SHAKE-256 and AES-128

We integrate AES-128 and SHAKE-256 as separate modules outside the systolic array, as shown in Figure 7. We utilize the AES implementation from [27], which is using a fully-pipelined approach, and modify it to suit our needs. Since AES-128 is utilized only in CTR mode for sampling segments of the public key, we only include the encryption module, omitting the decryption module. We implement AES with two variations: (1) single-round AES, and (2) fully-pipelined AES with 10 rounds or 4 rounds (for the round-reduced implementation). The former requires 1 cycle per AES round. In total the 10-round AES requires 12 cycles due to an additional 2-cycle overhead for setting the counter and buffering the output. For the pipelined implementation one block of AES output is generated every cycle. Because of the mismatch between the output size of AES-128 (128 bits) and the column size of the Macaulay matrix ($m \cdot \log_2 |\mathbb{F}_q|$), we introduce a BRAM-based buffer to

Table 20: Resource utilization of AES-128 and SHAKE256 (only run synthesized design).

| | LUTs | FFs | BRAM | DSP |
|---|---|---|---|---|
| AES-128 (not pipelined) | 2 351 | 2 371 | 0 | 0 |
| AES-128 (pipelined 10-round) | 8 366 | 5 161 | 0 | 0 |
| AES-128 (pipelined 4-round) | 4 324 | 3 625 | 0 | 0 |
| SHAKE256 | 3 210 | 2 693 | 0 | 0 |

store the AES output. We then adjust the output from the buffer and pad zeroes if needed to ensure that the systolic array operates on the right column of the Macaulay matrix. Since the read and write ports of BRAM are independent, AES-128 and the systolic array can operate concurrently, resulting in a reduction of the number of cycles required.

For SHAKE-256, we adopt the mid-range hardware architecture by Keccak team [10] in our design. Each Keccak permutation takes 24 cycles to finish. It stores the output of a squeeze in the internal registers of the SHAKE-256 module. We also read out 1088-bit from the SHAKE-256 module and store it in the BRAM-based buffer for the generation of $\mathbf{O}$. This buffer is the same as the one used in the AES-128 module to save BRAM utilization. Therefore, AES and SHAKE instructions cannot execute simultaneously.

Table 20 shows the resource utilization of non-pipelined AES-128, pipelined AES-128 (10-round and 4-round), and SHAKE256. The non-pipelined AES-128 uses 2 351 LUTs. The pipelined AES, on the other hand, requires 1.8× more LUTs for the 4-round version and 3.6× more LUTs for the 10-round version compared to the non-pipelined version. When considering both area and performance, the 4-round AES is the best option among the AES designs. While the non-pipelined version requires fewer resources, it takes multiple cycles to generate a block. On the other hand, the 4-round AES offers similar throughput as the 10-round version, but with nearly half the resource usage.

### 7.3.3 The microcontroller and Two-Phase Decoding

The microcontroller, the top left component in Figure 6, is responsible for controlling the program flow and sending control signals to other components. It fetches the instruction from the instruction memory and decodes the instruction in its decoder. The decoder separates control and function instructions, which is the initial phase of decoding. Depending on the type of instructions, the microcontroller either executes control instructions or dispatches function instructions to other components.

The second phase of decoding takes place in the components implementing their specific functionalities. For example, the decoding in the systolic array receives input signals indicating the specific matrix operation to be performed.

## 7.4 Results of Implementation

In this section, we evaluate the FPGA design by measuring the resource utilization and cycle counts for key generation, signing, and verification. All of the designs are synthesized and done implementation with Xilinx Vivado 2022.1 edition. The designs for `ovIp` and `ovIs` are evaluated on Xilinx Zynq-7000 Z-7020 and `ovIII` and `ovV` are evaluated on XC7A200T. We set the target frequency to 100MHz for both.

We report the resource utilization for OV with non-pipelined AES and the cycle counts in full-round AES mode in Table 21. The utilization of LUTs and Slices of the variants with the same security level are similar, except `ov-Is` and `ov-V-pkc+skc`. Their requirements for key storage exceed the limit of the BRAM on their target boards, resulting in an increase in LUTs. The utilization of BRAMs is close to what we expect from Table 16.

Table 21: The FPGA results with full-round AES for our low-area (no pipelined AES) design.

| | Utilization | | | | | Cycle Count | | | Freq. (MHz) |
|---|---|---|---|---|---|---|---|---|---|
| | Slices | LUTs | FFs | BRAM | DSP | KeyGen | Sign | Verify | |
| `ov-Ip` | 12 145 | 33 221 | 24 097 | 108.5 | 2 | 3 540 971 | 7 515 | 6 435 | 93.5 |
| `ov-Ip-pkc` | 12 073 | 32 134 | 22 969 | 81 | 2 | 4 170 749 | 7 515 | 192 411 | 91.4 |
| `ov-Ip-pkc+skc` | 12 106 | 32 422 | 23 262 | 48 | 2 | 3 807 119 | 352 621 | 192 411 | 94.8 |
| `ov-Is` | 12 860 | 44 974 | 27 433 | 140 | 2 | 9 916 182 | 13 070 | 12 986 | 92.2 |
| `ov-Is-pkc` | 11 740 | 29 385 | 25 328 | 110 | 2 | 11 922 375 | 13 070 | 284 379 | 94.8 |
| `ov-Is-pkc+skc` | 11 681 | 28 947 | 24 444 | 66 | 2 | 11 072 933 | 843 885 | 284 379 | 90.8 |
| `ov-III-pkc` | 17 610 | 41 761 | 31 543 | 310.5 | 4 | 18 221 241 | 19 285 | 823 108 | 97.5 |
| `ov-III-pkc+skc` | 16 574 | 38 352 | 29 446 | 184.5 | 4 | 16 727 607 | 1 465 182 | 823 108 | 96.0 |
| `ov-V-pkc+skc` | 27 038 | 77 352 | 38 217 | 359 | 4 | 39 066 651 | 3 308 031 | 1 921 513 | 92.5 |

The utilization of DSP and FF resources is low.

We discuss the results in full-round AES mode first. The cycle count of signing in `classic` mode, can be broken down into individual steps as follows to provide an approximation of the cycle count:

| | |
|---|---|
| Prepare $\mathbf{v}$ | 66 for `ov-V` or 24 otherwise. |
| Prepare $\mathbf{y}$ | $(n-m+1)(n-m)/2$ |
| Calculate $\mathbf{t}-\mathbf{y}$ | 5 |
| Prepare $\mathbf{L}$ | $(n-m+13)m$ (13 for flow controls) |
| Solve $\mathbf{Lx}=\mathbf{t}-\mathbf{y}$ | $\Sigma_{i=0}^{(\lceil m/N\rceil-1)}(m+2N)(\lceil m/N\rceil+1-i)$, where $N=32$ for `ov-Is` or $N=16$ otherwise. |
| Calculate $\mathbf{Ox}$ | $(n-m)\lceil m/N\rceil$ |
| Calculate $\mathbf{v}+\mathbf{Ox}$ | 5 |

As an example of `ov-Ip`, where $n=112$ and $m=44$, the cycle count is $24+3\,564+5+2\,992+684+187+5=7\,461$ which is quite close ot our results.

The signing cycle count in `pkc+skc` mode is dominated by the **ExpandSK()** function, specifically, the calculation of the $\mathbf{S}_i=(\mathbf{P}_i^{(1)}+\mathbf{P}_i^{(1)\intercal})\mathbf{O}+\mathbf{P}_i^{(2)}$. This calculation takes $(n-m)\cdot m\cdot(n-m+15)$ cycles, where the 15 includes flow control and other operations such as loading from and storing to temporary storage. In the case of `ov-Ip-pkc+skc`, **ExpandSK()** takes 248 336 cycles. The remaining computation includes 7 515 cycles for tasks such as Gaussian elimination and polynomial evaluation, and 189 618 cycles for expanding $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$ from $\mathsf{seed}_{\mathsf{pk}}$. In the end, with savings from overlapping these computations, it results in $248\,336+7\,515+175\,032-352\,621=78\,262$ cycles in `ov-Ip-pkc+skc`.

The cycle count of verification in `classic` mode is approximately $n\times(n+1)/2$ cycles, which is consistent with 6 328 for `ov-Ip`. On the other hand, the cycle count of verification in `pkc` mode, is limited by the throughput of the $\mathsf{Expand}_\mathbf{P}$ function. The AES module of our low area design generates 128-bit every 12 cycles. To generate $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$, It takes $(\log_2|\mathbb{F}_q|\cdot m\cdot((n+m)(n-m)/2)/128)\cdot 12$ cycles, which is 175 032 in `ov-Ip-pkc`. The additional $192\,411-(175\,032+6\,435)=10\,944$ cycles come from waiting for the secret quadratic terms $\mathbf{s}_i^\intercal\mathbf{s}_j$ while evaluating key polynomials. Both key polynomials and quadratic terms connect to the systolic array with the same signal path. This cost is hidden in the case of non-pipelined AES.

We also report the cycle counts when using a 4-round AES for $\mathsf{Expand}_\mathbf{P}$. in Table 22. It shows a reduction in cycles for verification in `pkc` mode and signing in `skc`. The saving for verification matches our expectation, which can be estimated by the difference in rounds multiplied by the number of calls to the AES module. It is $(8\cdot 44\cdot((112+44)(112-$

Table 22: Results of OV with 4-round AES for our low-area design. The resource information is the same as that of full-round AES.

| | Cycle Count | | |
|---|---|---|---|
| | KeyGen | Sign | Verify |
| ov-Ip | 3 393 299 | 7 515 | 6 435 |
| ov-Ip-pkc | 4 077 245 | 7 515 | 99 615 |
| ov-Ip-pkc+skc | 3 768 047 | 313 549 | 99 615 |
| ov-Is | 9 746 742 | 13 070 | 12 986 |
| ov-Is-pkc | 11 814 183 | 13 070 | 176 859 |
| ov-Is-pkc+skc | 11 026 181 | 797 133 | 176 859 |
| ov-III-pkc | 17 832 117 | 19 285 | 436 036 |
| ov-III-pkc+skc | 16 556 211 | 1 293 786 | 436 036 |
| ov-V-pkc+skc | 38 671 211 | 2 909 727 | 1 015 155 |

Table 23: The performance results using pipelined AES.

| AES rounds | Scheme | Utilization | | | | | Cycle Count | | | Freq. (MHz) |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Slices | LUTs | FFs | BRAM | DSP | KeyGen | Sign | Verify | |
| 10 | ov-Ip-pkc | 12 850 | 37 438 | 25 449 | 81 | 2 | 4 049 016 | 7 515 | 61 499 | 89.5 |
| | ov-Ip-pkc+skc | 12 491 | 37 623 | 25 767 | 48 | 2 | 3 757 662 | 303 164 | 61 499 | 91.8 |
| | ov-Is-pkc | 12 482 | 35 786 | 27 856 | 110 | 2 | 11 773 796 | 13 070 | 115 258 | 95.5 |
| | ov-Is-pkc+skc | 12 259 | 34 208 | 26 974 | 66 | 2 | 11 008 802 | 779 754 | 115 258 | 90.3 |
| | ov-III-pkc | 19 612 | 48 068 | 33 997 | 310.5 | 4 | 17 619 070 | 19 285 | 195 651 | 93.7 |
| | ov-III-pkc+skc | 18 177 | 43 166 | 31 982 | 184.5 | 4 | 16 462 364 | 1 199 939 | 195 651 | 94.1 |
| | ov-V-pkc+skc | 28 357 | 83 444 | 40 597 | 359 | 4 | 38 404 186 | 2 645 566 | 364 198 | 92.6 |
| 4 | ov-Ip-pkc | 12 164 | 33 220 | 23 913 | 81 | 2 | 4 048 566 | 7 515 | 61 121 | 94.8 |
| | ov-Ip-pkc+skc | 11 911 | 33 363 | 24 233 | 48 | 2 | 3 757 428 | 302 930 | 61 121 | 94.5 |
| | ov-Is-pkc | 11 958 | 31 227 | 26 327 | 110 | 2 | 11 772 350 | 13 070 | 113 914 | 94.2 |
| | ov-Is-pkc+skc | 11 845 | 31 006 | 25 444 | 66 | 2 | 11 008 124 | 779 076 | 113 914 | 92.4 |
| | ov-III-pkc | 18 323 | 43 408 | 32 439 | 310.5 | 4 | 17 617 420 | 19 285 | 194 115 | 96.3 |
| | ov-III-pkc+skc | 17 084 | 39 003 | 30 516 | 184.5 | 4 | 16 461 578 | 1 199 153 | 194 115 | 96.9 |
| | ov-V-pkc+skc | 27 753 | 79 918 | 39 206 | 359 | 4 | 38 403 352 | 2 644 732 | 362 626 | 95.7 |

$44)/2)/128) \cdot 6 = 87\,516$ cycles in the case of `ov-Ip`. For signing in `skc` variants, the saving is less significant because computing the $\mathbf{S}_i$ in **ExpandSK()** dominates the cycle count.

Finally, we present the results for our high-performance design using a fully pipelined AES in Table 23. We show only the results for `pkc` and `pkc+skc` as only those are majorly affected in signing and verification by the faster AES. Comparing to the results using the no-pipelined AES, verification improves by a factor of 3. As AES now generates one block per cycle, it requires $(8 \cdot 44 \cdot ((112+44)(112-44)/2)/128) = 14\,586$ cycles to generate $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$. The overhead $61\,499 - (14\,586 + 6\,435) = 40\,478$ cycles comes again from waiting for quadratic terms $\mathbf{s}_i^{\mathsf{T}}\mathbf{s}_j$. For the signing in `pkc+skc`, the cycle count slightly improves since the bottleneck is the computation of the $\mathbf{S}_i$. The cycles for 4-round and 10-round AES are similar since both are pipelined, generating 128-bits per cycle.

The utilization of LUTs and FFs for pipelined AES increases as discussed in Subsubsection 7.3.2. For the case of `ov-Ip-pkc+skc`, the pipelined versions use 16% and 3% more LUTs than the non-pipelined version for 10- and 4-round AES, respectively.

## Acknowledgements

Academia Sinica authors were supported by the Taiwan Ministry of Science and Technology through grant 109-2221-E-001-009-MY3, Academia Sinica Investigator Award AS-IA-109-

M01, and the Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

## References

[1] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-M4. In Giuseppe Ateniese and Daniele Venturi, editors, *ACNS 22*, volume 13269 of *LNCS*, pages 853–871, Rome, Italy, June 20–23, 2022. Springer, Heidelberg, Germany.

[2] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like ciphers. *IACR TCHES*, 2021(1):402–425, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8739.

[3] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone, and Yi-Kai Liu. Nistir 8413, status report on the third round of the nist post-quantum cryptography standardization process, September 2022.

[4] Apon, Daniel. NIST assignments of platforms on implementation efforts to PQC teams. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/cJxMq0_90gU/m/qbGEs3TXGwAJ. [Online 7-February-2019; accessed 12-October-2022].

[5] Arm. Arm intrinsics, 2022. https://developer.arm.com/architectures/instruction-sets/intrinsics/.

[6] Thomas Aulbach, Fabio Campos, Juliane Krämer, Simona Samardjiska, and Marc Stöttinger. Separating oil and vinegar with a single trace. Cryptology ePrint Archive, Report 2023/335, 2023. https://eprint.iacr.org/2023/335.

[7] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR TCHES*, 2022(1):221–244, 2022. https://tches.iacr.org/index.php/TCHES/article/view/9295.

[8] Daniel J Bernstein and Tanja Lange. ebacs: Ecrypt benchmarking of cryptographic systems, 2009. https://bench.cr.yp.to, accessed February 13, 2022.

[9] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR TCHES*, 2019(3):340–398, 2019. https://tches.iacr.org/index.php/TCHES/article/view/8298.

[10] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche1, and Ronny Van Keer. Keccak in VHDL, 2015. https://keccak.team/hardware.html.

[11] Ward Beullens. Improved cryptanalysis of UOV and Rainbow. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part I*, volume 12696 of *LNCS*, pages 348–373, Zagreb, Croatia, October 17–21, 2021. Springer, Heidelberg, Germany.

[12] Ward Beullens. Breaking rainbow takes a weekend on a laptop. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 464–479, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.

[13] Ard Biesheuvel. Accelerated aes for arm64 linux kernel, 2017. https://www.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/.

[14] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir, and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In Stefan Mangard and François-Xavier Standaert, editors, *CHES 2010*, volume 6225 of *LNCS*, pages 203–218, Santa Barbara, CA, USA, August 17–20, 2010. Springer, Heidelberg, Germany.

[15] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin Heidelberg, 2010.

[16] 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 2009*, volume 5747 of *LNCS*, pages 33–48, Lausanne, Switzerland, September 6–9, 2009. Springer, Heidelberg, Germany.

[17] Ming-Shing Chen and Tung Chou. Classic mceliece on the ARM cortex-M4. *IACR TCHES*, 2021(3):125–148, 2021. https://tches.iacr.org/index.php/TCHES/article/view/8970.

[18] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang, and Chen-Mou Cheng. Implementing 128-bit secure MPKC signatures. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 101-A(3):553–569, 2018.

[19] Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang. Complete and improved FPGA implementation of Classic McEliece. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):71–113, Jun. 2022. Fixed version with errata in TCHES version available at https://eprint.iacr.org/2022/412.

[20] Tung Chou, Matthias J. Kannwischer, and Bo-Yin Yang. Rainbow on cortex-M4. *IACR TCHES*, 2021(4):650–675, 2021. https://tches.iacr.org/index.php/TCHES/article/view/9078.

[21] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. eXtended Keccak Code Package. https://github.com/XKCP/XKCP.

[22] Jingtai Ding, Ming-Shing Chen, Matthias Julias Kannwischer, Albrecht Petzoldt, Jacques Patarin, Dieter Schmidt, and Bo-Yin Yang. Rainbow 3rd round submission, nist submission document and technical report, October 2020.

[23] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 164–175, New York, NY, USA, June 7–10, 2005. Springer, Heidelberg, Germany.

[24] Nir Drucker and Shay Gueron. Speed up over the rainbow. Cryptology ePrint Archive, Report 2020/408, 2020. https://eprint.iacr.org/2020/408.

[25] Shay Gueron. Intel advanced encryption standard (aes) new instructions set, 2010. https://www.intel.com.bo/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.

[26] B. Hochet, P. Quinton, and Y. Robert. Systolic gaussian elimination over GF(p) with partial pivoting. *IEEE Transactions on Computers*, 38(9):1321–1324, 1989.

[27] Hsing Homer. Tiny aes, 2012. https://opencores.org/projects/tiny_aes.

[28] Andreas Hulsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kolbl, Tanja Lange, Martin M Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan, and Ward Beullens. SPHINCS+. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[29] Intel. Intel intrinsics guide, 2022. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[30] Dougall Johnson. Apple m1 microarchitecture research. https://dougallj.github.io/applecpu/firestorm.html.

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

[32] Liam Keliher and Jiayuan Sui. Exact maximum expected differential and linear probability for two-round advanced encryption standard. *IET Inf. Secur.*, 1(2):53–57, 2007.

[33] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced Oil and Vinegar signature schemes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 206–222, Prague, Czech Republic, May 2–6, 1999. Springer, Heidelberg, Germany.

[34] Aviad Kipnis and Adi Shamir. Cryptanalysis of the Oil & Vinegar signature scheme. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 257–266, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[35] H.T. Kung and Leiserson. *Systolic Arrays for (VLSI)*. CMU-CS. Carnegie-Mellon University, Department of Computer Science, 1978.

[36] Arm Ltd. Arm cortex-m programming guide to memory barrier instructions, 2012. https://developer.arm.com/documentation/dai0321/latest.

[37] Arm Ltd. Arm cortex-a72 software optimization guide, 2015. https://developer.arm.com/documentation/uan0016/a/.

[38] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[39] Duc Tri Nguyen and Kris Gaj. Fast falcon signature generation and verification using armv8 neon instructions, 2022. https://csrc.nist.gov/csrc/media/Events/2022/fourth-pqc-standardization-conference/documents/papers/fast-falcon-signature-generation-and-verification-pqc2022.pdf.

[40] FIPS PUB 197 – advanced encryption standard (AES), 2001. https://doi.org/10.6028/NIST.FIPS.197.

[41] FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. https://doi.org/10.6028/NIST.FIPS.202.

[42] NIST. Call for additional digital signature schemes for the post-quantum cryptography standardization process, September 2022.

[43] Jacques Patarin. The Oil and Vinegar signature scheme. In *Dagstuhl Workshop on Cryptography*, September 1997.

[44] 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 2010*, volume 6498 of *LNCS*, pages 33–48, Hyderabad, India, December 12–15, 2010. Springer, Heidelberg, Germany.

[45] Albrecht Petzoldt, Enrico Thomae, Stanislav Bulygin, and Christopher Wolf. Small public keys and fast verification for $\mathcal{M}$ultivariate $\mathcal{Q}$uadratic public key systems. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 475–490, Nara, Japan, September 28 – October 1, 2011. Springer, Heidelberg, Germany.

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

[47] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 180–194, St. John's, NL, Canada, August 10–12, 2016. Springer, Heidelberg, Germany.

[48] Kyung-Ah Shim, Sangyub Lee, and Namhun Koo. Efficient implementations of Rainbow and UOV using AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):245–269, Nov. 2021. https://tches.iacr.org/index.php/TCHES/article/view/9296.

[49] Chengdong Tao, Albrecht Petzoldt, and Jintai Ding. Efficient key recovery for all HFE signature variants. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part I*, volume 12825 of *LNCS*, pages 70–93, Virtual Event, August 16–20, 2021. Springer, Heidelberg, Germany.

[50] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 156–171, Darmstadt, Germany, May 21–23, 2012. Springer, Heidelberg, Germany.

[51] Paul C. van Oorschot and Michael J. Wiener. Improving implementable meet-in-the-middle attacks by orders of magnitude. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 229–236, Santa Barbara, CA, USA, August 18–22, 1996. Springer, Heidelberg, Germany.

[52] Wen Wang, Jakub Szefer, and Ruben Niederhagen. Solving large systems of linear equations over GF(2) on fpgas. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Nov. 2016.

[53] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based key generator for the Niederreiter cryptosystem using binary goppa codes. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 253–274, Cham, 2017. Springer International Publishing.

[54] Xilinx, Inc. *XMP100: Cost-Optimized Portfolio Product Selection Guide*, 2.1 edition, November 2022. https://docs.xilinx.com/v/u/en-US/cost-optimized-product-selection-guide.