PROV: PRovable unbalanced Oil and Vinegar Specification v1.2 – 22/04/2024

Principal submitter: Louis Goubin
University of Versailles-St-Quentin-en-Yvelines
LMV Laboratory
45 avenue des Etats-Unis
FR-78035 Versailles Cedex
France
Louis.Goubin@uvsq.fr
phone: +33 1 39 25 43 29

Auxiliary submitters: Benoît Cogliati
Jean-Charles Faugère
Pierre-Alain Fouque
Robin Larrieu
Gilles Macario-Rat
Brice Minaud
Jacques Patarin
Jocelyn Ryckeghem

Inventors of the cryptosystem: The submitters
Relevant prior work is credited below where appropriate

Owner of the cryptosystem: Same as the submitters

Signature: See independent cover sheet.

Website: prov-sign.github.io

Contents

1 Introduction

PROV is a multivariate cryptography-based signature scheme, and its name stands for PRovable unbalanced Oil-and-Vinegar. As many attacks on Multivariate Cryptography have been published, the confidence in this alternative has been undermined. Consequently, we think it is highly important to support such schemes with a security proof. Since the introduction of UOV, some security proofs appeared at PQCrypto 2011 by Sakumoto *et al*. [SSH11], and more recently by Kosuge and Xagawa [KX24], who also provide a proof in the QROM. Here, we use another proof, which is reminiscent of the security proof of the MAYO signature scheme due to Beullens [Beu22]. The idea is to have a larger oil space than the output of the scheme. This variant is sometimes called UOV$^-$: it corresponds to the UOV scheme where some public equations have been removed. This classical transformation is known as the "Minus" method [Pat96] in the literature on Multivariate Cryptography.

1.1 The UOV signature scheme

The Unbalanced Oil-and-Vinegar has been proposed by Kipnis, Patarin and Goubin in [KPG99] about 25 years ago. It is a hash-and-sign signature scheme that follows the GPV framework [GPV08] and its adaptation to multivariate cryptography in [KX24]. From a high level, the UOV algorithm works with two sets of variables: o oil and v vinegar variables. The secret key consists in a tuple of o random quadratic forms $\mathcal{Q}$ that does not involve any product between oil variables. This structure is hidden using a secret linear map $\mathbf{T}$ that mixes oil and vinegar variables. The public key $\mathcal{P}$ is then the composition $\mathcal{Q} \circ T$. The signing procedure of a message msg is very simple: once the vinegar variables are randomly fixed to some vector $\mathbf{v}$, the system $\mathcal{Q}(\mathbf{v}, \mathbf{o}) = \mathsf{hash}(\mathsf{msg})$ becomes linear, and thus easy to solve. In the case where the linear system has no solution, the algorithm simply restarts from the beginning.

1.2 Multivariate Cryptography (MQ)

The main advantage of MQ is to propose very short signature, despite the public key size is quite large, compared to lattice-based schemes or MPC-in-the-head signatures. In this family, the most secure signature scheme is the UOV scheme, which resists to attacks. However, as many attacks have undermined the confidence in MQ, we propose a security proof for PROV.

1.3 Design rationale

Our goal in this document is to propose a provably secure variant of UOV. In [SSH11], the authors present an UOV variant, dubbed SaltedUOV, whose EUF-CMA-security can be directly linked to the probability of inverting an UOV public key. The main difference with the standard UOV construction is the use of a salt that is hashed alongside the message. In the case where, during the signing procedure, the linear system of equations does not admit any solution, the salt will be resampled instead of the vinegar variables. Even though this simple change makes the UOV scheme provably secure, it can have significant drawbacks. Indeed, the running time of the signature algorithm is now directly linked to the rank of the linear system of equations implied by the choice of $\mathbf{v}$.

With PROV, our goal is to alleviate this problem. In order to do so, we increase the number of oil variables beyond the number of public quadratic forms in such a way that the rank of the

linear system of equations will be full with overwhelming probability[1]. This fact and our choice of parameters ensure that the signature algorithm is unlikely to ever need more than one iteration in order to output a signature.

We can summarize our design rationale as follows:

- **simplicity**: one of the main advantages of the UOV family of signature schemes is its simplicity; the algorithms are easy to describe, understand and implement;
- **provable security**: PROV can be proven secure both in the classical and quantum Random Oracle Model, and our choice of parameters is guided by the bound;
- **signature size**: multivariate cryptography is a good candidate for short signature schemes, and PROV is no exception; it is important to note that we make some concession on the signature size in order to attain provable security;
- **reasonable public key size**: we implement well-known optimizations to reduce the public key size, which is arguably one of the weak points of multivariate cryptography;
- **security beyond unforgeability**: we incorporate a simple design tweak based on the BUFF construction [CDF+21] in order to provide several advanced security guarantees (we refer the reader to Section 3.2 for more information).

1.4 Organization of this document

Section 2 is a complete specification of PROV. Section 3 discusses the security of PROV, and describes our parameter set. Section 4 deals with implementation issues and possible optimizations. Finally, Section 5 concludes by presenting advantages and limitations of PROV.

1.5 Known Answer Test values

Known answer test values (KAT) for PROV 1.2 are available on the PROV website:

prov-sign.github.io

2 Specification of PROV

2.1 Preliminaries and notations

Let $\mathbb{F}$ be the Galois field $\mathsf{GF}(2^8)$ with the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. We denote with bold lower-case letters vectors of elements of $\mathbb{F}$. Unless otherwise stated, these will be column vectors. Matrices will be denoted with bold upper-case letters. For any vector $\mathbf{v}$ or matrix $\mathbf{M}$, we denote with $\mathbf{v}^\mathsf{T}$ and $\mathbf{M}^\mathsf{T}$ their respective transpose.

Let $\mathcal{X}$ and $\mathcal{Y}$ be two sets, and let F be a function from $\mathcal{X}$ to $\mathcal{Y}$. We denote by $\mathsf{Img}(F)$ the image of F, i.e. the set of $y \in \mathcal{Y}$ such that there exists $x \in \mathcal{X}$ satisfying $F(x) = y$. When $\mathcal{X}$ is finite, we denote with $x \leftarrow_\$ \mathcal{X}$ the sampling of x uniformly at random in $\mathcal{X}$.

[1] We can thus see PROV as a specific instance of SaltedUOV$^-$.

2.2 Parameter space

The main parameters involved in PROV are:

- λ the security parameter of PROV,
- m the number of equations in the public key,
- n the total number of variables,
- δ the difference between the number of oil variables and the number m of equations in the public key,
- $\mathsf{len}_{\mathsf{s_{pk}}}$ the length of the public key seed in bits,
- $\mathsf{len}_{\mathsf{s_{sk}}}$ the length of the private key seed in bits,
- $\mathsf{len}_{\mathsf{salt}}$ the length of salts in bits,
- $\mathcal{H} : \{0,1\}^* \to \mathbb{F}^{\mathsf{m}}$ a hash function,
- $\mathcal{H}' : \{0,1\}^* \to \{0,1\}^{\mathsf{len}_{\mathsf{hpk}}}$ a hash function,
- an Extendable-Output Function Expand that takes as input a bitstring $M \in \{0,1\}^*$ and a non-negative integer d, and outputs a pseudo-random bitstring of length d. This will be used for seed expansion. To indicate domain separation, a subscript is added: $\mathsf{Expand}_{\mathbf{O}}$, $\mathsf{Expand}_{\mathsf{pk}}$ are two independent functions with the previous properties.

2.3 Key generation

We adopt the description of the UOV algorithm due to Beullens et al. [Beu21, BCH+23]. It relies on an alternative key generation algorithm that compresses public keys without degrading the security of the scheme that was developed in [PBB10, BPB10, PTBW11]. From a high level, the public key consists in a multivariate quadratic map $\mathcal{P} : \mathbb{F}^{\mathsf{n}} \to \mathbb{F}^{\mathsf{m}}$ that is identically zero on a secret $(\mathsf{m}+\delta)$-dimensional vector subspace $O \subset \mathbb{F}^{\mathsf{n}}$:

$$\forall \mathbf{o} \in O, \mathcal{P}(\mathbf{o}) = 0. \tag{1}$$

The key generation algorithm starts with the choice of O under the form of a matrix $\begin{pmatrix} \mathbf{O}^{\mathsf{T}} & \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix}^{\mathsf{T}}$ whose columns form a basis of O. The matrix $\mathbf{O} = \mathsf{Expand}_{\mathbf{O}}(\mathsf{s_{sk}}) \in \mathbb{F}^{(\mathsf{n}-\mathsf{m}-\delta)\times(\mathsf{m}+\delta)}$ is generated by deterministically expanding a uniformly random secret key seed $\mathsf{s_{sk}}$ of length $\mathsf{len}_{\mathsf{s_{sk}}}$. The next step consists in the sampling of the public quadratic map $\mathcal{P} = (p_1, \ldots, p_{\mathsf{m}})$. Each quadratic form p_i can be uniquely represented by an upper triangular matrix $\mathbf{P}_i$ such that

$$\forall \mathbf{x} \in \mathbb{F}^{\mathsf{n}}, p_i(\mathbf{x}) = \mathbf{x}^{\mathsf{T}} \mathbf{P}_i \mathbf{x}.$$

Each matrix $\mathbf{P}_i$ will be decomposed into three blocks $\mathbf{P}_i^1 \in \mathbb{F}^{(\mathsf{n}-\mathsf{m}-\delta)\times(\mathsf{n}-\mathsf{m}-\delta)}$, $\mathbf{P}_i^2 \in \mathbb{F}^{(\mathsf{n}-\mathsf{m}-\delta)\times(\mathsf{m}+\delta)}$, and $\mathbf{P}_i^3 \in \mathbb{F}^{(\mathsf{m}+\delta)\times(\mathsf{m}+\delta)}$, such that

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

where both $\mathbf{P}_i^1$ and $\mathbf{P}_i^3$ are upper triangular. Condition (1) amounts to the fact that the following matrix:

$$\begin{pmatrix} \mathbf{O}^\mathsf{T} & \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix} \mathbf{P}_i \begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix} = \mathbf{O}^\mathsf{T}\mathbf{P}_i^1\mathbf{O} + \mathbf{O}^\mathsf{T}\mathbf{P}_i^2 + \mathbf{P}_i^3$$

is symmetric, and its diagonal coefficients are 0[2]. Once $\mathbf{P}_i^1$ and $\mathbf{P}_i^2$ are fixed, this condition uniquely determines $\mathbf{P}_i^3$ as this matrix is upper triangular. Hence, we can simply derive the coefficients of $\mathbf{P}_i^1$ and $\mathbf{P}_i^2$ deterministically from a uniformly random public seed $\mathsf{s_{pk}}$ of length $\mathsf{len_{s_{pk}}}$, and then fix $\mathbf{P}_i^3 = \mathsf{Sym}(-\mathbf{O}^\mathsf{T}\mathbf{P}_i^1\mathbf{O} - \mathbf{O}^\mathsf{T}\mathbf{P}_i^2)$, where $\mathsf{Sym}(\mathbf{M})$ is the unique upper triangular matrix such that $\mathbf{M} + \mathsf{Sym}(\mathbf{M})$ is symmetric and its diagonal coefficients are 0.

- The **public key** of PROV is $\mathsf{pk} = (\{\mathbf{P}_i^3\}_{i=1,\dots,\mathsf{m}}, \mathsf{s_{pk}}, \mathsf{hpk})$, where $\mathsf{hpk} = \mathcal{H}(\{\mathbf{P}_i^3\}_{i=1,\dots,\mathsf{m}} \| \mathsf{s_{pk}})$ stands for *hashed public key*, and is included for performance reasons. The symbol $\|$ denotes concatenation.
- The **secret key** of PROV is $\mathsf{sk} = (\mathsf{hpk}, \mathsf{s_{sk}})$.

In the following section, we will also describe an expanded private key that will make the signature computation more efficient, in exchange for a larger key size. Overall, the size in bits of a PROV public key is:

$$\frac{\mathsf{m}(\mathsf{m}+\delta)(\mathsf{m}+\delta+1)}{2}\lceil \log_2(|\mathbb{F}|)\rceil + \mathsf{len_{s_{pk}}} + \mathsf{len_{hpk}}.$$

The size in bits of a secret key is:

$$\mathsf{len_{s_{sk}}} + \mathsf{len_{hpk}}.$$

A pseudocode description of key generation can be found in Algorithm 1.

Algorithm 1 Key generation algorithm.

procedure KEYGENERATION
  $\mathsf{s_{sk}} \leftarrow_\$ \{0,1\}^{\mathsf{len_{s_{sk}}}}$
  $\mathsf{s_{pk}} \leftarrow_\$ \{0,1\}^{\mathsf{len_{s_{pk}}}}$
  $\mathbf{O} \leftarrow \mathsf{Expand}_{\mathbf{O}}(\mathsf{s_{sk}})$
  $(\mathbf{P}_i^1, \mathbf{P}_i^2)_{i=1,\dots,\mathsf{m}} \leftarrow \mathsf{Expand}_{\mathsf{pk}}(\mathsf{s_{pk}})$
  for $i = 1$ to m **do**
    $\mathbf{P}_i^3 \leftarrow \mathsf{Sym}(-\mathbf{O}^\mathsf{T}\mathbf{P}_i^1\mathbf{O} - \mathbf{O}^\mathsf{T}\mathbf{P}_i^2)$
  $\mathsf{pk} \leftarrow (\mathsf{s_{pk}}, (\mathbf{P}_i^3)_{i=1,\dots,\mathsf{m}})$
  $\mathsf{hpk} \leftarrow \mathcal{H}'(\mathsf{pk})$
  $\mathsf{sk} \leftarrow (\mathsf{s_{pk}}, \mathsf{s_{sk}}, \mathsf{hpk})$
  return $(\mathsf{pk}, \mathsf{sk})$

2.4 Signature computation

Let $\mathsf{msg} \in \{0,1\}^*$ be a message to be signed. The goal of the signature procedure is to find a vector $\mathbf{s} \in \mathbb{F}^\mathsf{n}$ such that $\mathcal{P}(\mathbf{s}) = \mathcal{H}(\mathsf{hpk}||\mathsf{msg}||\mathsf{salt})$, where salt is a bitstring of length $\mathsf{len_{salt}}$. It will be

[2]Recall that $\mathbb{F}$ is a field of characteristic 2. This would otherwise correspond to the matrix being skew-symmetric.

computed as

$$\mathbf{s} = \begin{pmatrix} \mathbf{v} \\ 0 \end{pmatrix} + \begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix} \mathbf{o} \tag{2}$$

with $\mathbf{v} \in \mathbb{F}^{\mathsf{n}-\mathsf{m}-\delta}$, and $\mathbf{o} \in \mathbb{F}^{\mathsf{m}+\delta}$. The signature will then consist in the pair $(\mathsf{salt}, \mathbf{s})$, whose length in bits is

$$\mathsf{n}\lceil \log_2(|\mathbb{F}|) \rceil + \mathsf{len}_{\mathsf{salt}}.$$

Note that, for $i = 1, \ldots, \mathsf{m}$, one has

$$p_i(\mathbf{s}) = p_i\left(\begin{pmatrix} \mathbf{v} \\ 0 \end{pmatrix}\right) + \begin{pmatrix} \mathbf{v}^\mathsf{T} & 0 \end{pmatrix} (\mathbf{P}_i + \mathbf{P}_i^\mathsf{T}) \begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix} \mathbf{o} + p_i\left(\begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix} \mathbf{o}\right),$$

with $p_i\left(\begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix} \mathbf{o}\right) = 0$ by construction. Hence, one gets

$$p_i(\mathbf{s}) = \mathbf{v}^\mathsf{T} \mathbf{P}_i^1 \mathbf{v} + \mathbf{v}^\mathsf{T} \left(\left(\mathbf{P}_i^1 + \mathbf{P}_i^{1\,\mathsf{T}} \right) \mathbf{O} + \mathbf{P}_i^2 \right) \mathbf{o}.$$

Overall, once $\mathbf{v}$ and salt have been fixed, computing the signature $\mathbf{s}$ of the message m corresponds to solving the following system of linear equations:

$$\begin{cases} \mathbf{v}^\mathsf{T} \left(\left(\mathbf{P}_1^1 + \mathbf{P}_1^{1\,\mathsf{T}} \right) \mathbf{O} + \mathbf{P}_1^2 \right) \mathbf{o} &= h_1 - \mathbf{v}^\mathsf{T} \mathbf{P}_1^1 \mathbf{v} \\ & \vdots \\ \mathbf{v}^\mathsf{T} \left(\left(\mathbf{P}_\mathsf{m}^1 + \mathbf{P}_\mathsf{m}^{1\,\mathsf{T}} \right) \mathbf{O} + \mathbf{P}_\mathsf{m}^2 \right) \mathbf{o} &= h_\mathsf{m} - \mathbf{v}^\mathsf{T} \mathbf{P}_\mathsf{m}^1 \mathbf{v}, \end{cases} \tag{3}$$

with $(h_1, \ldots, h_\mathsf{m}) = \mathcal{H}(\mathsf{hpk}||\mathsf{msg}||\mathsf{salt}) \in \mathbb{F}^\mathsf{m}$. This process can be repeated by sampling new salt values until the system (3) admits solutions. Then, a vector $\mathbf{o}$ can be chosen uniformly at random from the set of all solutions of (3). Note that, in order to speed up the computation, it is possible to store the secret matrices $\mathbf{S}_i = \left(\mathbf{P}_i^1 + \mathbf{P}_i^{1\,\mathsf{T}} \right) \mathbf{O} + \mathbf{P}_i^2$ alongside the secret seed s_{sk}. We refer to $\mathsf{esk} = ((\mathbf{S}_i)_{i=1,\ldots,m}, \mathsf{s}_{\mathsf{pk}}, \mathsf{sk})$ as the *expanded* secret key esk, whose size in bits is

$$\mathsf{m}(\mathsf{n} - \mathsf{m} - \delta)(\mathsf{m} + \delta)\lceil \log_2(|\mathbb{F}|) \rceil + \mathsf{len}_{\mathsf{s}_{\mathsf{pk}}} + \mathsf{len}_{\mathsf{hpk}} + \mathsf{len}_{\mathsf{s}_{\mathsf{sk}}}.$$

A pseudocode description of this step can be found in Algorithm 2.

Algorithm 2 Signature algorithm.

procedure SIGN($\mathsf{sk}, \mathsf{msg}$)
  $(\mathsf{s_{pk}}, \mathsf{s_{sk}}, \mathsf{hpk}) \leftarrow \mathsf{sk}$
  $\mathbf{O} \leftarrow \mathsf{Expand}_{\mathbf{O}}(\mathsf{s_{sk}})$
  $(\mathbf{P}_i^1, \mathbf{P}_i^2)_{i=1,\ldots,\mathsf{m}} \leftarrow \mathsf{Expand}_{\mathsf{pk}}(\mathsf{s_{pk}})$
  $\mathbf{v} \leftarrow_{\$} \mathbb{F}^{\mathsf{n}-\mathsf{m}-\delta}$
  repeat
    $\mathsf{salt} \leftarrow_{\$} \{0,1\}^{\mathsf{len_{salt}}}$
    $(h_1, \ldots, h_{\mathsf{m}}) \leftarrow \mathcal{H}(\mathsf{hpk}||\mathsf{msg}||\mathsf{salt})$
    for $i = 1$ to m **do**
      $t_i \leftarrow h_i - \mathbf{v}^\mathsf{T}\mathbf{P}_i^1\mathbf{v}$
      $\mathbf{a}_i \leftarrow \mathbf{v}^\mathsf{T}\left(\left(\mathbf{P}_i^1 + \mathbf{P}_i^{1\,\mathsf{T}}\right)\mathbf{O} + \mathbf{P}_i^2\right)$
    $\mathbf{A_v} \leftarrow (\mathbf{a}_i)_{i=1,\ldots,\mathsf{m}}$
    $\mathbf{t} \leftarrow (t_i)_{i=1,\ldots,\mathsf{m}}$
    $S \leftarrow \mathsf{LinSolve}(\mathbf{A_v}, \mathbf{t})$
  until $S \neq \emptyset$
  $\mathbf{o} \leftarrow_{\$} S$
  $\mathbf{s} \leftarrow \begin{pmatrix} \mathbf{v} \\ 0 \end{pmatrix} + \begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix} \mathbf{o}$
  return $(\mathsf{salt}, \mathbf{s})$

2.5 Signature verification

The signature verification simply consists in verifying that the equation $\mathcal{P}(\mathbf{s}) = \mathcal{H}(\mathsf{hpk}||\mathsf{msg}||\mathsf{salt})$ holds. A pseudocode description of this step can be found in Algorithm 3.

Algorithm 3 Signature verification.

procedure VERIFY($\mathsf{pk}, \mathsf{msg}, \mathsf{sig}$)
  $(\mathsf{s_{pk}}, (\mathbf{P}_i^3)_{i=1,\ldots,\mathsf{m}}) \leftarrow \mathsf{pk}$
  $(\mathbf{P}_i^1, \mathbf{P}_i^2)_{i=1,\ldots,\mathsf{m}} \leftarrow \mathsf{Expand}_{\mathsf{pk}}(\mathsf{s_{pk}})$
  $(\mathsf{salt}, \mathbf{s}) \leftarrow \mathsf{sig}$
  $\mathsf{hpk} \leftarrow \mathcal{H}'(\mathsf{pk})$
  $\mathbf{h} \leftarrow \mathcal{H}(\mathsf{hpk}||\mathsf{msg}||\mathsf{salt})$
  for $i = 1$ to m **do**
    $\mathbf{P}_i \leftarrow \begin{pmatrix} \mathbf{P}_i^1 & \mathbf{P}_i^2 \\ \mathbf{0} & \mathbf{P}_i^3 \end{pmatrix}$
    $t_i \leftarrow \mathbf{s}^\mathsf{T}\mathbf{P}_i\mathbf{s}$
  return $(t_i)_{i=1,\ldots,\mathsf{m}} \stackrel{?}{=} \mathbf{h}$

2.6 Determinization

For ease of exposition, the above description presents a variant of PROV where various quantities are sampled uniformly at random. In reality, all randomness is ultimately derived from the secret seed and the message. In particular, PROV signatures are deterministic: the signature of a given

message is always the same. We now explain in detail how the previous construction is made deterministic.

2.6.1 Seed expansion

Public and secret seed expansion are computed using AES in counter mode (AES-CTR). In more details, for (PROV-I, PROV-III, PROV-V), the public seed $\mathsf{s_{pk}}$ (resp. secret seed $\mathsf{s_{sk}}$) is of length $(128, 192, 256)$ bits respectively. It is used as key for (AES128, AES192, AES256), respectively. In the following explanation, byte order for the initialization vector (IV) follows the NIST-specified byte order for AES. The 9th byte (counting from 1) is reserved for domain separation: it is set to a distinct value for each occurence of AES-CTR within PROV, specified below in each case. The counter is incremented starting from the first byte of the IV. Note that this is not the default behavior of AES-CTR in its NIST specfication, which corresponds to the reverse byte order ; but it is conform to the recommendations of the same document, which give some leeway for the implementation of the counter. Since the matrices we generate are much smaller than 2^64, the counter never spills over the domain separation byte, ensuring that each AES input for a given seed is unique. For information about the order in which bits are inserted into each matrix, please refer to Section 2.6.3.

1. For *public* seed expansion ($\mathsf{Expand_{pk}}$), we use AES reduced to four rounds. As in full-round AES, the last round skips the `MixColumns` step. The matrices $\mathbf{P}_i^1$ (resp. $\mathbf{P}_i^2$) are generated using successive outputs of AES-CTR reduced to four rounds, with the domain separation byte set to 1 (resp 2). If the size of $\mathbf{P}_i^1$ is not a multiple of 128 bits, the last AES output is truncated, and leftover bits are carried over as the initial bits for the next matrix $\mathbf{P}_{i+1}^1$. Leftover bits from the last matrix $\mathbf{P}_m^1$ (resp. $\mathbf{P}_m^2$) are discarded.

2. For *secret* seed expansion ($\mathsf{Expand_O}$), we use full-round AES. The matrices $\mathbf{O}$ is generated using successive outputs of AES-CTR, with the domain separation byte set to 3. If the size of $\mathbf{O}$ is not a multiple of 128 bits, the last AES output is truncated, and leftover bits are discarded.

2.6.2 Hashing

All hash computations use SHAKE256, from now on written $\mathcal{H}$, with a unique one-byte prefix for domain separation.

1. The public seed is derived from the secret seed with the prefix 0: $\mathsf{s_{pk}} = \mathcal{H}(0 \| \mathsf{s_{sk}})$.

2. The vinegar value $\mathbf{v}$ is as $\mathbf{v} = \mathcal{H}(4 \| \mathsf{s_{sk}} \| \mathsf{msg})$. The same SHAKE256 instance is then squeezed to produce the initial oil value (used in LinSolve, as explained in Section 2.6.4), and finally to produce successive salt values.

3. The hashed message $\mathbf{h}$ is computed as $\mathbf{h} = \mathcal{H}(5 \| \mathsf{hpk} \| \mathsf{msg} \| \mathsf{salt})$.

4. The hashed public key is computed as $\mathsf{hpk} = \mathcal{H}(6 \| \mathsf{s_{pk}} \| (\mathbf{P}_i^3))$.

2.6.3 Byte order in vectors and matrices

Bytes are read and written into vectors in the same order as they are produced by AES and SHAKE256: the first byte output becomes the first coordinate of the vector. For matrices, bytes are inserted in row-major order: that is, starting from coordinates $(0, 0)$, and proceeding along rows. The choice of inserting along rows is explained in Section 4.

2.6.4 Determinization of LinSolve

By design of PROV, the linear system computed during the signing process is expected to have many solutions. The solution output by the signing algorithm is chosen uniformly among the solution set. We now explain how this process is made deterministic.

As explained in Section 2.6.2, the message is used to derive deterministically the vinegar value $\mathbf{v}$, as well as an initial value for the vector $\mathbf{o}$. LinSolve take this initial value as input, and will use it as its random coins. In more detail, LinSolve solves the system by computing its row echelon form. Then it uses back substitution to adjust the coefficients of initial oil value so that it is a solution of the system; but it does so by *only* modifying the entries at positions corresponding to the leading coefficients of the row echelon form. Note that those positions are uniquely determined, and so is the corresponding modification of the coefficients. Also observe that if the initial oil value is uniformly random, the output solution is uniformly random among the solution set by linearity. This solution was adopted because it is easy to implement, and does not depend on low-level details of the linear solver. A more in-depth discussion of implementation aspects is offered in Section 4.

2.7 Parameter sets

Variant	λ	n	m	δ	|seed|	|salt|	|hpk|	|sig|	|pk|	|sk|	|esk|
PROV-I	128	142	49	8	16	24	32	166	81045	48	237469
PROV-III	192	206	74	8	24	32	48	238	251894	72	752528
PROV-V	256	270	100	8	32	40	64	310	588696	96	1749728

Table 1: Parameter sets and corresponding key and signature sizes for the PROV signature scheme, in bytes. The field size is always 256 bits. Although the security parameters are listed as (128, 192, 256) respectively, this should be understood in the sense of the NIST call: that is, they correspond to a bit-security of 143, 207, 272 respectively.

3 Security analysis

3.1 Security proofs

Security models. In this document, we consider the standard Existential UnForgeability under Chosen-Message Attack (EUF-CMA) notion for signature schemes. In this scenario, the adversary gets a PROV public key, and has access to the corresponding signing oracle that can be queried at most 2^{64} times. Its goal is to generate a valid signature for a new message. Formally, one has the following definition.

Definition 1 (EUF-CMA security)**.** *Let $\mathcal{H}$ be a random oracle, and let $\mathcal{A}$ be an adversary. The advantage of $\mathcal{A}$ against the* EUF-CMA *security of a signature scheme $S = (S.\text{KeyGeneration}, S.\text{Sign}^{\mathcal{H}}, S.\text{Verify}^{\mathcal{H}})$*

is defined as

$$\mathsf{Adv}_S^{\mathsf{EUF\text{-}CMA}}(\mathcal{A}) = \Pr\left[S.\textsc{Verify}(\mathsf{pk}, \mathsf{msg}, \mathsf{sig}) = \top \textit{ and } S.\textsc{Sign}^{\mathcal{H}}(\mathsf{sk}, \cdot) \textit{ was never queried on } \mathsf{msg}\right],$$

where $(\mathsf{pk}, \mathsf{sk}) \leftarrow S.\textsc{KeyGeneration}()$ *and* $(\mathsf{msg}, \mathsf{sig}) \leftarrow \mathcal{A}^{\mathcal{H}, S.\textsc{Sign}^{\mathcal{H}}(\mathsf{sk}, \cdot)}(\mathsf{pk})$.

In this section, we provide proofs that PROV is EUF-CMA-secure both in the Random Oracle Model (ROM) and the Quantum ROM (QROM) as long as the UOV problem is hard to solve for our choice of parameters. In the quantum setting, we will rely on a generic result from [KX24]. To this end, it is necessary to introduce the notion of Preimage-Sampleable Function (PSF), as tailored to the Multivariate Cryptography setting in [KX24].

Definition 2 (Weak Preimage-Sampleable Function (WPSF) [KX24])**.** *A WPSF* T *consists of four algorithms:*

- $\textsc{Gen}$*: this algorithm takes as input a security parameter and outputs a function* $\mathrm{F} : \mathcal{X} \to \mathcal{Y}$ *with a trapdoor* I*;*
- F*: this algorithm takes as input a value* $x \in \mathcal{X}$ *and deterministically outputs* $\mathrm{F}(x)$*;*
- $\mathrm{I} = (\mathrm{I}^1, \mathrm{I}^2)$*: the first algorithm takes no input and samples a value* $z \in \mathcal{Z}$*; the second one algorithm takes as input* $z \in \mathcal{Z}$*,* $y \in \mathcal{Y}$*, and outputs* $x \in \mathcal{X}$ *such that* $\mathrm{F}(x) = y$*, or outputs* $\perp$ *in case of failure;*
- $\textsc{SampDom}$*: this algorithm takes as input a function* $\mathrm{F} : \mathcal{X} \to \mathcal{Y}$ *and outputs* $x \in \mathcal{X}$*.*

The security of a WPSF is defined as follows.

Definition 3 (PS security [KX24])**.** *Let* T *be a WPSF. The advantage of an adversary* $\mathcal{A}$ *against the* PS *security of* T *is defined as follows:*

$$\mathsf{Adv}_T^{\mathsf{PS}}(\mathcal{A}) = \left|\Pr\left[\mathsf{PS}_0^{\mathcal{A}} = 1\right] - \Pr\left[\mathsf{PS}_1^{\mathcal{A}} = 1\right]\right|,$$

where PS_0 *and* PS_1 *are the games defined in Algorithm 4.*

Finally, we define the INV game against a WPSF T as follows.

Definition 4 (INV security)**.** *Let* $\mathcal{A}$ *be an* INV *adversary against* T*, trying to inverse the public function* F*. We define its advantage as*

$$\mathsf{Adv}_{\mathsf{T}}^{\mathsf{INV}}(\mathcal{A}) = \Pr\left[\mathrm{F}(x) = y\right],$$

with $(\mathrm{F}, \cdot) \leftarrow \textsc{Gen}(1^\lambda)$ *and* $y \leftarrow_\$ \mathcal{Y}$*,* $x \leftarrow \mathcal{A}(\mathrm{F}, y)$*.*

Hardness assumptions. The UOV problem has been well-studied since its introduction in 1999. Over the years, multiple slight variants have been introduced. The mathematical problem that underlies PROV is the well-known UOV^- problem, and can be defined as follows.

Definition 5 (UOV^- problem)**.** *Let* $\mathcal{P}$ *the quadratic map associated with a* PROV *public key* pk*. The* UOV^- *problem asks to find* $\mathbf{s} \in \mathbb{F}^n$ *such that* $\mathcal{P}(\mathbf{s}) = \mathbf{t}$*. More formally, the advantage of an adversary* $\mathcal{A}$ *against the* INV *security of the* UOV^- *is defined as*

$$\mathsf{Adv}_{\mathsf{UOV}^-}^{\mathsf{INV}}(\mathcal{A}) = \Pr\left[\mathcal{P}(\mathbf{s}) = \mathbf{y}\right],$$

where $(\mathsf{pk}, \mathsf{sk}) \leftarrow \textsc{KeyGeneration}()$*,* $\mathbf{y} \leftarrow_\$ \mathbb{F}^m$*, and* $\mathcal{P}$ *is the quadratic map corresponding to* pk*.*

Algorithm 4 Preimage sampling game.

procedure PS_b
  $(\mathrm{F}, \mathrm{I}) \leftarrow \textsc{Gen}(1^\lambda)$
  $b^* \leftarrow \mathcal{A}^{\mathsf{Sample}_b}(\mathrm{F})$
  return b^*

procedure Sample_1
  $x_i \leftarrow \textsc{SampDom}(\mathrm{F})$
  return x_i

procedure Sample_0
  $z_i \leftarrow \mathrm{I}^1()$
  repeat
    $y_i \leftarrow_\$ \mathcal{Y}$
    $x_i \leftarrow \mathrm{I}^2(z_i, y_i)$
  until $x_i \neq \perp$
  return x_i

Note that the UOV^- problem can also be recast as a an inversion (INV) problem for the following WPSF, dubbed $\mathsf{T}_{\mathsf{PROV}}$:

- the $\textsc{Gen}$ algorithm corresponds to the key generation function;
- F is the evaluation of the public quadratic map;
- $\textsc{SampDom}$ samples a value in $\mathbb{F}^{\mathsf{n}}$ uniformly at random;
- I corresponds to the pair $(\mathrm{I}^1, \mathrm{I}^2)$ described in Algorithm 5.

One clearly has $\mathsf{Adv}^{\mathsf{INV}}_{\mathsf{UOV}^-} = \mathsf{Adv}^{\mathsf{INV}}_{\mathsf{T}_{\mathsf{PROV}}}$.

Algorithm 5 Algorithms I^1 and I^2 of the $\mathsf{T}_{\mathsf{PROV}}$ WPSF.

procedure I^1
  $\mathbf{v} \leftarrow_\$ \mathbb{F}^{\mathsf{n}-\mathsf{m}-\delta}$
  return $\mathbf{v}$

procedure $\mathrm{I}^2(\mathsf{sk}, \mathbf{v}, \mathbf{y})$
  $(\mathsf{s}_{\mathsf{pk}}, \mathsf{s}_{\mathsf{sk}}) \leftarrow \mathsf{sk}$
  $\mathbf{O} \leftarrow \mathsf{Expand}_{\mathbf{O}}(\mathsf{s}_{\mathsf{sk}})$
  $(\mathbf{P}^1_i, \mathbf{P}^2_i)_{i=1,\ldots,\mathsf{m}} \leftarrow \mathsf{Expand}_{\mathsf{pk}}(\mathsf{s}_{\mathsf{pk}})$
  for $i = 1$ to m **do**
    $t_i \leftarrow \mathbf{y}_i - \mathbf{v}^\mathsf{T}\mathbf{P}^1_i\mathbf{v}$
    $\mathbf{a}_i \leftarrow \mathbf{v}^\mathsf{T}\left(\left(\mathbf{P}^1_i + \mathbf{P}^{1\,\mathsf{T}}_i\right)\mathbf{O} + \mathbf{P}^2_i\right)$
  $\mathbf{A}_{\mathbf{v}} \leftarrow (\mathbf{a}_i)_{i=1,\ldots,\mathsf{m}}$
  $\mathbf{t} \leftarrow (t_i)_{i=1,\ldots,\mathsf{m}}$
  $S \leftarrow \mathsf{LinSolve}(\mathbf{A}, \mathbf{t})$
  if $S = \emptyset$ **then**
    return $\perp$
  $\mathbf{o} \leftarrow_\$ S$
  $\mathbf{s} \leftarrow \begin{pmatrix} \mathbf{v} \\ 0 \end{pmatrix} + \begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix}\mathbf{o}$
  return $\mathbf{s}$

Classical security. The classical security of Hash-and-Sign with Retry was analyzed by a subset of the PROV authors in [CFGM24]. Because the analysis is involved, we refer the reader to [CFGM24] for a self-contained treatment, and cite here only the main result, and its application to PROV. In order to state the result, it is useful to have condition expressing the fact that the preimage-sampleable TDF (specifically, I_2) does not return $\perp$ too often. This is the purpose of the following definition.

Definition 6 ([CFGM24]). *We say that a preimage-sampleable TDF is (f, ε)-well-behaved for the signature scheme if it satisfies the following properties:*

- *In the main loop of the real signature algorithm I_2 returns $\vec{s} \neq \perp$ with probability $p = 1$, or some probability $p \leq 1/2$, over the randomness of the salt.*
- *Except with probability ε, over the choice of $\mathbf{v}$, i.e. the randomness of r_1, $p \geq f$.*

Theorem 1 ([CFGM24], Theorem 1). *Let T be a correct, (f, ε)-well-behaved WPSF, and let $\mathsf{HaS}_{\mathsf{T}}$ be the instantiation of the Hash-and-Sign with Retry construction using T as the trapdoor. Let $\mathcal{A}$ be an adversary against the $\mathsf{EUF\text{-}CMA}$-security of $\mathsf{HaS}_{\mathsf{T}}$ that issues at most $q_{\mathcal{H}}$ random oracle queries, q_{sign} signature queries, and runs in time at most t. Then, there exists an adversary $\mathcal{B}$ against the PS-security of T and an adversary $\mathcal{C}$ against its INV-security such that*

$$\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HaS}_{\mathsf{T}}}(\mathcal{A}) \leq \mathsf{Adv}^{\mathsf{PS}}_{\mathsf{T}}(\mathcal{B}) + q\varepsilon + q_h \mathsf{Adv}^{\mathsf{INV}}_{\mathsf{T}}(\mathcal{C}) + \frac{1}{|\mathcal{Y}|} + \mathcal{O}\left(\frac{\log(f^{-1})}{\sqrt{f}} \cdot \frac{\sqrt{q}}{N}\right) + q e^{-\Omega(fN)}.$$

where $q = q_{\mathcal{H}} + q_{\mathsf{sign}}$. Besides, $\mathcal{B}$ (resp. $\mathcal{C}$) runs in time $t' = t + \mathcal{O}(q_{\mathsf{sign}} + q_h)$ (resp. $t'' = t + (q_{\mathcal{H}} + q_{\mathsf{sign}} + 1)(t_{\mathsf{T}} + O(1))$ where t_{T} is an upper-bound on the running time to evaluate the $\mathsf{T}.\mathrm{F}$ function). Moreover, $\mathcal{B}$ is allowed at most $q_h + q_{\mathsf{sign}}$ queries.

In order to apply Theorem 1 to PROV, the critical step is to find two real numbers f and ε such that $\mathsf{T}_{\mathsf{PROV}}$ is (f, ε)-well-behaved. Note that, during the computation of I_2, the value of p is exactly $\frac{1}{|\mathbb{F}|^{m-r}}$ where r is the rank of the linear map $\mathcal{Q}(\mathbf{v}, \cdot)$. Let us denote $\mathbf{M}_{\mathbf{v}} \in \mathbb{F}^{m \times (m+\delta)}$ the matrix of the linear map $\mathcal{Q}(\mathbf{v}, \cdot)$ for any $\mathbf{v} \in \mathbb{F}^{n-m-\delta}$, and q_i the i-th component of $\mathcal{Q}$ for $i = 1, \ldots, m$. By definition, it is clear that the j-th coefficient of the i-th row of $\mathbf{M}_{\mathbf{v}}$ is exactly the coefficient of $x_{n-m-\delta+j}$ of $q_i(\mathbf{v}, \cdot)$. More formally, this coefficient is equal to

$$\sum_{k=1}^{n-m-\delta} \alpha^i_{k,j} v_k, \text{ where } q_i(\mathbf{x}) = \sum_{j=1}^{n-m-\delta} \sum_{k=j}^{n} \alpha^i_{j,k} x_j x_k.$$

It is clear that, if $\mathbf{v}$ is null, then the rank of $\mathbf{M}_{\mathbf{v}}$ is 0. Otherwise, $\mathbf{M}_{\mathbf{v}}$ is a uniformly random matrix since the $\alpha^i_{j,k}$ are uniformly random and independent. In order to accurately study the rank of $\mathbf{M}_{\mathbf{v}}$, we rely on the following result.

Lemma 1 ([Lan95, Lev05]). *Let n, N and r be three integers such that $1 \leq r \leq N \leq n$. One has*

$$\begin{aligned} p(|\mathbb{F}|, N, n, r) &:= \Pr\left[\mathsf{rank}(A) = r\right] \\ &= |\mathbb{F}|^{(r-N)(n-r)} \frac{\prod_{j=N-r+1}^{N} \left(1 - |\mathbb{F}|^{-j}\right) \prod_{j=n-r+1}^{n} \left(1 - |\mathbb{F}|^{-j}\right)}{\prod_{j=1}^{r} \left(1 - |\mathbb{F}|^{-j}\right)}, \end{aligned}$$

where the probability is taken over the uniformly random choice of A in $\mathbb{F}^{N \times n}$.

Variant	n	m	δ	|salt|	|sig|	|pk|	|sk|	f	$\log(f^{-1})/\sqrt{f}$	ε	τ
PROV-I	142	49	8	24	166	81045	48	2^{-8}	2^{7}	2^{-159}	2^{-71}
PROV-III	206	74	8	32	238	251894	72	2^{-16}	2^{12}	2^{-263}	2^{-71}
PROV-V	270	100	8	40	310	588696	96	2^{-16}	2^{12}	2^{-263}	2^{-71}

Table 2: Parameter sets and corresponding key and signature sizes for the PROV signature scheme, in bytes. τ is an upper-bound on the probability that $\mathbf{M_v}$ is not full-rank.

Variant	I	III	V
Alternative |salt|	9	14	18
New |sig|	145	214	282

Table 3: Alternative salt sizes (in Bytes) for PROV based on the application of Theorem 1. Updated corresponding signature sizes (in Bytes) are provided for reference.

Combining our previous remark and Lemma 1, we get

$$\Pr\left[\mathsf{rank}(\mathbf{M_v}) = r\right] = \begin{cases} \frac{1}{|\mathbb{F}|^{n-m-\delta}} + \left(1 - \frac{1}{|\mathbb{F}|^{n-m-\delta}}\right)\frac{1}{|\mathbb{F}|^{m(m+\delta)}} & \text{if } r = 0, \\ \left(1 - \frac{1}{|\mathbb{F}|^{n-m-\delta}}\right) p(|\mathbb{F}|, m, m+\delta, r) & \text{if } r = 1, \ldots, m. \end{cases}$$

For any λ, let $r_0(\lambda)$ be the biggest r such that $2^\lambda \Pr\left[\mathsf{rank}(\mathbf{M_v}) < r\right] \leq 1$. In that case, $\mathsf{T}_{\mathsf{UOV}_\delta^-}$ is $\left(\frac{1}{|\mathbb{F}|^{m-r_0(\lambda)}}, \Pr\left[\mathsf{rank}(\mathbf{M_v}) < r_0(\lambda)\right]\right)$-well-behaved. Applying Theorem 1, we get the following result.

Corollary 1. *Let $\mathcal{A}$ be an adversary against the* EUF-CMA*-security of* PROV *that issues at most $q_{\mathcal{H}}$ random oracle queries, q_{sign} signature queries, and runs in time at most t. Then, there exists an adversary $\mathcal{B}$ against the* INV*-security of $\mathsf{T}_{\mathsf{PROV}}$ such that*

$$\begin{aligned} \mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{PROV}}(\mathcal{A}) \leq \mathcal{O}\left((m - r_0(\lambda))\log(|\mathbb{F}|)|\mathbb{F}|^{(m-r_0(\lambda))/2}\frac{\sqrt{q}}{N}\right) \\ + q\Pr\left[\mathsf{rank}(\mathbf{M_v}) < r_0(\lambda)\right] + q\mathsf{Adv}^{\mathsf{INV}}_{\mathsf{T}_{\mathsf{UOV}}}(\mathcal{B}) + qe^{-\Omega\left(\frac{N}{|\mathbb{F}|^{m-r_0(\lambda)}}\right)}, \end{aligned}$$

where $q = q_{\mathcal{H}} + q_{\mathsf{sign}}$, and $N = 2^{\mathsf{len}_{\mathsf{salt}}}$. Besides, $\mathcal{B}$ runs in time $t' = t + (q_{\mathcal{H}} + q_{\mathsf{sign}} + 1)(t_{\mathsf{UOV}} + O(1))$ where t_{UOV} is an upper-bound on the running time to evaluate the $\mathsf{T}_{\mathsf{UOV}}$.F.

In Table 2, we evaluate the security of PROV using the bound from Corollary 1. We remark that our bounds could be used to reduce the $\mathsf{len}_{\mathsf{salt}}$ parameter (although we choose not to do so). This is presented in Table 3.

Remark 1. *When $r = \mathsf{rank}(\mathbf{M_v}) > 0$, the expected number of salt sampling is $|\mathbb{F}|^{m-r}$. This means that the signature algorithm may leak the value of r with a simple timing attack, depending on the probability τ that $\mathbf{M_v}$ is not full-rank. Table 2 illustrates the fact that an adequate choice for δ can protect against such attacks, as it is unlikely that* PROV *will ever need to sample an additional salt as long as $q_{\mathsf{sign}} \ll 2^{71}$.*

Post-quantum security. In this paragraph, we consider that the adversary has quantum access to the underlying hash function, and classical access to its signing oracle. In this context, we will rely on a generic result from [KX24]. To this end, we first recast PROV as an instantiation of the Hash-and-Sign paradigm tailored to the UOV algorithm, as presented in Algorithm 1, composed with a variant of the BUFF generic transformation where messages are always prefixed by a hash of the public key.

Algorithm 6 The probabilistic HaS paradigm with retry.

procedure $\mathsf{HaS}[\mathsf{T},\mathcal{H}].\textsc{KeyGeneration}(1^\lambda)$
  $(\mathrm{F},\mathrm{I}) \leftarrow \textsc{Gen}(1^\lambda)$
  return (F,I)

procedure $\mathsf{HaS}[\mathsf{T},\mathcal{H}].\textsc{Verify}(\mathrm{F},\mathsf{msg},(\mathsf{salt},\mathbf{s}))$
  return $\mathrm{F}(\mathbf{s}) \stackrel{?}{=} \mathcal{H}(\mathsf{msg}||\mathsf{salt})$

procedure $\mathsf{HaS}[\mathsf{T},\mathcal{H}].\textsc{Sign}(\mathrm{I},\mathsf{msg})$
  $\mathbf{v} \leftarrow \mathrm{I}^1()$
  repeat
    $\mathsf{salt} \leftarrow_\$ \{0,1\}^{\mathsf{len}_{\mathsf{salt}}}$
    $\mathbf{s} \leftarrow \mathrm{I}^2(\mathcal{H}(\mathsf{msg}||\mathsf{salt}))$
  until $\mathbf{s} \neq \perp$
  return $(\mathsf{salt},\mathbf{s})$

One has the following generic theorem.

Theorem 2 ([KX24], Proposition 4.1)**.** *For any quantum* EUF-CMA *adversary* $\mathcal{A}$ *of* $\mathsf{HaS}[\mathsf{T},\mathcal{H}]$ *issuing at most* q_s *classical queries to the signing oracle and* q_h *quantum random oracle queries to* $\mathcal{H}$*, there exist an* INV *adversary* $\mathcal{B}$ *and a* PS *adversary* $\mathcal{C}$ *against* T *issuing* q_s *sampling queries such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HaS}[\mathsf{T},\mathcal{H}]}(\mathcal{A}) \leq & (2q_h+1)^2\mathsf{Adv}^{\mathsf{INV}}_{\mathsf{T}}(\mathcal{B}) + \mathsf{Adv}^{\mathsf{PS}}_{\mathsf{T}}(\mathcal{C}) \\ & + \frac{3}{2}q'_s\sqrt{\frac{q'_s+q_h+1}{|\mathcal{R}|}} + 2(q_s+q_h+2)\sqrt{\frac{q'_s-q_s}{|\mathcal{R}|}},\end{aligned}$$

where q'_s *is a bound on the total number of queries to* $\mathcal{H}$ *in all the signing queries, and the running time of* $\mathcal{B}$ *and* $\mathcal{C}$ *are about that of* $\mathcal{A}$*.*

As per [CDF+21], the BUFF transformation has no impact on the EUF-CMA security of the transformed scheme (both in the classical and quantum setting). Hence, we can simply apply Theorem 2 to PROV, as discussed in the proof of [KX24, Proposition 5.3]. Moreover, $q'_s > q_s$ with probability at most

$$\frac{q_s}{|\mathbb{F}|^{\mathsf{n}-\mathsf{m}-\delta}} + \frac{q_s}{|\mathbb{F}|^\delta(|\mathbb{F}|-1)}.$$

Thus, one gets the following corollary.

Corollary 2. *For any quantum* EUF-CMA *adversary* $\mathcal{A}$ *of* PROV *issuing at most* q_s *classical queries to the signing oracle and* q_h *quantum random oracle queries to* $\mathcal{H}$*, there exist an* INV *adversary* $\mathcal{B}$ *such that*

$$\begin{aligned}\mathsf{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{HaS}[\mathsf{T},\mathcal{H}]}(\mathcal{A}) \leq & (2q_h+1)^2\mathsf{Adv}^{\mathsf{INV}}_{\mathsf{T}_{\mathsf{PROV}}}(\mathcal{B}) \\ & + \frac{3}{2}q_s\sqrt{\frac{q_s+q_h+1}{2^{\mathsf{len}_{\mathsf{salt}}}}} + \frac{q_s}{|\mathbb{F}|^{\mathsf{n}-\mathsf{m}-\delta}} + \frac{q_s}{|\mathbb{F}|^\delta(|\mathbb{F}|-1)},\end{aligned}$$

where the running time of $\mathcal{B}$ *is about that of* $\mathcal{A}$*.*

3.2 Resistance to known cryptanalysis

Direct attacks. In a forgery attack, the adversary tries to directly invert the public quadratic map. We estimate the complexity of this approach as solving a uniformly random quadratic map from $\mathbb{F}^{\mathsf{n}}$ to m. In this case, the most efficient known algorithm is the so-called hybrid attack that tries to guess k variables, and then attempts to inverse the resulting quadratic map, whose complexity is given by the formula [BFP10]:

$$\min_k \left(3|\mathbb{F}|^k \binom{\mathsf{n}-k+d_{\mathrm{reg}}}{d_{\mathrm{reg}}}^2 \binom{\mathsf{n}-k}{2} \right),$$

where d_{reg} is the smallest integer d so that the coefficient of z^d in

$$\frac{(1-z^2)^{\mathsf{n}}}{(1-z)^{\mathsf{n}-k}}$$

is non-positive. This estimate is based on the XL-Wiedemann solver. In order to get an accurate evaluation of the hardness of solving this system, we rely on the automatic tool by Bellini et al. in [BMSV22] that also includes other solvers.

Quantum direct attacks. We take into account the quantum version of the direct attack from [SW16, FHK+17]. Its complexity for solving quadratic systems of e equations in e variables over $\mathbb{F}_2$ is $O(2^{0.462e})$ quantum gates.

Moreover, we also consider the hybrid attack from the previous paragraph, where the search part is accelerated using Grover's algorithm. Its complexity is

$$\min_k \left(3|\mathbb{F}|^{k/2} \binom{\mathsf{n}-k+d_{\mathrm{reg}}}{d_{\mathrm{reg}}}^2 \binom{\mathsf{n}-k}{2} \right),$$

where d_{reg} is defined as above.

Kipnis-Shamir attack. In this attack [KS98], the adversary tries to directly recover the oil space by combining on two quadratic forms. This attack targeted the original UOV scheme, where $\mathsf{n} = 2\mathsf{m}$, and $\delta = 0$. Since then, it has been extended to an attack with a complexity in $\widetilde{O}(|\mathbb{F}|^{v-o})$, where v and o respectively denote the number of oil and vinegar variables [KPG99]. In the case of PROV and ignoring polynomial factors, this attack has a complexity of $|\mathbb{F}|^{\mathsf{n}-2\mathsf{m}-2\delta}$.

Intersection attack. In this attack [Beu21], the adversary tries to recover k vectors in vector spaces of the form $\mathbf{M}_i \begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix} \cap \mathbf{M}_j \begin{pmatrix} \mathbf{O} \\ \mathbf{I}_{\mathsf{m}+\delta} \end{pmatrix}$, where $\mathbf{M}_i$ and $\mathbf{M}_j$ are the matrices of the polar forms associated to the i-th and j-th components of the public quadratic map, and both matrices are invertible. This attack essentially corresponds to solving a random system of

$$\binom{k+1}{2} o - 2\binom{k}{2}$$

equations in $k(v+o)-(2k+1)o$ variables. In the case of PROV, this corresponds to a system of

$$\binom{k+1}{2}(\mathsf{m}+\delta)-2\binom{k}{2}$$

equations in m variables. We follow the strategy of [BCH+23] to find an optimal parameter k.

Security of the symmetric primitives. Our construction relies of symmetric primitives such as keyed and unkeyed hash functions. No attack more efficient than brute force is known against the chosen algorithms.

Side-channel attacks. While PROV was not specifically designed for resistance against side-channel attacks, we expect it to provide a good level of security in such scenarios. Indeed, contrary to the SaltedUOV construction, our choice of parameters implies that it is unlikely for the signature algorithm to need more than a single salt sampling (it happens with probability $\approx |\mathbb{F}|^{-\delta-1}$). This, along with the fact that signature generation and verification involve the evaluation of symmetric primitives and linear algebra over $\mathbb{F}$, will allow efficient masked implementations of PROV. Moreover, as we reuse the field of AES, we can benefit from various masked implementation for the multiplication.

Other attacks. In [CDF+21], Cremers et al. introduce the BUFF generic construction that, given an EUF-CMA-secure signature schemes, builds a new signature algorithm that also satisfies the following security notions:

- exclusive ownership [PS05]: a signature should verify only under a single public key;
- message-bound signature: a signature should only be valid for a single message;
- non re-signability [JCCS19]: one should not be able to produce a signature under another key given a signature for an unknown message.

This transformation simply hashes the public key along with the message to be signed in the signature generation and verification algorithm. Given the size of our public keys, this would entail a significant performance overhead. Hence, we chose to instead rely on a hash of the public key. Under the assumption that the underlying hash function is collision-resistant, this is sufficient to guarantee the same additional security guarantees as the BUFF transformation.

3.3 Expected security of parameter sets

As seen in Section 3, PROV can be proven secure under the assumption that the UOV$^-$ problem is hard to solve. This bounds the possible information leakage about the secret key by the signing oracle. Our parameter selection was in part guided by the bound from Corollary 1. More concretely, we have the following criterion:

$$\frac{q_s}{|\mathbb{F}|^{\mathsf{n}-\mathsf{m}-\delta}}+\frac{q_s}{|\mathbb{F}|^{\delta}(|\mathbb{F}|-1)}+\frac{q_s(q_h+q_s)}{2^{\mathsf{len}_{\mathsf{salt}}}}\lesssim 1, \tag{4}$$

when $q_s \leq 2^{64}$ and $q_h \leq 2^{\lambda}$. However, due to the $(q_s + q_h + 1)$ factor in the reduction to the UOV$^-$ inversion problem, we would need to ensure 2λ bits of security for the underlying INV problem in order to provably guarantee λ bits of security for PROV. Since this would come with a prohibitive performance cost, we chose to ensure at least λ bits of security for the underlying INV problem. This seems reasonable as multivariate cryptography schemes have thus far never been attacked through the information leakage of their signature algorithm. As far as quantum security is concerned, Corollary 2 only provides asymptotic security for PROV. Hence, we rely on the criterion from Equation (4) and on the analysis from Section 3.2. Concrete parameters for NIST levels I, III and V can be found in Table 1.

3.4 Practical estimates

The analysis derived from the security proof in Section 3.3 mainly serves to choose the size of δ, and the salt length, in relation to the proof. It remains to assess the practical performance of the various attacks presented earlier in this section against our parameter choices. The results are depicted on Figure 4.

The first two attacks are direct attacks. The system is solved either using the XL Wiedemann algorithm, estimated using the same formula as in most UOV-based NIST submission, cf. Section 3. We have also evaluated the complexity of other direct attacks using the automatic tool by Bellini et al. [BMSV22]. The tool covers many attacks, but in all cases, the Hybrid F5 algorithm achieves the best performance against the PROV parameters. We report its performance here. We also note that it appears to slightly outperform the XL Wiedemann estimate commonly used as reference in the UOV literature: integrating the Hybrid F5 estimator leads to PROV having more conservative parameters. For both attacks, the value k in parenthesis shows the optimal number of guessed variables for the algorithm.

The last two attacks are attacks against the UOV problem. We have been especially conservative in our choice of parameters with regard to those attacks, since the hardness of the UOV problem is not as well-established as the hardness of the MQ problem. For the intersection attack, the value of k indicates the optimal number of intersected spaces. Furthermore, note that any distinguishing attack against PROV implies a distinguishing attack against an UOV system with m equations in $n + \delta$ variables, since the PROV public key gives strictly less information to the attacker (it contains δ fewer equations relative to the corresponding UOV public key), and PROV signatures provably leak no information (Section 3). We have chosen PROV parameters such that the corresponding UOV problem is difficult. This means that any attack against PROV implies an attack against UOV with parameters that are considered secure by the sate of the art.

	PROV-I	PROV-III	PROV-V
λ	128	192	256
XL-Wiedemann	151.7 $(k=2)$	216.6 $(k=5)$	281.8 $(k=6)$
Hybrid F5	**145.6** $(k=2)$	**209.9** $(k=3)$	**274.7** $(k=6)$
Kipnis-Shamir	245.3	358.4	455.2
Intersection	151.5 $(k=3)$	249.7 $(k=2)$	311.5 $(k=2)$

Table 4: Complexity of the main attacks against PROV.

4 Implementation and performance

4.1 Implementation techniques

HP-PROV [Ryc24] is the library designed for efficiently implementing PROV in C programming language, and is used as optimized implementation exploiting the AVX2 instruction set. This library is distributed under the GNU Lesser General Public License, version 3 or later, and was implemented by the last author. Here, we reveal the fundamental techniques used in HP-PROV to product an efficient implementation immune against timing attacks (except the number of iterations in the repeat loop of the signing process). Except the AVX2 instruction set, we also use the PCLMULQDQ instruction for computing the dot product over $\mathsf{GF}(2^8)$ (Section 4.1.3), and optionally the BLSI instruction (Section 4.1.7) from the BMI1 instruction set. We use the AESKEYGENASSIST, AESENC and AESENCLAST instructions from the AES-NI instruction set only to significantly speed up the seed expander based on AES in counter mode. In certain sections, we also study the possibility of using other implementations depending on the target architecture.

4.1.1 Overview

In PROV, we make the choice of storing multivariate quadratic equations one by one. Here, we study how to efficiently implement linear algebra operations with this format. The choice of the format is crucial for the performance of the implementation, and we think that efficient implementations for each format should be compared in order to determine the best format. In particular, we could consider the monomial representation [CLP+17], *i.e.* when terms of each equation are stored together. An efficient implementation for this format could be proposed in the future.

Then, for $1 \leqslant i \leqslant \mathsf{m}$, we make the choice of storing $\mathbf{P}_i^1, \mathbf{P}_i^2, \mathbf{P}_i^3, \mathbf{S}_i, \mathbf{O}$ in the row-major order. Intermediate matrices are also stored in the row-major order. The $\mathbf{P}_i^1$ and $\mathbf{P}_i^3$ matrices are upper triangular where the null lower triangular part is not stored. We note that the storage of $\mathbf{P}_i^1$ also provides $\mathbf{P}_i^{1\mathsf{T}}$, a lower triangular matrix stored in the column-major order.

Now, we show how we perform the linear algebra part of cryptographic operations. For the keypair generation, for $1 \leqslant i \leqslant \mathsf{m}$, we compute $\mathbf{Q} \leftarrow \mathbf{P}_i^1\mathbf{O} + \mathbf{P}_i^2$, then $\mathbf{P}_i^3 \leftarrow \mathsf{Sym}(\mathbf{O}^\mathsf{T}\mathbf{Q})$. If the secret-key is expanded, then we compute $\mathbf{S}_i \leftarrow \mathbf{P}_i^{1\mathsf{T}}\mathbf{O} + \mathbf{Q}$, and we replace $\mathbf{Q}$ by $\mathbf{S}_i$ in the three steps, *i.e.* we use the memory zone of $\mathbf{S}_i$ to store intermediate results. The symmetrize operation of the square matrix $\mathbf{O}^\mathsf{T}\mathbf{Q} \in \mathbb{F}^{(\mathsf{m}+\delta)\times(\mathsf{m}+\delta)}$ is detailed in Section 4.1.5. When we perform a matrix product, for each row of the left operand, we multiply each component by the corresponding row of the right operand. To do it, we use multiplication lookup tables by a scalar. We can directly load the table from the scalar if the left operand is public, which is true for $\mathbf{P}_i^1$ and its transpose. For the product of $\mathbf{O}^\mathsf{T}$ by $\mathbf{Q}$, the left operand is secret. However, $\mathbf{O}^\mathsf{T}$ is used for m matrix products. Therefore, we can precompute the multiplication table for each coefficient of $\mathbf{O}^\mathsf{T}$ (without computing the transpose of $\mathbf{O}$), then use them to speed up m matrix products. We refer to Section 4.1.4 for further details.

For the signing process, for $1 \leqslant i \leqslant \mathsf{m}$, we compute t_i and $\mathbf{a}_i$ with four vector-matrix products and one dot product, as follows.

1. $\mathbf{w} = \mathbf{v}^\mathsf{T}\mathbf{P}_i^1$,
2. $t_i \leftarrow h_i - \mathbf{w}\mathbf{v}$,
3. $\mathbf{a}_i \leftarrow \mathbf{v}^\mathsf{T}\mathbf{P}_i^2$,

4. $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}^\mathsf{T}\mathbf{P}_i^{1\,\mathsf{T}}$,

5. $\mathbf{a}_i \leftarrow \mathbf{a}_i + \mathbf{w}\mathbf{O}$.

If the secret-key is expanded, we merge Steps 3, 4 and 5 by computing $\mathbf{a}_i \leftarrow \mathbf{v}^\mathsf{T}\mathbf{S}_i$, which saves two vector-matrix products. The secret vector $\mathbf{v}$ is used for 3m vector-matrix products. Therefore, for each component of $\mathbf{v}$, similarly to the matrix products, we can precompute the multiplication lookup table by this element, in order to speed up the multiplication of each component of $\mathbf{v}$ by the corresponding row of the matrix. However, $\mathbf{P}_i^{1\,\mathsf{T}}$ is stored in the column-major order, so we cannot use this strategy without computing the transpose. For the constant-time computation of $\mathbf{o} \leftarrow_\$ \mathsf{LinSolve}(\mathbf{A}_\mathbf{v}, \mathbf{t})$, we refer to Sections 4.1.6 and 4.1.7. The linear solver requires inverting elements of $\mathsf{GF}(256)^\times$, which is detailed in Section 4.1.2. It also requires computing dot products, which is detailed in Section 4.1.3. Finally, the computation of $\mathbf{v} + \mathbf{O}\mathbf{o}$ is performed as $\mathbf{v} + \mathbf{o}^\mathsf{T}\mathbf{O}^\mathsf{T}$. Once again, we cannot use multiplication lookup tables without computing the transpose of $\mathbf{O}$.

For the verifying process, for $1 \leqslant i \leqslant \mathsf{m}$, we set $\mathbf{v}, \mathbf{o} \leftarrow \mathbf{s}$, and we compute t_i with three vector-matrix products and one dot product, as follows: $t_i \leftarrow (\mathbf{v}^\mathsf{T}\mathbf{P}_i^1, \mathbf{v}^\mathsf{T}\mathbf{P}_i^2 + \mathbf{o}^\mathsf{T}\mathbf{P}_i^3)\mathbf{s}$. Here, $\mathbf{v}$ and $\mathbf{o}$ are public vectors. So, the multiplication lookup tables can be loaded from the components of $\mathbf{v}$ and $\mathbf{o}$ in order to speed up their multiplication by the corresponding row of each matrix.

In Section 4.1.8, we present some minor changes which improve the performance, such as multiplying a vector by two matrices during the signing and verifying processes.

4.1.2 Modular inverse in $\mathsf{GF}(256)^\times$

During the Gauss-Jordan elimination (Section 4.1.6), we need to multiply the pivot row by the inverse of the pivot, before using it for the elimination step. The pivot is a secret data and has to be inverted in constant-time to prevent timing attacks. Several methods can be used for computing the inverse of the pivot: the constant-time lookup in an inversion table, the exponentiation algorithm, and the constant-time extended Euclid-Stevin algorithm [BY19]. In the optimized implementation, we perform the constant-time lookup in an inversion table. The inversion table takes the element to invert as index, and returns its inverse. We set the inverse of zero to any value different from zero (*e.g.* the field polynomial modulo x^8) because of a specificity of the constant-time Gauss-Jordan elimination. The latter requires that the pivot row is multiplied by a value different from zero for keeping the same solutions.

Now, we present several inversion algorithms. The choice of the most efficient inversion method depends on the target architecture.

Constant-time lookup. With the AVX2 instruction set, the constant-time lookup in an inversion table is the most efficient method. We load the table of 256 bytes with only 8 calls to the load instruction. We apply a mask on the loaded values in order to set to zero the seven 256-bit data which do not contain the inverse, then we accumulate them (with a logical OR). Once the accumulator is computed, we use the VPSHUFB instruction to extract the byte containing the inverse.

Constant-time extended Euclid-Stevin algorithm. In the polynomial basis of $\mathsf{GF}(2^8)$, each element A is a degree-7 binary polynomial. We can compute the inverse of A modulo the field polynomial f with the extended Euclidean algorithm, which computes a Bezout relationship between A and f: $Au + fv = \gcd(A, f) = 1$, with $u, v \in \mathsf{GF}(2)[x]$. The Bezout coefficient u is the

inverse of A in $\mathsf{GF}(2^8)$. For computing the Bezout coefficient u in constant-time, we can use a variant: the extended Euclid-Stevin algorithm [BY19]. We propose Algorithm 7, which is an implementation in C programming language of [Ryc21, Algorithm 53], coupled to [Ryc21, Remark 25], and by adding the computation of the Bezout coefficient u. We refer to [Ryc21, Section B.8] for further details. We note that this implementation can easily be adapted to 16-bit processors. In fact, we use 32-bit registers only to compute two 16-bit instructions via one 32-bit instruction. The operations applied on a or b have to be applied on their Bezout coefficient.

Exponentiation algorithm. Based on Fermat's little theorem, the inverse of $A \in \mathsf{GF}(256)^\times$ is equal to A^{254}. The way of raising A to the power of 254 is defined by the so-called addition chain [IT88]. An addition chain of a positive integer n is a list of integers such that the last integer is n, and each integer can be computed by adding two integers which precede it in the list. In the exponentiation algorithm, these integers correspond to the powers of A which are computed. The addition chain is not unique and impacts the performance. For computing A^{254}, we recommend the following addition chain: $(1, 2, 3, 6, 7, 14, 15, 30, 60, 120, 240, 254)$. The corresponding computation of A^{254} is the following:

1. $A_2 = A^2$,
2. $A_3 = A_2 \times A$,
3. $A_6 = (A_3)^2$,
4. $A_7 = A_6 \times A$,
5. $A_{14} = (A_7)^2$,
6. $A_{15} = A_{14} \times A$,
7. $A_{240} = (A_{15})^{2^4}$,
8. $A_{254} = A_{240} \times A_{14}$.

With this addition chain, the inversion algorithm requires 4 field multiplications, 3 field squarings and one step of multi-squaring (computing 4 successive squares). If the target architecture has an instruction for computing the multiplication in $\mathsf{GF}(256)$, this method requires at most 11 calls to this instruction. Otherwise, this addition chain is interesting because it allows to compute $(A^{15})^{2^4}$ with a multi-squaring table, depending only on the field representation of $\mathsf{GF}(256)$. Moreover, three of the four multiplications takes A as right operand. This allows to perform a step of precomputation about A to compute more quickly these multiplications.

We also recommend a similar addition chain: $(1, 2, 3, 6, 12, 14, 15, 30, 60, 120, 240, 254)$. The corresponding computation of A^{254} is the following:

1. $A_2 = A^2$,
2. $A_3 = A_2 \times A$,
3. $A_{12} = (A_3)^{2^2}$,
4. $A_{14} = A_{12} \times A_2$,

Algorithm 7 32-bit implementation in C programming language of the inversion in $\mathsf{GF}(2^8)$ via the extended Euclid-Stevin algorithm. The field polynomial is $b = x^8 + x^4 + x^3 + x + 1$.

```
uint8_t inv_xes_cb_gf256_32(uint8_t A)
{
  uint32_t au,bq,delta,sw,mask_lc_a;
  uint8_t i;

  /* maxdeg(A)-deg(b) */
  delta=-1;
  /* (a<<1)||(u<<1), u=1, we multiply a and u by x^-delta, i.e. x */
  au=(((uint32_t)A)<<17)|2;
  /* b||q, q=0 */
  bq=((uint32_t)0x11b)<<16;

  /* maxdeg(A)+deg(b)=15 */
  for(i=0;i<15;++i)
  {
    mask_lc_a=-(au>>24);
    /* if lc(A)==1 and delta<0, we swap operands (before the elimination) */
    sw=mask_lc_a&(-(delta>>31));
    delta^=(delta^(-delta))&sw;
    --delta;

    /* elimination of the degree-8 term of a: new_a=a*b_8+b*a_8 */
    /* necessary, b_8=1, and the missing swap does not impact the result */
    au^=bq&mask_lc_a;
    /* achieve the conditional swap between a and b */
    /* note that sw!=0 implies mask_lc_a!=0 */
    bq^=au&sw;
    /* multiplication by x (alignment of leading terms) */
    au<<=1;
  }

  /* division by x^8, where 8 is max(deg(A),deg(0x11b)) */
  return (uint8_t)(bq>>8);
}
```

5. $A_{15} = A_{12} \times A_3$,
6. $A_{240} = (A_{15})^{2^4}$,
7. $A_{254} = A_{240} \times A_{14}$.

With this addition chain, we use only one field squaring but 2 steps of multi-squaring. The right operand of the multiplications changes but the dependencies between the operations is lower.

Square-and-multiply exponentiation algorithm. We can also compute the inverse of $A \in \mathsf{GF}(256)^\times$ as A^{254} with the classical square-and-multiply exponentiation algorithm. To do it, initialize B to A and perform six times $B \leftarrow B^2 \times A$, then return B^2. This method is more expensive than the previous one, because it requires 6 field multiplications and 7 field squarings. However, squaring is linear so we can include the squaring step in a lookup table of multiplication by A. Thus, we introduce the *square-and-multiply lookup table*, which computes the square followed by the multiplication by A. For example, we can generate the lookup table of each element of the polynomial basis of $\mathsf{GF}(256)$: we successively compute $A, Ax^2, Ax^4, Ax^6, Ax^8, Ax^{10}, Ax^{12}, Ax^{14}$ in $\mathsf{GF}(256)$, by computing Ax^{2i} as $(Ax^{2i-2}) \times x^2$ for $1 \leqslant i \leqslant 7$. Then, for $B \in \mathsf{GF}(256)$, $B^2 \times A$ can be computed by computing the dot product of the vector of the eight previous values by the bits of B. Note that this method can also be used to compute the square (or repeated squares), with the possibility of precomputing $1, x^2, x^4, x^6, x^8, x^{10}, x^{12}, x^{14}$ in $\mathsf{GF}(256)$ for a fixed field polynomial. Note that we could also consider the use of the PSHUFB instruction (defined in Section 4.1.4) coupled to two 16-byte square-and-multiply lookup tables.

4.1.3 Dot product over $\mathsf{GF}(2^8)$

In this section, we use the PCLMULQDQ instruction, which performs a 64-bit carry-less multiplication. It computes the product of two binary polynomials such that their degree is strictly less than 64. Inputs and output are 128-bit registers, and only half of each input is used.

In [Ryc21, Section 9.5.1], a trick based on PCLMULQDQ allows to perform the multiplication of degree-one polynomials whose coefficients are degree-20 binary polynomials. In particular, this trick allows to perform two multiplications of degree-20 binary polynomials with one call to the PCLMULQDQ instruction. However, the format of the output is efficient only for the accumulation of products, *e.g.* for the dot product of binary polynomials. In this section, we adapt this trick to perform the multiplication of degree-three polynomials whose coefficients are degree-7 binary polynomials. Then, we remark that the degree-three term of the result is a dot product of the coefficient vectors of the inputs (the order of coefficients is reversed for one of the inputs).

Here, we consider the polynomial basis of $\mathsf{GF}(2^8)$, allowing to write each element as a degree-7 binary polynomial. Let $\mathbf{u} = (u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7)$ and $\mathbf{v} = (v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7)$ be two vectors of 8 degree-7 elements of $\mathsf{GF}(2)[x]$. We perform the dot product of $\mathbf{u}$ and $\mathbf{v}$ with only two calls to the PCLMULQDQ instruction. To do it, we start by reversing the order of 16-bit blocks of one of both 64-bit vectors, for example $\mathbf{u}$. We obtain $\mathbf{u}' = (u_6, u_7, u_4, u_5, u_2, u_3, u_0, u_1)$. Then, for both vectors, we use 16-bit block shifts to perform:

- $\mathbf{u}'_e \leftarrow (0, u_6, 0, u_4, 0, u_2, 0, u_0)$,
- $\mathbf{v}_e \leftarrow (0, v_0, 0, v_2, 0, v_4, 0, v_6)$,

- $\mathbf{u}'_o \leftarrow (u_7, 0, u_5, 0, u_3, 0, u_1, 0)$,
- $\mathbf{v}_o \leftarrow (v_1, 0, v_3, 0, v_5, 0, v_7, 0)$,

and we compute $p_e \leftarrow \text{PCLMULQDQ}(\mathbf{u}'_e, \mathbf{v}_e)$ and $p_o \leftarrow \text{PCLMULQDQ}(\mathbf{u}'_o, \mathbf{v}_o)$. Here, the operands of PCLMULQDQ are considered as degree-63 binary polynomials, which is equivalent to consider the dot product of each operand with $(1, x^8, x^{16}, x^{24}, x^{32}, x^{40}, x^{48}, x^{56})$. Then, the multiplication of binary polynomials is performed. Thanks to the null coefficients in $\mathbf{u}'_e, \mathbf{v}_e$ (respectively $\mathbf{u}'_o, \mathbf{v}_o$), the coefficients of p_e from x^{64} to x^{79} (respectively of p_o from x^{48} to x^{63}) correspond to the dot product of the even (respectively odd) coefficients of $\mathbf{u}$ and $\mathbf{v}$. We extract these coefficients as degree-14 binary polynomials, and the sum of both is the dot product of $\mathbf{u}$ and $\mathbf{v}$ over $\mathsf{GF}(2)[x]$. Finally, applying the modular reduction by the field polynomial allows to convert the result in $\mathsf{GF}(2^8)$.

For the sake of simplicity, we have presented the dot product for vectors of 8 elements. In the optimized implementation, the previous process is extended to vectors of 16 elements, which requires four calls to the PCLMULQDQ instruction. We recall that this instruction allows to choose between the 64-bit lower and higher parts of each 128-bit operand.

We perform the dot product of vectors of n elements by applying the previous process on each 128-bit block. All products are accumulated in p_e and p_o, in function of the position of the null coefficients. Finally, the extraction step followed by the modular reduction by the field polynomial is performed only one time, at the end of the algorithm.

When the same vector $\mathbf{u}$ is used for several dot products, we reverse only one time the order of 16-bit blocks of this vector. We sometimes precompute $\mathbf{u}'_e$ and $\mathbf{u}'_o$ for accelerating the dot product, in particular when we multiply a vector by a matrix stored in the column-major order (*e.g.* $\mathbf{P}_i^{1\,\mathsf{T}}$ and $\mathbf{O}^\mathsf{T}$).

We note that the standard way of efficiently computing the dot product over $\mathsf{GF}(256)$ requires the use of the tower field representation of $\mathsf{GF}(256)$. With this representation, the dot product can be computed by block of 32 elements with logarithm and exponential tables. The tower field representation could be considered in the future, and we refer to [Ryc21, Section 7.4.9] for further details about parallel multiplications. However, our current representation of $\mathsf{GF}(256)$ could be more efficient for other architectures, in particular if they have an instruction computing several 8-bit carry-less multiplications in parallel. For example, the PMULL instruction of the Arm Neon architecture extension performs eight 8-bit carry-less multiplications in parallel. We also note that on advanced Intel processors, the GFNI instruction set performs field multiplications in parallel only for our representation of $\mathsf{GF}(256)$. Finally, we note that if the VPCLMULQDQ is available, then our dot product algorithm can be extended to 512-bit registers, dividing by 4 the number of calls to a carry-less multiplication instruction.

4.1.4 Multiplication of a vector by a scalar over $\mathsf{GF}(256)$

The multiplication of a vector by a scalar is the fundamental operation of the optimized implementation. We use it to perform the vector-matrix products, matrix multiplications, and elimination steps by the pivot row during the Gauss-Jordan algorithm. The crucial instruction to efficiently perform this multiplication is VPSHUFB from the AVX2 instruction set, which performs two times the PSHUFB instruction in parallel. The PSHUFB instruction from the SSSE3 instruction set takes sixteen indices on four bits, and looks up the corresponding 8-bit elements in a 16-byte lookup table. In fact, each index is on eight bits, but only the four lower bits are considered. However, if the highest bit is set, then the corresponding output will be null.

We multiply a vector by a scalar by using multiplication lookup tables coupled to the VPSHUFB instruction. The VPSHUFB instruction is used twice: one time for multiplying the 4 lower bits of each element of the vector by the scalar, and one time for multiplying the 4 higher bits. For any scalar λ in $\mathsf{GF}(256)$, this fact implies to load two tables: a table of multiplication by λ, and another table of multiplication by $\lambda \times \beta_4$, where β_4 is the element of the basis of $\mathsf{GF}(256)$ whose multiplication is equivalent to left shift by 4 positions (*e.g.* $\beta_4 = x^4$ in the polynomial basis). So, the optimized implementation provides a lookup table T of size $256 \times 32 = 8192$ bytes. For each element of $\mathsf{GF}(256)$, we store both 16-byte lookup tables. Thus, both tables can be loaded with one call to the load instruction, then we extend each table to 32-byte registers via the VPERMQ instruction.

When the scalar is public, we directly load both tables by using the scalar as index of T. Otherwise, the optimized implementation provides another lookup table T_x of size $8 \times 32 = 256$ bytes. This table is a smaller version of T which only contains the multiplication tables by the elements of the basis of $\mathsf{GF}(256)$ (*e.g.* $1, x, x^2, x^3, x^4, x^5, x^6, x^7$ in the polynomial basis). In this way, we can generate $T[\lambda]$ as the linear combination of T_x by the bits of λ. We present some constant-time techniques for multiplying the 8 bits of λ by the 32-byte elements of T_x. For all techniques, λ has to be duplicated 32 times in order to fill a 32-byte register.

Most methods are based on creating a 256-bit mask with each bit of λ, then applying this mask on T_x with a logical AND. The performance of each method is target dependent. Therefore, a tuning should be performed in order to select the best method for the current processor.

If we start by shifting the i-th bit of λ to the most significant bit for $0 \leqslant i < 7$ (the 7-th bit is already the most significant bit), there are several strategies:

- Use the VPSHUFB instruction with the negation of zero as lookup table. This method requires considering the negation of the previous result before applying the final logical AND. This final step can directly be performed with the VPANDN instruction.
- Use the arithmetic right shift (VPSRAW).
- Perform the 16-bit vector signed higher multiplication with one (VPMULHW).
- Look if zero is strictly greater than the data (VPCMPGTB).

These strategies require at most one extra mask. Otherwise, if we allow a larger number of masks, we can also start by setting to zero all bits except the i-th bit (via a logical AND). When $i = 7$, using the previous techniques is faster because the i-th bit is directly duplicated in one instruction. Here are some possible strategies for duplicating the i-th bit, mainly useful for $i < 7$:

- Perform the 16-bit vector signed higher multiplication with the negation of zero (VPMULHW).
- Look if the data is equal to zero (VPCMPEQB). This method requires considering the negation of the previous result before applying the final logical AND. This final step can directly be performed with the VPANDN instruction.
- Look if the data is equal to the mask which has extracted the bit (VPCMPEQB).
- Look if the data is strictly greater than zero (VPCMPGTB), for $i < 7$. Look if zero is strictly greater than the data, for $i = 7$.

In [BCH+23, Figure 2], the authors propose to slightly modify the last method in order to decrease the number of masks. First, shift λ to the right by one position. Second, for $0 \leqslant i < 4$, extract the $2i$-th bit of λ (respectively $\lambda \gg 1$) with a logical AND with the precomputed mask 2^{2i}, then looks if the result is strictly greater than zero. Third, perform a logical AND of the previous result with $T_x[2i]$ (respectively $T_x[2i+1]$). We note that the method of [BCH+23] can be improved. To do it, we use a left shift instead of the right shift by one position, which implies to multiply by two each precomputed mask. We also use 8-bit vector instructions instead of 16-bit vector instructions (except the left shift). Then, for $i = 3$, we look if zero is strictly greater than the data. Thanks to the left shift, the 6-th and 7-th bits of λ are at the most significant bit position, so no mask is required. Our improvement is faster because we use only 14 logical AND versus 16 logical AND for the original method. Most of Intel processors can perform three AND in parallel, so we save a triplet. We also use three masks instead of four. We also can adapt this idea for the VPCMPEQB instruction, which could be faster.

Another method proposed in [BCH+23] is to start by shifting the i-th bit of λ to the least significant bit, then computing a logical AND with one, and finally computing the 16-bit vector signed lower multiplication with $T_x[i]$ (VPMULLW), for $0 \leqslant i < 8$. Once again, we note that this method is faster if the most significant bit is directly duplicated in one instruction.

In Algorithm 8, we introduce a very efficient strategy based on VPSHUFB: we directly look up the mask in a table. Here, the T_x table corresponds to `mulxtab_x0_x4_gf256` in the optimized implementation. The use of VPCMPGTB for the 7-th bit slightly speeds up the implementation. The drawback of Algorithm 8 is the large number of registers, which is not recommended if it is inlined in another function. If the function is inlined, then we recommend the use of the VPSHUFB instruction with the negation of zero as lookup table, or sometimes the use of the arithmetic right shift (VPSRAW).

Finally, when we need to generate the multiplication table for $16j$ scalars stored in j 16-byte registers for $j \geqslant 1$, the authors of [BCH+23] propose a method more efficient. We refer to [BCH+23, Figure 3] for further details. In Algorithm 9, we propose a variant of this method which decreases the number of required registers. We successively use VPUNPCKLBW, VPUNPCKLWD and VPUNPCKLDQ in the core loop of [BCH+23, Figure 3], in order to successively generate three masks from another mask which is incremented at each iteration. The use of these instructions is similar to the matrix transpose of the next section.

Algorithm 9 loads data by block of 32 bytes. Therefore, the memory zone allocated for u has to be a multiple of 32 bytes, in particular when j is odd.

4.1.5 Computing the transpose of a square matrix over $\mathsf{GF}(256)$

The keypair generation requires symmetrizing m square matrices in order to obtain an upper triangular matrix. Let $\mathbf{M} \in \mathbb{F}^{(\mathsf{m}+\delta)\times(\mathsf{m}+\delta)}$. To symmetrize $\mathbf{M}$, we have to add the transpose of the strictly lower triangular part of $\mathbf{M}$ to its upper triangular part. To do it, we decompose $\mathbf{M}$ into square blocks of size 16×16. Then, for each block of the lower triangular part of $\mathbf{M}$, we compute its transpose then we add it to the corresponding block in the upper triangular part. Here, we explain how to efficiently perform this operation, via the SSE2 instruction set. However, we note that the diagonal blocks require a different implementation since they contains both the lower and upper triangular parts. Moreover, if $\mathsf{m} + \delta$ is not a multiple of 16, then we have to use a specific implementation to transpose the last rows of $\mathbf{M}$.

Algorithm 8 AVX2 implementation in C programming language of T[u[i]] for $0 \leqslant i < n$.

```
 1: void vec_to_mulTab_v1_gf256_avx2(uint8_t *T, const uint8_t *u, unsigned int n)
 2: {
 3:   const __m256i x0=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256);
 4:   const __m256i x1=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+1);
 5:   const __m256i x2=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+2);
 6:   const __m256i x3=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+3);
 7:   const __m256i x4=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+4);
 8:   const __m256i x5=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+5);
 9:   const __m256i x6=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+6);
10:   const __m256i x7=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+7);
11:   const __m256i m_0f=_mm256_set1_epi8(15);
12:   const __m256i m_ff00=_mm256_slli_epi16(~_mm256_setzero_si256(),8);
13:   const __m256i m_ffff0000=_mm256_slli_epi32(~_mm256_setzero_si256(),16);
14:   const __m256i m_ffffffff00000000=_mm256_slli_epi64(~_mm256_setzero_si256(),32);
15:   const __m256i mh=_mm256_slli_si256(~_mm256_setzero_si256(),8);
16:   __m256i u0,u1,r;
17:   unsigned int i;

18:   for(i=0;i<n;++i)
19:   {
20:     u0=_mm256_set1_epi8(u[i]);
21:     r=_mm256_cmpgt_epi8(_mm256_setzero_si256(),u0)&x7;
22:     u1=_mm256_srli_epi16(u0,4)&m_0f;
23:     u0&=m_0f;
24:     r^=_mm256_shuffle_epi8(m_ff00,u0)&x0;
25:     r^=_mm256_shuffle_epi8(m_ff00,u1)&x4;
26:     r^=_mm256_shuffle_epi8(m_ffff0000,u0)&x1;
27:     r^=_mm256_shuffle_epi8(m_ffff0000,u1)&x5;
28:     r^=_mm256_shuffle_epi8(m_ffffffff00000000,u0)&x2;
29:     r^=_mm256_shuffle_epi8(m_ffffffff00000000,u1)&x6;
30:     r^=_mm256_shuffle_epi8(mh,u0)&x3;
31:     _mm256_storeu_si256((__m256i*)T+i,r);
32:   }
33: }
```

Algorithm 9 AVX2 implementation in C programming language of T[u[i]] for $0 \leqslant i < n = 16j$.

```
1: void vec_to_mulTab_gf256_avx2(uint8_t *T, const uint8_t *u, unsigned int n) {
2:   const __m256i x0=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256);
3:   const __m256i x1=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+1);
4:   const __m256i x2=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+2);
5:   const __m256i x3=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+3);
6:   const __m256i x4=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+4);
7:   const __m256i x5=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+5);
8:   const __m256i x6=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+6);
9:   const __m256i x7=_mm256_loadu_si256((__m256i*)mulxtab_x0_x4_gf256+7);
10:  const __m256i mask_0f=_mm256_set1_epi8(15);
11:  const __m256i c256=_mm256_set1_epi16(256);
12:  __m256i u32,u0,u1,ul,u_x0_x4,u_x0,u_x1,u_x2,u_x3,mask,mask2;
13:  unsigned int i,k,l;
14:  for(i=0;i<(n>>4);i+=2) {
15:    u32=_mm256_loadu_si256((__m256i*)u); /* 16-byte padding of u may be required */
16:    u_x0_x4=_mm256_permute4x64_epi64(u32,0x44);
17:    u32    =_mm256_permute4x64_epi64(u32,0xee);
18:    for(k=0;k<2;++k) {
19:      u0=u_x0_x4&mask_0f;
20:      u1=_mm256_srli_epi16(u_x0_x4,4)&mask_0f;
21:      u_x0=_mm256_shuffle_epi8(x0,u0)^_mm256_shuffle_epi8(x4,u1);
22:      u_x1=_mm256_shuffle_epi8(x1,u0)^_mm256_shuffle_epi8(x5,u1);
23:      u_x2=_mm256_shuffle_epi8(x2,u0)^_mm256_shuffle_epi8(x6,u1);
24:      u_x3=_mm256_shuffle_epi8(x3,u0)^_mm256_shuffle_epi8(x7,u1);
25:      /* l 0xf0 l 0xf0 l 0xf0 l 0xf0 l 0xf0 l 0xf0 l 0xf0 l 0xf0 */
26:      mask=_mm256_srli_epi16(mask_0f,4);
27:      for(l=0;l<16;++l) {
28:        ul =_mm256_shuffle_epi8(u_x0,mask);
29:        /* l l 0xf0 0xf0 l l 0xf0 0xf0 l l 0xf0 0xf0 l l 0xf0 0xf0 */
30:        mask2=_mm256_unpacklo_epi8(mask,mask);
31:        ul^=_mm256_shuffle_epi8(u_x1,mask2);
32:        /* l l l l 0xf0 0xf0 0xf0 0xf0 l l l l 0xf0 0xf0 0xf0 0xf0 */
33:        mask2=_mm256_unpacklo_epi16(mask2,mask2);
34:        ul^=_mm256_shuffle_epi8(u_x2,mask2);
35:        /* l l l l l l l l 0xf0 0xf0 0xf0 0xf0 0xf0 0xf0 0xf0 0xf0 */
36:        mask2=_mm256_unpacklo_epi32(mask2,mask2);
37:        ul^=_mm256_shuffle_epi8(u_x3,mask2);
38:        _mm256_storeu_si256((__m256i*)T,ul);
39:        /* l+1 f0 l+1 f0 l+1 f0 l+1 f0 l+1 f0 l+1 f0 l+1 f0 l+1 f0 */
40:        mask=_mm256_add_epi16(mask,c256);
41:        T+=32;
42:      } u+=16;
43:      if((i+1+k)==(n>>4)) break;
44:      u_x0_x4=u32;
45: }}}
```

First, we transpose a block of size 8×16 as follows. We load the eight rows of this block, which generates eight 128-bit registers. Then we apply PUNPCKLBW and PUNPCKHBW on the four couples of successive rows, in order to obtain the transpose of four blocks of size 2×16, stored on two 128-bit registers. Now, we repeat this process by applying PUNPCKLWD and PUNPCKHWD on both couples of transpose of blocks of size 2×16. We obtain two transposes of blocks of size 4×16 stored on four 128-bit registers. We repeat this process by applying PUNPCKLDQ and PUNPCKHDQ on the couple of transpose of blocks of size 4×16. Then, we obtain the transpose of a block of size 8×16 stored on eight 128-bit registers.

Second, we transpose a square block of size 16×16 as follows. We load the first eight rows of this block, and via the previous process, we obtain its transpose. We could repeat this process for the last eight rows, then apply PUNPCKLQDQ and PUNPCKHQDQ, but we think that the numbers of required registers would be too large. So, we split each of the eight registers in two 64-bit data with a 64-bit zero padding, via PSLLDQ and PSRLDQ. Then, we load the 16 rows of the corresponding block of the upper triangular part of $\mathbf{M}$ and we update them, *i.e.* we xor them to the 64-bit data and we store the results in-place on this block. Note that we do not directly load the 16 rows, but we progressively update the block row by row, in order to use less than sixteen 128-bit registers. Now, we repeat this process with the last eight rows of the square block, achieving the process of adding the transpose of the block of the lower triangular part to the corresponding block of the upper triangular part.

Once $\mathbf{M}$ is symmetrized, we load all rows of the upper triangular part and we store them in a new memory zone, in order to store a compact version of the upper triangular matrix (*i.e.* without storing the null elements of the strictly lower triangular part). This transformation is implemented with the AVX2 instruction set (although the AVX instruction set is enough to do it).

For certain implementations of PROV, we could consider to perform the transpose of $\mathbf{P}_i^1$ or $\mathbf{O}$ in the row-major order. The process of adding the transpose of the strictly lower triangular part of $\mathbf{M}$ to its upper triangular part can easily be modified to perform the transpose of a matrix. However, we recommend to be careful if this operation is performed in-place, in particular for rectangular matrices.

4.1.6 Constant-time Gauss-Jordan elimination over $\mathsf{GF}(256)$

During the signing process, we have to solve a linear system of m equations in $\mathsf{m} + \delta$ variables over $\mathsf{GF}(256)$. In [BCH+23, Algorithm 2], the authors use a constant-time Gaussian elimination to solve a square linear system, and abort the process as soon as no pivot is found, *i.e.* if the solution does not exist or is not unique. Therefore, we extend this algorithm to solve overdetermined system. This leads to propose a constant-time Gauss-Jordan elimination (*i.e.* the row echelon form is reduced), because we cannot know if the elimination step is performed with zero or a non-zero pivot without generating a leakage of this information.

Let $\mathbf{A} \in \mathbb{F}^{\mathsf{m} \times (\mathsf{m}+\delta+1)}$ be the augmented matrix associated to the system, and i be the position of the pivot row during the algorithm, initialized to zero. For $0 \leqslant j < \mathsf{m} + \delta$, we use the pivot row for eliminating the coefficients of the j-column of $\mathbf{A}$. To do it, there are several steps.

1. We need to create a non-zero pivot. So, we start by creating a null row $\mathbf{L}$, corresponding to the i-th row of $\mathbf{A}$ if $i < \mathsf{m}$. For $0 \leqslant k < \mathsf{m}$, we add the k-th row of $\mathbf{A}$ to $\mathbf{L}$ if and only if:
   - $k \geqslant i$, in order not to use the pivot rows of the previous iterations,

- we cannot find $\mathbf{A}_{\ell,j} \neq 0$ such that $\ell < k$ and $k \geqslant i$.

These conditional statements are implemented in constant-time with several masks. Note that if $i < \mathsf{m}$, then the i-th row of $\mathbf{A}$ is necessarily added to $\mathbf{L}$.

2. If we have found a non-zero pivot, then we multiply $\mathbf{L}$ by the inverse of the pivot. Else, we multiply $\mathbf{L}$ by any non-zero element of $\mathsf{GF}(256)$, in order not to change the solutions.
3. We use $\mathbf{L}$ for eliminating all rows of $\mathbf{A}$.
4. We replace the i-th row of $\mathbf{A}$ by $\mathbf{L}$. Here, we know that $i \leqslant \min(j, \mathsf{m}-1)$. Therefore, for $0 \leqslant k \leqslant \min(j, \mathsf{m}-1)$, we use a conditional store which erases the k-th row of $\mathbf{A}$ by itself if $k \neq i$, or by $\mathbf{L}$ otherwise.
5. We increment i if and only if a non-zero pivot was found.

Finally, we look if the system is consistent. For $0 \leqslant i < \mathsf{m}$, we perform the logical OR between all elements of the i-th row of $\mathbf{A}$ such that the index of the column is greater or equal to i (note that the other coefficients are necessarily null), and different from $\mathsf{m} + \delta$. Then we generate an integer whose sign bit is one if and only if the previous result is zero and $\mathbf{A}_{i,\mathsf{m}+\delta} \neq 0$. Finally, we perform the logical OR between all integers, and the sign bit is zero if and only if the system is consistent.

We note that with the AVX2 instruction set, the logical OR between the elements of the rows of $\mathbf{A}$ is performed on 256-bit registers. So the result is accumulated in a 256-bit register, but we need to know if at least one of the bytes is not zero. However, we note that the i-th row of $\mathbf{A}$ (without its last coefficient) is not null if and only if there exists an element set to one. For our representation of $\mathsf{GF}(256)$, this is equivalent to having an odd byte. Therefore, we shift the least significant bit of each byte of the accumulator to the most significant bit, then we use the VPMOVMSKB instruction which generates a 32-bit mask from the most significant bit of each byte, and finally we look if this mask is null.

4.1.7 Constant-time backward substitution over $\mathsf{GF}(256)$

In this section, we assume that the augmented matrix $\mathbf{A} \in \mathbb{F}^{\mathsf{m} \times (\mathsf{m}+\delta+1)}$ is in row echelon form such that the pivot is one for non-zero rows. Moreover, we assume that the solution vector is initialized with the choice of free variables. Here, we assume that the field element zero is the null byte, and the field element one is an odd byte.

For i from $\mathsf{m}-1$ to 0, we perform the dot product of the i-th row of $\mathbf{A}$ with the current solution vector, by starting to the i-th position because previous coefficients are necessarily null. We compute the XOR of the previous result and $\mathbf{A}_{i,\mathsf{m}+\delta}$, and we obtain $d \in \mathsf{GF}(256)$. Then, we have to update the variable. To do it, we initialize a mask M to the negation of zero, and we search the first non-zero element of the i-th row. For j from i to $\mathsf{m}+\delta-1$, we generate a mask M_j from $\mathbf{A}_{i,j}$. This mask is zero for a null value, and is the negation of zero otherwise. While we find a null element, M_j is set to zero and we xor (M_j AND M AND d) to x_j, which does not modify its value. When the first non-zero element is found at the position j, we xor (M_j AND M AND d) to x_j, which replaces the free variable by the solution variable. As soon as x_j is found and updated, we set the mask M to zero. Thus, the future XOR instructions will not change the solution vector.

In the optimized implementation, we apply the previous process by taking 8 elements at each step, *i.e.* a 64-bit data D. To generate eight successive values of M_j in parallel, we use the BLSI

instruction which extracts the lowest set bit of D, then we generate a 8-bit mask with this bit. This strategy assumes that the pivot is an odd byte, in order to the lowest set bit found by BLSI is the pivot. The BLSI instruction is not necessary to find the lowest set bit, because we can also find it by computing D AND $-D$. Unfortunately, this strategy cannot be efficiently extended to the SSE2 instruction set, because the computation of $-D$ cannot be directly performed on 128-bit. About the field element d, we duplicate it on 64-bit by multiplying it by 0x101010101010101.

4.1.8 Making the implementation faster for the processor

In the optimized implementation, many algorithms perform operations on matrix rows then accumulate them. The accumulator has the size of the largest row. Our implementation uses the AVX2 instruction set, so the accumulator is a small number of 256-bit registers. When the size of the rows is decreasing during an algorithm, *e.g.* during the Gauss-Jordan elimination, or during vector-matrix products where the matrix is triangular, we specialize the implementation in function of the number of required 256-bit registers. As soon as a register becomes useless, a version of the code without this register is executed.

Then, when the number of registers is small, the potential of the processor is not totally exploited, because the number of identical instructions is small. Therefore, for the vector-matrix products of the signing and verifying processes, we specialize the implementation to perform the product of a vector by two matrices. In this way, the multiplication table of the vector is loaded only one time, and we double the number of identical instructions. The Intel's processors can perform two instructions in parallel for a large number of instructions. We note that this strategy is more efficient than unrolling the for loops.

Finally, during the Gauss-Jordan elimination (Section 4.1.6), the memory zone allocated for each row of the augmented matrix is a multiple of 32 bytes. In this way, each row can be loaded as a small number of 256-bit registers, modified then stored without changing the other rows.

4.2 Performance results

Table 5 presents the performance of PROV in its default mode, using the compact secret key. Table 6 presents the performance when using an expanded secret key. The expanded secret key is much larger (See Section 2.7), but allows for faster signatures when storing an expanded key is acceptable. All results are benchmarked using the optimized AVX2 implementation available on the PROV website. The platform and methodology are presented in Section 4.2.1.

Variant	KeyGen	Sign	Verify
PROV-I	4.41 Mc	0.488 Mc	0.196 Mc
PROV-III	16.4 Mc	1.34 Mc	0.616 Mc
PROV-V	47.5 Mc	2.81 Mc	1.40 Mc

Table 5: Performance of PROV in megacycles, measured with the setup in Section 4.2.1.

Variant	Exp. KeyGen	Exp. Sign	Verify
PROV-I	5.81 Mc	0.150 Mc	0.0902 Mc
PROV-III	22.6 Mc	0.398 Mc	0.281 Mc
PROV-V	64.2 Mc	0.822 Mc	0.649 Mc

Table 6: Performance of PROV in megacycles, using an expanded secret key, measured with the setup in Section 4.2.1.

4.2.1 Platform and benchmarking methodology

computer	processor	cores	frequency	max freq.	architecture
cryptodome	Intel Core i3-8100 CPU	4	3.6 GHz	3.6 GHz	Coffee Lake

Table 7: Processor.

computer	operating system	L1d	L1i	L2	L3	RAM
cryptodome	Ubuntu 20.04.6 LTS	32 KiB	32 KiB	256 KiB	6 MiB	4 GB

Table 8: OS and memory. All cores have the same cache size. The L3 cache is shared.

Tables 7 and 8 summarize the main information about the platform used in the experimental measurements. The measurements used one core of the CPU, and the `C` code was compiled with `gcc -O2 -maes -mavx2 -mpclmul -mbmi`. We used the version 10.5.0 of GCC. Turbo Boost is not used since it is not available. The XKCP (Extended Keccak Code Package) was generated with `make Haswell/libkeccak.a`, enabling the AVX2 instruction set for SHAKE256. We note that the version of XKCP is old. It was downloaded before 2021. This version is provided with the optimized implementation.

For each cryptographic operation of PROV, we run a number of tests such that the elapsed time is greater than 4 seconds, and this time is divided by the number of tests. The signing and verifying processes of PROV were benchmarked with 256 different documents of 32 bytes.

5 Advantages and limitations

5.1 Advantages

Simplicity. Like most UOV-based signature schemes, PROV has a very simple design that is both easy to understand and to implement. Indeed, the only operations that are required in order to generate and verify a signature are matrix multiplications and solving linear systems of equations over the finite field $\mathbb{F}$.

Provable security. PROV can be proven secure both in the ROM and the QROM, under the assumption that the UOV$^-$ problem is hard to solve. Moreover, this hardness assumption has been well-studied since the publication of UOV in [KPG99].

It is worth noting that distinguishing a PROV system with from a uniformly random system of equations is strictly harder than distinguishing a corresponding UOV cryptosystem with with

m equations in $n + \delta$ variables. Indeed, the attacks has access to strictly less information (due to δ missing equations). This point is valid in practice: our parameter sets are expected to resist attacks even if the number of equations was increased to $m + \delta$.

Short signatures. As is often the case in multivariate cryptography (see e.g.[Beu22, BCH+23, FIKT21]), one of the main selling points of PROV is its small signature size. This is especially true when compared to MPC-in-the-head schemes that are directly based on the Multivariate Quadratic problem whose signatures are larger than 10KB for a security level of 128 bits (see e.g. [CHR+16, Beu20]).

5.2 Limitations

Key sizes. Like most other multivariate schemes, the main limitation of PROV is its relatively large public key and expanded secret key size. Moreover, in order to provide a meaningful security proof, we had to increase the dimension of the oil space. Hence, this also lead to a corresponding increase in the number of vinegar variables, which increased both the signature size and the public key sizes when compared with other constructions based on UOV such as [BCH+23].

6 Update history

- June 2023: PROV version 1.0 is submitted to the NIST selection process.
- February 2024: PROV version 1.1 is published. This version corrects an error in the specification of the original 1.0 version, where the secret key seed was not included in the input to the hash function when generating the vinegar vector during signing. The mistake was noticed by River Moreira Ferreira and Ludovic Perret, who showed that it could be exploited to break the scheme. It was also known to the authors, and was scheduled to be corrected in the next version of PROV. In the interest of time, it was eventually decided to release it as a separate update, which constitutes PROV version 1.1. The authors would like to express their gratitude to River Moreira Ferreira and Ludovic Perret, who communicated their findings to us transparently.
- April 2024: PROV version 1.2 is published. This new update brings several new features.
  1. Section 3: new proof techniques based on [CFGM24], which yield sharper security bounds.
  2. Section 4: new optimized implementation for AVX2, together with an in-depth discussion of implementation with the AVX2 instruction set. The new implementation results in vastly better performance (Section 4.2).
  3. Section 2.7: slightly larger parameters to properly account for the computational target of 2^{128} (resp. 2^{192}, 2^{256}) AES computation-equivalents, rather than the same number of basic operations, as required by the NIST call.

  Beside the above features, PROV 1.2 makes a few low-level modifications to the specification, in order to achieve better performance. In particular using AES in place of SHAKE for seed expansion results in a large performance boost. The core PROV design is otherwise unchanged.

References

[BCH+23] Ward Beullens, Ming-Shing Chen, Shih-Hao Hung, Matthias J. Kannwischer, Bo-Yuan Peng, Cheng-Jhih Shih, and Bo-Yin Yang. Oil and vinegar: Modern parameters and implementations. *IACR TCHES*, 2023(3):321–365, 2023.

[Beu20] Ward Beullens. Sigma protocols for MQ, PKP and SIS, and Fishy signature schemes. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part III*, volume 12107 of *LNCS*, pages 183–211. Springer, Heidelberg, May 2020.

[Beu21] 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. Springer, Heidelberg, October 2021.

[Beu22] Ward Beullens. MAYO: Practical post-quantum signatures from oil-and-vinegar maps. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *LNCS*, pages 355–376. Springer, Heidelberg, September / October 2022.

[BFP10] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret. Hybrid approach for solving multivariate systems over finite fields. *J. Math. Crypt*, 2:1–22, 01 2010.

[BMSV22] Emanuele Bellini, Rusydi H. Makarim, Carlo Sanna, and Javier A. Verbel. An estimator for the hardness of the MQ problem. In Lejla Batina and Joan Daemen, editors, *AFRICACRYPT 22*, volume 2022 of *LNCS*, pages 323–347. Springer Nature, July 2022.

[BPB10] Stanislav Bulygin, Albrecht Petzoldt, and Johannes Buchmann. Towards provable security of the unbalanced Oil and Vinegar signature scheme under direct attacks. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 17–32. Springer, Heidelberg, December 2010.

[BY19] 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.

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

[CFGM24] Benoît Cogliati, Pierre-Alain Fouque, Louis Goubin, and Brice Minaud. New security proofs and techniques for Hash-and-Sign with Retry signature schemes. Cryptology ePrint Archive, Paper 2024/609, 2024. https://eprint.iacr.org/2024/609.

[CHR+16] Ming-Shing Chen, Andreas Hülsing, Joost Rijneveld, Simona Samardjiska, and Peter Schwabe. From 5-pass MQ-based identification to MQ-based signatures. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 135–165. Springer, Heidelberg, December 2016.

[CLP+17] Ming-Shing Chen, Wen-Ding Li, Bo-Yuan Peng, Bo-Yin Yang, and Chen-Mou Cheng. Implementing 128-bit secure MPKC signatures. Cryptology ePrint Archive, Report 2017/636, 2017. https://eprint.iacr.org/2017/636.

[FHK+17] Jean-Charles Faugère, Kelsey Horan, Delaram Kahrobaei, Marc Kaplan, Elham Kashefi, and Ludovic Perret. Fast quantum algorithm for solving multivariate quadratic equations. *CoRR*, abs/1712.07211, 2017.

[FIKT21] Hiroki Furue, Yasuhiko Ikematsu, Yutaro Kiyomura, and Tsuyoshi Takagi. A new variant of unbalanced Oil and Vinegar using quotient ring: QR-UOV. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*, pages 187–217. Springer, Heidelberg, December 2021.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 197–206. ACM Press, May 2008.

[IT88] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.

[JCCS19] Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2165–2180. ACM Press, November 2019.

[KPG99] 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. Springer, Heidelberg, May 1999.

[KS98] 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. Springer, Heidelberg, August 1998.

[KX24] Haruhisa Kosuge and Keita Xagawa. Probabilistic Hash-and-Sign with Retry in the quantum random oracle model. The International Conference on Practice and Theory in Public Key Cryptography (PKC), 2024. https://eprint.iacr.org/2022/1359.

[Lan95] G. Landsberg. Über eine Anzahlbestimmung und eine damit Zusammenhangende Reihe. *J. Reine Angew. Math.*, III:87–88, 1895.

[Lev05] A.A Levitskaya. Systems of random equations over finite algebraic structures. *Cybern Syst Anal*, 2005(41):67–93, 2005.

[Pat96] Jacques Patarin. Hidden fields equations (HFE) and isomorphisms of polynomials (IP): Two new families of asymmetric algorithms. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 33–48. Springer, Heidelberg, May 1996.

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

[PS05] Thomas Pornin and Julien P. Stern. Digital signatures do not guarantee exclusive ownership. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *ACNS 05*, volume 3531 of *LNCS*, pages 138–150. Springer, Heidelberg, June 2005.

[PTBW11] 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. Springer, Heidelberg, September / October 2011.

[Ryc21] Jocelyn Ryckeghem. *Cryptographie post-quantique : conception et analyse en cryptographie multivariée*. PhD dissertation, Sorbonne Université, February 2021. English, 284 pages.

[Ryc24] Jocelyn Ryckeghem. High-Performance PROV library (HP-PROV), April 2024. Licensed under the GNU Lesser General Public License, version 3 or later.

[SSH11] Koichi Sakumoto, Taizo Shirai, and Harunaga Hiwatari. On provable security of UOV and HFE signature schemes against chosen-message attack. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 68–82. Springer, Heidelberg, November / December 2011.

[SW16] Peter Schwabe and Bas Westerbaan. Solving binary *MQ* with grover's algorithm. In Claude Carlet, M. Anwar Hasan, and Vishal Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, volume 10076 of *Lecture Notes in Computer Science*, pages 303–322. Springer, 2016.