

UOV: Unbalanced Oil and Vinegar

Algorithm Specifications and Supporting Documentation
Version 3.0

Ward Beullens, Ming-Shing Chen, Jintai Ding, Boru Gong,
Matthias J. Kannwischer, Jacques Patarin, Bo-Yuan Peng,
Dieter Schmidt, Cheng-Jhih Shih, Chengdong Tao, Bo-Yin Yang

uovsig@gmail.com

<https://www.uovsig.org>

August 30, 2026

Changelog (Round 2 → Round 3)

The following main changes have been made in the Round-3 submission of UOV.

First, the concrete security analysis of UOV has been improved: the recently proposed wedge attack is specified, the specification of direct attack is improved, and the memory-aware model, *i.e.*, the MEM-2D model, is proposed so as to consider the implications of memory access onto the cost estimates of PWXL-related attacks against UOV.

Moreover, three recommended parameter sets of UOV, *i.e.*, **uov-Ip**, **uov-III** and **uov-V** are updated, so as to achieve the desired security level well when the (classical) attacks against UOV discussed in Section 4 are taken into consideration even under the standard RAM model. And the performances of these recommended UOV instances are updated accordingly.

Finally, for the **pkc+skc** variant of UOV, the implementation of its signing algorithm is optimized by observing that the computation of $\mathbf{v}^T \mathbf{S}_i$ does not have to have recourse to the computation of $\mathbf{S}_i$ at all, as observed by Amagasa, Ueno and Homma [5].

Changelog (Round 1 → Round 2)

In the Round-1 submission of UOV, seed_{sk} and seed_{pk} were drawn uniformly and *independently*. Instead, now seed_{pk} is derived pseudo-randomly from seed_{sk} , which is achieved by adjusting the procedure `UOV.ExpandSK( $\cdot$ )` to output both $\mathbf{O}$ and seed_{pk} . Therefore, the size of *compressed* secret key of UOV has been reduced by 16 bytes, *i.e.*, from 48 bytes to 32 bytes.

Contents

1	Introduction	4
2	Preliminaries	7
2.1	Notations and Conventions	7
2.2	A Brief Introduction to MPKC	7
2.3	An Overview of UOV	8
3	Specifications	11
3.1	Design Rationale	11
3.2	The UOV Digital Signature Scheme	13
3.3	Data Layout of Keys and Signature	19
3.4	Recommended Parameter Sets	21
4	Concrete Security Analysis	22
4.1	Collision Attack	22
4.2	Direct Attack	24
4.3	Kipnis-Shamir Attack	24
4.4	Intersection Attack	25
4.5	MinRank Attack	25
4.6	Wedge Attack	25
4.7	Memory-Aware Costs of PWXL-related Attacks	27
4.8	Quantum Attacks	28
5	Implementations and Performance	29
5.1	Common Implementation Techniques	29
5.2	x86 AVX2 Implementation	31
5.3	Arm Neon Implementation	33
5.4	Arm Cortex-M4 Implementation	35
5.5	FPGA Implementation	36
6	Summary: Advantages and Limitations	41
	References	42
A	The salt-UOV and Its EUF-CMA Security	49

1 Introduction

This document introduces **Unbalanced Oil and Vinegar (UOV)**, a digital signature scheme using the hash-and-sign paradigm from a trapdoored multivariate quadratic map. Since its introduction in 1999 [57], UOV has resisted all known cryptanalytic attacks for over two decades, demonstrating its remarkable security robustness and long-term reliability.

UOV offers favorable signing performance and compact signatures. Specifically, when it comes to NIST security level 1 [66], UOV offers competitive signing performance and signature sizes compared with standardized post-quantum digital signature schemes such as Dilithium [62], Falcon [75], and SPHINCS+ [52] in the following ways:

- **Signature Size.** UOV signatures are much more compact, and their lengths are significantly shorter than those of the signatures produced by Dilithium, Falcon, and SPHINCS+.
- **Signing Speed.** The `classic` variant of UOV is at least twice as fast as Dilithium, Falcon, and SPHINCS+ when it comes to generating signatures across the benchmark platforms evaluated in this submission, including x86-64 and Armv8-A platforms.
- **Verification Speed.** In terms of verifying signatures, UOV matches the efficiency of Dilithium and significantly surpasses Falcon and SPHINCS+, resulting in lower verification cost.

In summary, UOV is competitive with new NIST standards by most measures, except for public key size; for instance, for NIST security level 1, the `classic` variant of UOV has a public key of 313 KB, which is significantly larger than those of Dilithium, Falcon and SPHINCS+. To address this challenge, we have proposed new variants of UOV with smaller keys (*e.g.*, 45 KB at NIST security level 1), albeit at the cost of verification speed. This document is organized as follows.

§2: Preliminaries. The brief history of MPKC and the general idea behind UOV, together with the notations in this submission, are presented in Section 2.

Multivariate public key cryptosystems (MPKC) date back to the 1980s, and since then many leading cryptographers have been trying to build various types of MPKCs. For instance, two multivariate digital signature schemes, *i.e.*, Rainbow [33] and GeMSS [27], made it into the third round of the NIST PQC competition [1].

In an MPKC, the public/secret key pair is composed of multivariate polynomials, and the hardness of MPKC is firmly connected to the hardness of solving a system of multivariate equations. Years of research show that multivariate polynomials are well suited to building digital signature schemes [22, 27, 38, 57, 69, 72, 73]. Take the UOV signature scheme [57] as an example. Generally speaking, the secret key in UOV is $(\mathcal{F}, \mathcal{T})$, where $\mathcal{F} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ is a *specific* quadratic map and is usually called *central map* due to its critical role in UOV, and the invertible linear transformation $\mathcal{T} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^n$ is used to “hide” the structure of the central map in the public key; moreover, the associated public key is $\mathcal{P} = \mathcal{F} \circ \mathcal{T} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ that consists of a set of multivariate *quadratic* polynomials, *i.e.*,

$$\mathcal{P} = \left(p^{(1)}(x_1, \dots, x_n), p^{(2)}(x_1, \dots, x_n), \dots, p^{(m)}(x_1, \dots, x_n) \right),$$

where

$$p^{(k)}(x_1, \dots, x_n) = \sum_{i=1}^n \sum_{j=i}^n p_{i,j}^{(k)} \cdot x_i x_j + \sum_{i=1}^n p_i^{(k)} \cdot x_i + p_0^{(k)}, \quad k = 1, \dots, m.$$

and all coefficients $p_{i,j}^{(k)}, p_i^{(k)}, p_0^{(k)}$ are taken from the finite field $\mathbb{F}_q$. For cryptographic purposes, the central map $\mathcal{F}$ is carefully designed so that it is “hard” to carry out the

inversion operation of $\mathcal{P}$, but becomes easy once the trapdoor $(\mathcal{F}, \mathcal{T})$ is given; moreover, the hardness of the UOV scheme relies on the UOV assumption that no (even quantum) efficient algorithms can carry out the inversion operation of $\mathcal{P}$ with non-negligible probability.

Given the public/secret key pair $(\mathcal{P}, (\mathcal{F}, \mathcal{T}))$, it is straightforward to build the original UOV scheme [57] according to the hash-and-sign paradigm, as the following shows.

- In the signing algorithm, the given message is first hashed to a target vector $\mathbf{t} \in \mathbb{F}_q^m$, and the secret key $(\mathcal{F}, \mathcal{T})$ enables us to efficiently find a preimage $\mathbf{s} \in \mathbb{F}_q^n$ of $\mathbf{t}$ under the map $\mathcal{P}$; finally, $\mathbf{s}$ is outputted as a valid signature of the given message.
- In the verification algorithm, it accepts the given message-signature pair and outputs **True** if the evaluation of the signature under the public key $\mathcal{P}$ is equal to the hash value of the given message; otherwise, **False** is outputted, indicating the given message-signature pair is deemed invalid.

§3: Specifications. Section 3 specifies the design of our UOV scheme in full detail.

First, we summarize the design rationale behind our UOV digital signature scheme. Then we specify three UOV variants, *i.e.*, **classic**, **pkc** and **pkc+skc**, which offer various tradeoffs between space efficiency and time efficiency, so as to accommodate a variety of efficiency considerations in practice. Furthermore, we specify how objects in UOV are encoded as byte strings in the implementations. We conclude by proposing four recommended parameter sets summarized in Table 1, so as to accommodate different security requirements. The $12 = 3 \times 4$ UOV instances implemented in this submission package correspond precisely to the three UOV variants in combination with the four recommended parameter sets, and the benchmarking results of their implementations over NIST PQC Reference Platform can be found in Table 2.

It is worth noting that given the importance of NIST security level 1 in practice, we propose two recommended parameter sets for UOV, as indicated in Table 1: **uov-Ip** that aims to optimize the size of the (compressed) public key, and **uov-Is** that aims to optimize the size of the signature.

Table 1: Four recommended parameter sets of UOV. The notation **epk** denotes the public key in its *expanded* representation, whereas **cpk** denotes its *compact* representation; similar notations apply to the secret key.

	NIST S.L.	n	m	q	epk (bytes)	esk (bytes)	cpk (bytes)	csk (bytes)	signature (bytes)
uov-Ip	1	119	45	256	321 300	278 087	46 591	32	135
uov-Is	1	160	64	16	412 160	348 704	66 576	32	96
uov-III	3	193	72	256	1 347 912	1 167 440	189 232	32	209
uov-V	5	259	96	256	3 232 320	2 801 024	446 992	32	275

§4: Concrete security analysis. Section 4 is devoted to the security analysis of UOV.

The security analysis for UOV is usually carried out by enumerating all the known attacks against UOV that may influence its concrete hardness, as is done in this document. Generally speaking, our confidence in the security of the UOV digital signature scheme lies in the following facts: UOV remains secure after more than two decades of cryptanalysis, and the theoretical hardness estimates for UOV are consistent with experimental results.

We present in Section 4 those well-known attacks against UOV, including the collision attack, the direct attack, the Kipnis-Shamir attack [58], the intersection attack [14], an improved MinRank attack [7], as well as the recently proposed wedge attack [80, 54, 44]. Taking all those attacks into consideration, we choose the foregoing four recommended

Table 2: Benchmarking results of AVX2 implementations of UOV. The performance numbers are measured on Intel Xeon E3-1230L v3 1.80GHz (Haswell) and Intel Xeon E3-1275 v5 3.60GHz (Skylake) with turbo boost and hyper-threading disabled. The performance numbers are the median CPU cycles of 1000 signing and verification operations, and of 100 key generations.

	KeyGen	Haswell Sign	Verify	KeyGen	Skylake Sign	Verify
uov-Ip-classic	3 587 232	120 984	83 724	3 275 886	117 422	74 252
uov-Ip-pkc	3 537 640		354 036	3 189 084		252 876
uov-Ip-pkc+skc	3 523 032	573 328		3 218 816	471 258	
uov-Is-classic	4 895 116	136 712	66 408	4 253 780	120 526	60 660
uov-Is-pkc	4 846 220		409 428	4 314 950		286 888
uov-Is-pkc+skc	4 362 000	678 528		3 978 980	520 234	
Dilithium 2 [†] [62]	97 621*	281 078*	108 711*	70 548	194 892	72 633
Falcon-512 [77]	19 189 801*	792 360*	103 281*	26 604 000	948 132	81 036
SPHINCS+ [‡] [52]	1 334 220	33 651 546	2 150 290	1 510 712*	50 084 397*	2 254 495*
uov-III-classic	22 512 648	326 116	245 868	18 459 512	299 814	216 772
uov-III-pkc	22 337 004		1 383 436	18 360 852		967 100
uov-III-pkc+skc	23 996 620	2 111 064		18 851 832	1 628 832	
uov-V-classic	66 626 076	673 972	463 424	53 227 288	648 574	433 102
uov-V-pkc	65 580 064		3 183 152	51 692 258		2 229 086
uov-V-pkc+skc	59 292 764	4 499 944		48 217 758	3 467 808	

[†] Security level II. [‡] Sphincs+-SHA2-128f-simple. * Data from SUPERCOP [40].

parameter sets presented in Table 1 so that they can achieve the desired NIST security levels in the usual RAM model, respectively. Also the cost estimates of those PWXL-related attacks are also considered under the MEM-2D model [35], and this sensitivity analysis provides additional confidence in our choice of the recommended UOV parameter sets.

§5: Implementations. To fully demonstrate the strengths of UOV in practice, we describe in Section 5 the implementations of $12 = 3 \times 4$ UOV instances among many popular platforms, together with the experimental results. Please refer to [11] for the comprehensive and detailed information regarding our implementations.

First come several implementation techniques shared among our software implementations, introduced in Section 5. Then we present our optimization for x86-64 platforms, *i.e.*, the reference platform in NIST PQC standardization [67]. More precisely, we focus on the optimization for the AVX2 instruction set due to its availability on modern x86 platforms. In addition to the Intel Haswell microarchitecture specifically required by NIST, we also implement our UOV recommended instances in the Intel Skylake microarchitecture with better performance. And our experimental results for x86-64 platforms are summarized in Table 2.

Besides, we also present the optimization of UOV for the Armv8-A architecture in Section 5.3, the optimization of UOV for the Arm Cortex-M4 in Section 5.4, as well as the optimization of UOV on the popular FPGA platforms in Section 5.5.

§6: Advantages and limitations. The advantages and limitations of UOV are summarized in Section 6.

2 Preliminaries

2.1 Notations and Conventions

Let λ denote the security parameter in this document. For the binary strings $x, y \in \{0, 1\}^*$, the notation $x||y$ denotes their concatenation. All logarithms in this document are to the base 2. When k is a positive integer, let $[k]$ denote the index set $\{1, 2, \dots, k\}$. For the *finite* set S , let $x \leftarrow S$ denote the process of sampling an element from S uniformly at random and assigning it to the variable x . For a *possibly randomized* algorithm A , let the notation $y \leftarrow A(x)$ denote the process of running A on input x , and assigning the output to the variable y ; in particular, when the algorithm A is *deterministic*, the notation $y := A(x)$ is applied for emphasis. The expression $A == B$ evaluates to **True** if the given objects A and B are equal, and to **False** otherwise.

Throughout this document, q always denotes a prime power, and $\mathbb{F}_q$ denotes a finite field with q elements; all polynomials, vectors and matrices are defined over $\mathbb{F}_q$. By convention, vectors are assumed to be in column form and are written using bold lower-case letters, whereas matrices are written as bold capital letters, and $(\cdot)^T$ denotes the matrix transposition operation; in particular, $\mathbf{0}_k$ denotes the k -dimensional zero vector $[0, 0, \dots, 0]^T$, and $\mathbf{I}_k$ denotes the k -by- k identity matrix (over $\mathbb{F}_q$). The notation $[a_i]_{i \in [k]}$ represents a k -dimensional column vector whose i -th coordinate is a_i , and the subscript can be omitted when the index and the dimension are clear from the context.

A digital signature scheme $\Pi = (\text{KeyGen}, \text{Sign}, \text{Verify})$ usually consists of three probabilistic polynomial-time algorithms:

- $(\text{pk}, \text{sk}) \leftarrow \text{KeyGen}(1^\lambda, \text{params})$. **KeyGen** is the key generation algorithm that, on input the security parameter λ (in unary form) as well as the system parameter **params**, outputs a public key **pk** and its associated secret key **sk**. Here, the security parameter λ is usually implicit in **params** and hence omitted for simplicity.
- $\sigma \leftarrow \text{Sign}(\text{sk}, \mu)$. **Sign** is the signing algorithm that, on input the secret key **sk** and the message $\mu \in \{0, 1\}^*$ to be signed, outputs a signature σ .
- $b := \text{Verify}(\text{pk}, \mu, \sigma)$. **Verify** is the *deterministic* verification algorithm that, on input the public key **pk** and the message-signature pair (μ, σ) , outputs $b \in \{\text{True}, \text{False}\}$, indicating whether it accepts the signature σ as a valid signature on μ for the public key **pk** (*i.e.*, $b = \text{True}$) or not (*i.e.*, $b = \text{False}$).

We say a digital signature scheme $\Pi = (\text{KeyGen}, \text{Sign}, \text{Verify})$ is *correct*, if for any sufficiently large λ , it holds that

$$\Pr[\text{Verify}(\text{pk}, \mu, \text{Sign}(\text{sk}, \mu)) = \text{True}] = 1 - \text{negl}(\lambda),$$

where the probability is taken over the randomness of the key generation and signing algorithms, and $\text{negl}(\lambda)$ denotes a function that is negligible in the security parameter λ . Moreover, the *standard* security definition [55] for a digital signature scheme requires that it should be existentially unforgeable under chosen-message attack, or of EUF-CMA security for short; roughly speaking, the existential unforgeability requirement on Π states that, given a public key **pk** and access to a signing oracle that outputs $\text{Sign}(\text{sk}, \mu)$ (where **sk** is the secret key corresponding to **pk**) for any given message $\mu \in \{0, 1\}^*$, every computationally bounded adversary is unable to come up with a valid signature for a new message μ' that was not previously fed to the signing oracle, except with negligible probability.

2.2 A Brief Introduction to MPKC

The multivariate public key cryptosystems (MPKC) are a family of candidate post-quantum cryptographic schemes. Roughly speaking, its public/secret key pair is composed

of multivariate polynomials, and the hardness of MPKC is firmly connected to the hardness of solving a system of multivariate equations. The idea of MPKC dates back to 1980s, and many leading cryptographers (Ong, Schnorr, Matsumoto, Imai, Harashima, Diffie, Fell, Miyagawa, Tsujii, Kurosawa, Fujioka and others) built various types of MPKCs [22, 27, 38, 57, 69, 72, 73]. However, the linearization equations attack proposed by Jacques Patarin [71] against the Matsumoto-Imai cryptosystem provided the major impetus for the development of MPKC theory.

In a multivariate public-key cryptosystem, the public key $\mathcal{P}$ consists of a sequence of m multivariate polynomials in n variables $\mathbf{x} = [x_i]_{i \in [n]}$, and hence could be seen as a *nonlinear* map from $\mathbb{F}_q^n$ to $\mathbb{F}_q^m$, where n, m, q are public parameters. Polynomials in $\mathcal{P}$ are usually quadratic in practice, which explains why MPKCs are often referred to as Multivariate Quadratic (MQ) cryptosystems; in this case, we have

$$\mathcal{P} = \left(p^{(1)}(x_1, \dots, x_n), p^{(2)}(x_1, \dots, x_n), \dots, p^{(m)}(x_1, \dots, x_n) \right),$$

where

$$p^{(k)}(x_1, \dots, x_n) = \sum_{i=1}^n \sum_{j=i}^n p_{i,j}^{(k)} \cdot x_i x_j + \sum_{i=1}^n p_i^{(k)} \cdot x_i + p_0^{(k)}, \quad k \in [m],$$

and all coefficients $p_{i,j}^{(k)}, p_i^{(k)}, p_0^{(k)}$ are taken from $\mathbb{F}_q$.

For cryptographic purposes, $\mathcal{P}$ should be carefully chosen so that it works like a trapdoored one-way function. On the one hand, the evaluation operation $\mathbf{x} \mapsto \mathcal{P}(\mathbf{x})$ associated with $\mathcal{P}$ is obviously efficient. On the other hand, the time complexity of the inversion operation associated with $\mathcal{P}$, as well as the security of the associated public-key cryptosystems, is firmly connected to the hardness of the following problem.

Definition 1 (MQ problem). Given $(n, m, q, \mathcal{P})$ where $\mathcal{P} = (p^{(1)}, p^{(2)}, \dots, p^{(m)}) : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ is a multivariate quadratic map, find an n -dimensional vector $\mathbf{u} \in \mathbb{F}_q^n$ such that

$$p^{(1)}(\mathbf{u}) = p^{(2)}(\mathbf{u}) = \dots = p^{(m)}(\mathbf{u}) = 0 \in \mathbb{F}_q.$$

In general, the MQ problem is proven to be NP-complete on every finite field $\mathbb{F}_q$ [76], and the proof is particularly simple and direct when $q = 2$ [49]. Concretely, efficient algorithms are known [13, 28, 31, 41, 42, 47, 63] when $m \gg n$ or $m \ll n$, and the most difficult instances of the MQ problem are obtained when m and n are of the same order of magnitude. It is also interesting to note that very often the best known algorithms on the MQ problem have a similar complexity for worst cases and for average cases, which is also true for quantum algorithms. Until now, no efficient quantum algorithm against the MQ problem has been found; taking the NP-completeness of MQ into consideration, it is generally believed that this will still be the case in the future.

When $m \geq n$, we can construct a public-key encryption scheme based on the map $\mathcal{P}$, which is similar to the “Plain RSA” encryption scheme [56]. Conversely, when $m < n$, the multivariate polynomials are well suited to building secure digital signature schemes using hash-and-sign paradigm.

2.3 An Overview of UOV

The history of UOV scheme [57], as well as its variants, could be traced back to Patarin’s linearization equations attack [71] in 1995 against the Matsumoto-Imai cryptosystem. Two years later, Patarin converted the idea behind this attack into the design of Oil and Vinegar (OV) signature scheme [72] in 1997. After the *balanced* version of this scheme, *i.e.*, $n - m = m$, was broken by an invariant subspace attack [58] in 1998, Kipnis, Patarin and

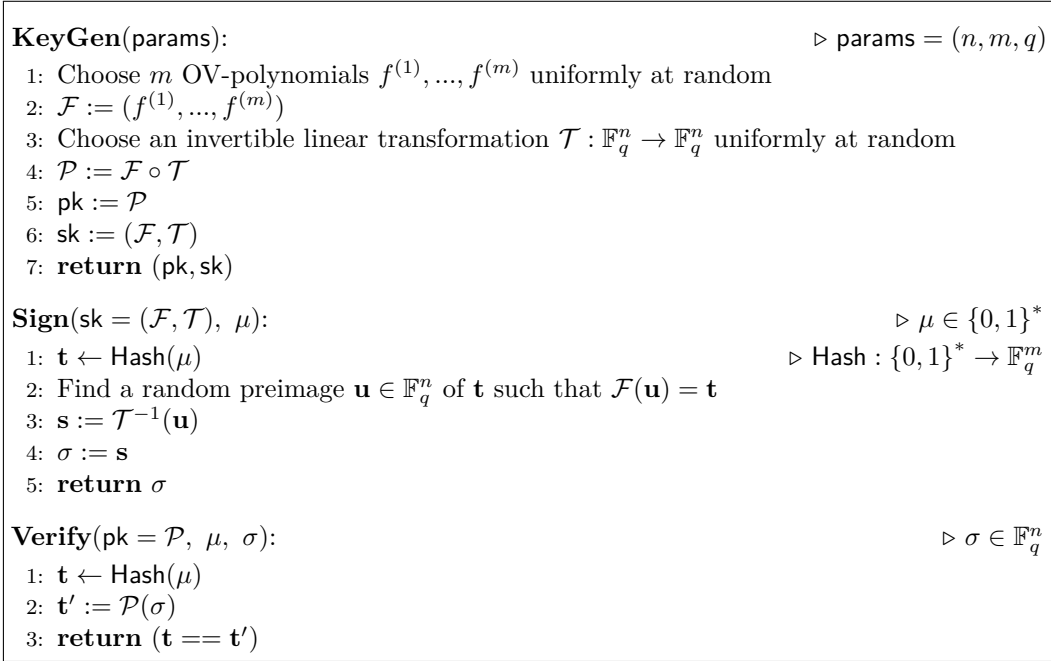


Figure 1: The key generation, signing, and verification algorithms of the original UOV [57].

Goubin proposed the Unbalanced Oil and Vinegar (UOV) digital signature scheme [57] in 1999, where $n - m > m$, or equivalently, $n > 2m$. The simplicity in the UOV design, and the fact that no fundamental improvement on attacks against UOV has been made after more than twenty years of cryptanalysis give us the confidence in the security of the UOV digital signature scheme.

Original UOV. When the multivariate quadratic map $\mathcal{P} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ is applied for public-key cryptographic purposes, $\mathcal{P}$ is the public key, and it should be well designed so that we can generate its associated trapdoor td , which enables the *efficient* inversion operation of $\mathcal{P}$ and hence serves as the associated secret key. With the desired key pair $(\mathcal{P}, \text{td})$, we can construct a digital signature scheme by following the hash-and-sign paradigm:

- For the key generation algorithm, given the security parameter, it outputs a random key pair $(\text{pk} = \mathcal{P}, \text{sk} = \text{td})$.
- For the signing algorithm, given the secret key $\text{sk} = \text{td}$ as well as a message $\mu \in \{0, 1\}^*$ to be signed, it first computes $\mathbf{t} \leftarrow \text{Hash}(\mu)$, and then find a preimage $\mathbf{s} \in \mathbb{F}_q^n$ of $\mathbf{t}$ with the aid of td ; finally, it returns the signature $\sigma := \mathbf{s}$. Here $\text{Hash} : \{0, 1\}^* \rightarrow \mathbb{F}_q^m$ denotes a hash function.
- For the verification algorithm, given the public key $\text{pk} = \mathcal{P}$ and a message-signature pair $(\mu, \sigma) \in \{0, 1\}^* \times \mathbb{F}_q^n$, it simply computes $\mathbf{t}' = \mathcal{P}(\sigma) \in \mathbb{F}_q^m$, and returns **True** if and only if the equality $\mathbf{t}' = \text{Hash}(\mu)$ holds; otherwise, it returns **False**, indicating that (μ, σ) is deemed an invalid message-signature pair.

Similar to FDH [23], its security clearly is firmly connected to the design of the key pair $(\mathcal{P}, \text{td})$ as well as the choice of parameters. For instance, in the original UOV digital signature scheme [57] depicted in Figure 1, we have $n > 2m$, $\text{td} = (\mathcal{F}, \mathcal{T})$, and the public key $\mathcal{P}$ is the composite of the maps $\mathcal{F}$ and $\mathcal{T}$, where:

- The *central map* $\mathcal{F} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ is a *special* multivariate quadratic map that consists of m quadratic polynomials $f^{(1)}, \dots, f^{(m)}$ in n variables; concretely, each polynomial $f^{(k)}$ is in the form of

$$f^{(k)}(x_1, \dots, x_n) = \sum_{i=1}^v \sum_{j=i}^n \alpha_{i,j}^{(k)} \cdot x_i x_j,$$

where $v = n - m$. In the literature, the v variables $x_1, \dots, x_v$ in the UOV scheme are called the *vinegar variables*, the remaining m variables $x_{v+1}, \dots, x_n$ are called the *oil variables*, and these m quadratic polynomials are usually referred to as *oil-vinegar polynomials*, or simply *OV-polynomials*;

- $\mathcal{T} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^n$ is an *invertible* linear transformation, which is used to hide the structure of the central map $\mathcal{F}$ in $\mathcal{P}$.

Inversion operation of $\mathcal{P}$. To finish the efficiency analysis of the signing algorithm in original UOV, it remains to show that the inversion operation of $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$ is efficiently computable, provided $(\mathcal{F}, \mathcal{T})$ is given; furthermore, since $\mathcal{T}$ is an invertible linear transformation, it suffices to show that given $\mathcal{F}$, the inversion operation of $\mathcal{F}$ is efficiently computable. This is indeed true, as the following analysis indicates.

For a randomly chosen $\mathcal{F} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ and every uniformly chosen vector $\mathbf{u} \in \mathbb{F}_q^v$, they correspond to a linear transformation $\mathcal{F} \left(\begin{bmatrix} \mathbf{u} \\ \cdot \end{bmatrix} \right)$ on $\mathbb{F}_q^m$; thus, with Gaussian elimination we can recover, if possible, a preimage $\begin{bmatrix} \mathbf{u} \\ \mathbf{w} \end{bmatrix} \in \mathbb{F}_q^n$ of $\mathbf{t}$ under $\mathcal{F}$, and hence a preimage $\mathbf{s} \in \mathbb{F}_q^n$ of $\mathbf{t}$ under $\mathcal{P}$, where $\mathbf{s} = \mathcal{T}^{-1} \left(\begin{bmatrix} \mathbf{u} \\ \mathbf{w} \end{bmatrix} \right)$; moreover, since the induced linear transformation $\mathcal{F} \left(\begin{bmatrix} \mathbf{u} \\ \cdot \end{bmatrix} \right)$ on $\mathbb{F}_q^m$ is full-rank with probability approximately $1 - 1/q$, polynomial number of random attempts on the choice of $\mathbf{u}$ enables us to recover a random preimage for $\mathbf{t} \in \mathbb{F}_q^m$, except with negligible probability. This shows that $\mathcal{F}$ is indeed efficiently invertible.

Remarks. The invertible linear transformation $\mathcal{T}$ could be generalized to the affine invertible map onto $\mathbb{F}_q^n$; nevertheless, this generalization does not contribute to the security of UOV, as demonstrated in [24]. Similarly, in the central map $\mathcal{F}$, every OV-polynomial $f^{(k)}$ depicted previously is *homogeneous*, and an inhomogeneous generalization on $f^{(k)}$ does not seem to improve the security of UOV significantly.

UOV is *unbalanced* in the sense that we have more vinegar variables than oil variables, which is in sharp contrast to the *original* (balanced) OV scheme [72], where we have the same number of vinegar variables and oil variables.

In UOV, the key point lies in the design of the central map $\mathcal{F}$, where the oil variables never *mix* with oil variables in every OV-polynomial $f^{(k)}$, and (U)OV bears its name from this crucial design exactly. As indicated earlier, this design is crucial for the efficiency of the signing algorithm as well. Conversely, the mixing of these two types of variables in $\mathcal{P}$ is achieved via the introduction of the invertible map $\mathcal{T} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^n$, so as to guarantee the hardness of the inversion operation of $\mathcal{P}$. After more than twenty years of security analysis, it has been shown that *when parameters are appropriately chosen*, $\mathcal{P}$ is indeed “hard” to invert on average in the absence of the trapdoor $(\mathcal{F}, \mathcal{T})$, implying that $\mathcal{P}$ is a promising candidate of *trapdoored* one-way function.

3 Specifications

In this section we present the design of our UOV digital signature scheme in full detail.

Generally speaking, the sizes of public/secret keys in digital signature schemes may exert a direct impact on the practical utility of the cryptographic system. This is particularly crucial in resource-constrained scenarios. Conversely, as indicated in [Section 1](#), the (public) key size of UOV is usually much larger than that of other types of post-quantum digital signatures [52, 62, 75]. This drives us to design UOV so that its public key can be represented either in the expanded form, *i.e.*, **epk**, or in the compressed form, *i.e.*, **cpk**, and the public key expansion operation is applied when necessary; similar considerations apply to the secret key as well. The flexibility in the public/secret key representation enables us to design three variants of UOV, as the following summarizes.

- In the **classic** variant of UOV, it takes the expansion of public/secret keys as part of the key-generation algorithm, and hence it has larger public/secret key sizes but faster signing and verification speed.
- In the public-key-compressed **pkc** variant of UOV, the public key expansion is carried out in the verification algorithm, which decreases the public key size significantly and at the cost of slower verification speed.
- Compared with the **pkc** variant of UOV, the doubly-compressed variant **pkc+skc** goes further and also considers the secret key expansion to be part of the signing algorithm, which results in tiny secret keys but slower signing speed.

Note that the three UOV variants are *interoperable*, in the sense that a signature produced by the signing algorithm of one variant can be verified by the verification algorithm of the other ones.

This section is organized as follows. First, we present the design rationale behind our UOV digital signature scheme in [Section 3.1](#). [Section 3.2](#) is devoted to the specification of our UOV digital signature scheme: in addition to the system parameter and the choice of symmetric primitives, five procedures are also specified in pseudocode that serve as the building blocks of three variants of UOV. After discussing the data layout of UOV in [Section 3.3](#), we conclude in [Section 3.4](#) by proposing four recommended parameter sets for UOV specified in [Table 4](#). In [Section 5](#) we shall implement these $12 = 3 \times 4$ UOV instances that correspond precisely to the three UOV variants in combination with the four recommended parameter sets.

3.1 Design Rationale

First, we present the design rationale behind our UOV digital signature scheme, in comparison with the original UOV scheme [57].

About the trapdoor. In the original UOV scheme presented in [Section 2.3](#), given the pair $(\mathcal{F}, \mathcal{T})$, we can efficiently recover a preimage $\begin{bmatrix} \mathbf{u} \\ \mathbf{w} \end{bmatrix} \in \mathbb{F}_q^n$ of $\mathbf{t} \in \mathbb{F}_q^m$ under $\mathcal{F}$, and hence a preimage $\mathbf{s} \in \mathbb{F}_q^n$ of $\mathbf{t}$ under the $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$, where

$$\mathbf{s} = \mathcal{T}^{-1} \left(\begin{bmatrix} \mathbf{u} \\ \mathbf{w} \end{bmatrix} \right) = \mathcal{T}^{-1} \left(\begin{bmatrix} \mathbf{u} \\ \mathbf{0}_m \end{bmatrix} \right) + \mathcal{T}^{-1} \left(\begin{bmatrix} \mathbf{0}_v \\ \mathbf{w} \end{bmatrix} \right),$$

where $v = n - m$. Let $\mathbf{e}_i = [\delta_{i,j}]_{j \in [n]}$ be the constant vector in $\mathbb{F}_q^n$ where $\delta_{\cdot, \cdot}$ denotes the Kronecker delta function, and let $O \subseteq \mathbb{F}_q^n$ be the column space of the matrix $[\mathcal{T}^{-1}(\mathbf{e}_{v+1}), \dots, \mathcal{T}^{-1}(\mathbf{e}_n)] \in \mathbb{F}_q^{n \times m}$ that is often referred to as the *oil space* in the literature, *i.e.*, $O = \text{span}(\mathcal{T}^{-1}(\mathbf{e}_{v+1}), \dots, \mathcal{T}^{-1}(\mathbf{e}_n))$. As the foregoing analysis shows that the efficient

inversion of $\mathcal{P}$ relies on the efficient sampling in the oil space O , the trapdoor information associated with $\mathcal{P}$ could be seen as a “short” description of the subspace O , *i.e.*, an $\mathbb{F}_q$ -basis for O that can be represented by a matrix in $\mathbb{F}_q^{n \times m}$. As almost every m -dimensional subspace of $\mathbb{F}_q^n$ could be seen as the column space of a matrix in the form of $\begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix}$, where $\mathbf{O} \in \mathbb{F}_q^{v \times m}$, the trapdoor of $\mathcal{P}$ could be re-defined as the matrix $\mathbf{O} \in \mathbb{F}_q^{v \times m}$ [91]. Clearly this does not reduce the key space of UOV much, but can decrease the size of the secret key of UOV significantly.

About the public key. In the key generation algorithm, after the new trapdoor $\mathbf{O} \leftarrow \mathbb{F}_q^{v \times m}$ as well as $\overline{\mathbf{O}} = \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix}$ is chosen, it remains to generate an associated $\mathcal{P}$ by generating a sequence of multivariate quadratic polynomials $p_1, \dots, p_m$ that vanish on the oil space O determined by $\mathbf{O}$. Nevertheless, inspired by the design of the CyclicRainbow scheme [73, 74], in our UOV digital signature scheme the process of generating a public key from the secret key could be *partially reversible*, as the following analysis shows.

Each (homogeneous) multivariate quadratic polynomial p_i can be *uniquely* represented by an upper-triangular matrix $\mathbf{P}_i \in \mathbb{F}_q^{n \times n}$ such that $p_i(\mathbf{x}) = \mathbf{x}^\top \mathbf{P}_i \mathbf{x}$. Let

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

where $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{v \times v}$, $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{m \times m}$ are both upper-triangular, and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{v \times m}$. Then the quadratic polynomial p_i vanishes on the oil space O (*i.e.*, the column space of $\overline{\mathbf{O}}$) if and only if the matrix

$$[\mathbf{O}^\top \quad \mathbf{I}_m] \mathbf{P}_i \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix} = \mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} + \mathbf{O}^\top \mathbf{P}_i^{(2)} + \mathbf{P}_i^{(3)} \in \mathbb{F}_q^{m \times m}$$

is alternating.

Thus, given the trapdoor $\mathbf{O}$, we can generate a set of random matrices $\{\mathbf{P}_i\}_{i \in [m]}$, which characterizes the public key $\mathcal{P} = (p_1, \dots, p_m)$, as follows: first, pick m matrices $\mathbf{P}_i^{(2)} \leftarrow \mathbb{F}_q^{v \times m}$ and m upper-triangular matrices $\mathbf{P}_i^{(1)} \leftarrow \mathbb{F}_q^{v \times v}$ uniformly at random (note that this operation is *independent* of $\mathbf{O}$); then, compute

$$\mathbf{P}_i^{(3)} := \text{Upper} \left(-\mathbf{O}^\top \mathbf{P}_i^{(1)} \mathbf{O} - \mathbf{O}^\top \mathbf{P}_i^{(2)} \right), \quad i \in [m].$$

Here, $\text{Upper}(\mathbf{M})$ denotes the *unique* upper-triangular matrix $\mathbf{M}'$ such that the difference $\mathbf{M}' - \mathbf{M}$ is alternating; by definition, the function $\text{Upper}(\cdot)$ is *deterministic* and can be computed in polynomial time.

About the inversion operation. For the multivariate quadratic map $\mathcal{P} = (p_1, \dots, p_m)$ determined by $\{\mathbf{P}_i\}_{i \in [m]}$, together with its new trapdoor $\overline{\mathbf{O}} = \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix}$, we can improve the process of calculating a preimage $\mathbf{s} \in \mathbb{F}_q^n$ for a given $\mathbf{t} = [t_i]_{i \in [m]} \in \mathbb{F}_q^m$ as follows: first pick a random *vinegar vector* $\mathbf{v} \leftarrow \mathbb{F}_q^v$, and then try to recover a preimage $\mathbf{s}$ for $\mathbf{t}$ by calculating an appropriate $\mathbf{x} \in \mathbb{F}_q^m$, where $\mathbf{s}$ is of the following specific form

$$\mathbf{s} = \begin{bmatrix} \mathbf{v} \\ \mathbf{0}_m \end{bmatrix} + \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix} \cdot \mathbf{x}.$$

It remains to compute $\mathbf{x}$. First, it is routine to see

$$\mathcal{P}(\mathbf{s}) = \mathbf{t} \quad \text{if and only if} \quad \mathbf{v}^\top \mathbf{S}_i \cdot \mathbf{x} = t_i - y_i \quad \text{for every } i \in [m],$$

where $y_i = \mathbf{v}^\top \mathbf{P}_i^{(1)} \mathbf{v}$ is the evaluation of p_i at $\bar{\mathbf{v}} = \begin{bmatrix} \mathbf{v} \\ \mathbf{0}_m \end{bmatrix}$, and $\mathbf{S}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top})\mathbf{O} + \mathbf{P}_i^{(2)} \in \mathbb{F}_q^{v \times m}$. Let $\mathbf{L} \in \mathbb{F}_q^{m \times m}$ be the square matrix with the i -th row being $\mathbf{v}^\top \mathbf{S}_i$, and $\mathbf{y} = \mathcal{P}(\bar{\mathbf{v}}) = [y_i]_{i \in [m]} \in \mathbb{F}_q^m$. Then the foregoing argument can be simply rephrased as:

$$\mathcal{P}(\mathbf{s}) = \mathbf{t} \quad \text{if and only if} \quad \mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}.$$

When $\mathbf{L}$ is invertible (with probability approximately $1 - 1/q$), we can easily recover a desired $\mathbf{s}$ by calculating its corresponding $\mathbf{x} = \mathbf{L}^{-1} \cdot (\mathbf{t} - \mathbf{y})$; otherwise, repeat the foregoing process with a fresh new $\mathbf{v} \leftarrow \mathbb{F}_q^v$. Similar analysis shows that we can obtain a desired preimage $\mathbf{s}$ after very few attempts, when parameters are appropriately chosen.

Note that the *intermediate* matrices $\mathbf{S}_i \in \mathbb{F}_q^{v \times m}$ depend only on the public/secret key pair, and are *independent* of the message to be signed. Since these m intermediate matrices $\mathbf{S}_i$'s are relatively expensive to compute, it is optional to define them to be part of the *expanded* secret key and compute them only once in the key generation algorithm, which would improve the time efficiency of the signing algorithm.

About the key representation. For the implementations of cryptosystems, when the key size is of particular interest, it is customary to use short seeds to replace part of the key, with the use of the cryptographic pseudo-random number generator (PRNG).

Thus, two random seeds are introduced into our UOV digital signature scheme. First comes seed_{pk} that is used to expand the bulk of the public key deterministically via the PRNG $\text{Expand}_{\mathbf{P}}(\cdot)$ (to be defined) as follows:

$$\text{Expand}_{\mathbf{P}} : \text{seed}_{\text{pk}} \mapsto \{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i \in [m]}.$$

Moreover, we can define a short seed seed_{sk} that is used to expand the trapdoor $\mathbf{O}$ as well as seed_{pk} deterministically via the PRNG $\text{Expand}_{\text{sk}}(\cdot)$ as follows:

$$\text{Expand}_{\text{sk}} : \text{seed}_{\text{sk}} \mapsto (\text{seed}_{\text{pk}}, \mathbf{O}).$$

3.2 The UOV Digital Signature Scheme

This section is devoted to the specification of our UOV digital signature scheme.

3.2.1 System parameter of UOV

Our UOV digital signature scheme is parameterized by the following values

$$\text{params} = (n, m, q, \text{salt_len}, \text{pk_seed_len}, \text{sk_seed_len}),$$

where

- n denotes the number of variables in the multivariate quadratic polynomials in the public key;
- m denotes the number of multivariate quadratic polynomials in the public key;
- q denotes the size of a finite field $\mathbb{F}_q$;
- salt_len denotes the bit length of a binary string $\text{salt} \in \{0, 1\}^{\text{salt_len}}$;
- pk_seed_len denotes the bit length of the binary string $\text{seed}_{\text{pk}} \in \{0, 1\}^{\text{pk_seed_len}}$ used to expand (the bulk of) the public key; and
- sk_seed_len denotes the bit length of the binary string $\text{seed}_{\text{sk}} \in \{0, 1\}^{\text{sk_seed_len}}$ used to expand the secret key.

For the recommended parameter sets proposed in this submission, we always have $q \in \{16, 256\}$, $\text{salt_len} = 128$, $\text{pk_seed_len} = 128$, and $\text{sk_seed_len} = 256$. Also it is customary to let v denote the number of vinegar variables for convenience, *i.e.*, $v = n - m$.

3.2.2 Choice of symmetric primitives in UOV

Here we define a variety of symmetric primitives that are needed for the specification of our UOV digital signature scheme. The required primitives are as follows.

- **Hash** : $(\mu, \text{salt}) \mapsto \mathbf{t}$. Given a message $\mu \in \{0, 1\}^*$ and a salt $\in \{0, 1\}^{\text{salt_len}}$, it maps to the target vector $\mathbf{t} \in \mathbb{F}_q^m$.
- **Expand_v** : $(\mu, \text{salt}, \text{seed}_{\text{sk}}, \text{ctr}) \mapsto \mathbf{v}$. Given a message μ , a salt, the $\text{seed}_{\text{sk}} \in \{0, 1\}^{\text{sk_seed_len}}$ for the secret key and a counter $\text{ctr} \in \{0, 1\}^8$, it computes a vinegar vector $\mathbf{v} \in \mathbb{F}_q^v$. (In security analysis, **Expand_v**($\cdot$) is modeled as a pseudorandom function keyed by seed_{sk} .)
- **Expand_{sk}** : $\text{seed}_{\text{sk}} \mapsto (\text{seed}_{\text{pk}}, \mathbf{O})$. It expands the seed_{sk} for the secret key to the $\text{seed}_{\text{pk}} \in \{0, 1\}^{\text{pk_seed_len}}$ for the public key, as well as the matrix $\mathbf{O} \in \mathbb{F}_q^{v \times m}$ in column-major order.
- **Expand_P** : $\text{seed}_{\text{pk}} \mapsto (\{\mathbf{P}_i^{(1)}\}_{i \in [m]}, \{\mathbf{P}_i^{(2)}\}_{i \in [m]})$. It expands the seed_{pk} for the public key to the matrices $\{\mathbf{P}_i^{(1)}\}_{i \in [m]}$ and $\{\mathbf{P}_i^{(2)}\}_{i \in [m]}$. In our implementations, **Expand_P**($\cdot$) is instantiated with **aes128ctr** using the seed as the key and a zero nonce.

For **Hash**($\cdot$), **Expand_v**($\cdot$), and **Expand_{sk}**($\cdot$), their performances are not critical, and hence we instantiate them with **shake256** [46]. Conversely, the performance of **Expand_P**($\cdot$) has a high impact on the performance of the overall UOV scheme. The input and output of this function are public, so the instantiation of this function does not need to be side-channel resistant. Therefore, we instantiate **Expand_P**($\cdot$) with **aes128** [45] because this results in much faster implementations. Note that we do not require **Expand_P**($\cdot$) to be a cryptographically secure stream cipher. We (optionally) propose to use **aes128ctr** with 4, instead of 10, rounds, mostly because 4-round **aes128** has been proven to have a maximal differential probability of 2^{-114} [60] that is deemed sufficient for the purpose of public-key expansion in UOV.

3.2.3 Five Procedures for UOV

We specify five procedures for our UOV signature scheme, and their pseudocode can be found in Figure 2. These procedures are used to define the three UOV variants described below.

- **UOV.CompactKeyGen**: $\text{params} \mapsto (\text{cpk}, \text{csk})$. Given the parameter **params**, it outputs a key pair (cpk, csk) , where **cpk** and **csk** are compact representations of a UOV public key and its associated secret key, respectively.
- **UOV.ExpandSK**: $\text{csk} \mapsto \text{esk}$. It takes as input **csk**, the compact representation of a UOV secret key, and outputs **esk**, the expanded representation of that secret key. This process is deterministic and *reconstructible*.
- **UOV.ExpandPK**: $\text{cpk} \mapsto \text{epk}$. It takes as input **cpk**, the compact representation of a UOV public key, and outputs **epk**, the expanded representation of that public key. This process is also deterministic.
- **UOV.Sign**: $(\text{esk}, \mu) \mapsto \sigma$. It takes an expanded secret key **esk**, a message $\mu \in \{0, 1\}^*$ to be signed, and outputs a signature σ of μ .

- **UOV.Verify:** $(\text{epk}, \mu, \sigma) \mapsto b \in \{\text{True}, \text{False}\}$. It takes as input an expanded public key epk , a message-signature pair (μ, σ) , and returns $b = \text{True}$ if the message-signature pair is valid and $b = \text{False}$ otherwise.

As specified in the Changelog, we have updated in this submission the generation of the seed_{pk} that corresponds to the bulk of the public key, mostly driven by the design of MAYO [10] as well as [82].

Compact key generation procedure $\text{UOV.CompactKeyGen}(\cdot)$. This *randomized* procedure is to generate (cpk, csk) , the compact representation of a public/secret key pair.

It first samples the seed $\text{seed}_{\text{sk}} \leftarrow \{0, 1\}^{\text{sk_seed_len}}$ for the secret key uniformly at random. Then we derive both the seed seed_{pk} and the matrix $\mathbf{O} \in \mathbb{F}_q^{v \times m}$ by expanding the random seed $\text{seed}_{\text{sk}} \in \{0, 1\}^{\text{sk_seed_len}}$ using the PRNG $\text{Expand}_{\text{sk}}(\cdot)$, where seed_{pk} corresponds to the bulk of the public key, and $\mathbf{O}$ determines the oil space.

Furthermore, to determine the m multivariate quadratic polynomials $p_1, \dots, p_m$ that constitute the public key, we sample the m upper-triangular matrices

$$\mathbf{P}_i = \begin{bmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ \mathbf{0} & \mathbf{P}_i^{(3)} \end{bmatrix} \in \mathbb{F}_q^{n \times n},$$

where $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{v \times v}$, $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{v \times m}$ are both derived by expanding the random seed $\text{seed}_{\text{pk}} \in \{0, 1\}^{\text{pk_seed_len}}$ using the PRNG $\text{Expand}_{\text{P}}(\cdot)$, and

$$\mathbf{P}_i^{(3)} := \text{Upper} \left(-\mathbf{O}^T \mathbf{P}_i^{(1)} \mathbf{O} - \mathbf{O}^T \mathbf{P}_i^{(2)} \right), \quad i \in [m].$$

Finally, this procedure outputs the compact representation of the public key $\text{cpk} = (\text{seed}_{\text{pk}}, \{\mathbf{P}_i^{(3)}\}_{i \in [m]})$, and the compact representation of the secret key $\text{csk} = \text{seed}_{\text{sk}}$.

Secret key expansion procedure $\text{UOV.ExpandSK}(\cdot)$. This procedure simply rederives $(\text{seed}_{\text{pk}}, \mathbf{O})$ from seed_{sk} , as well as $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i \in [m]}$ from seed_{pk} . It also computes a sequence of matrices $\{\mathbf{S}_i\}_{i \in [m]}$, where

$$\mathbf{S}_i = \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)T} \right) \mathbf{O} + \mathbf{P}_i^{(2)}.$$

Finally, it outputs the *expanded* representation of the secret key $\text{esk} = (\text{seed}_{\text{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i \in [m]})$.

Signature generation procedure $\text{UOV.Sign}(\cdot)$. Given the message $\mu \in \{0, 1\}^*$ to be signed, this *randomized* procedure first computes the hash digest $\mathbf{t} = \text{Hash}(\mu \parallel \text{salt})$, where $\text{salt} \leftarrow \{0, 1\}^{\text{salt_len}}$ is sampled uniformly at random. The remaining part of the signing algorithm is devoted to computing a preimage $\mathbf{s} \in \mathbb{F}_q^n$ of $\mathbf{t}$ under the map $\mathcal{P}$ via rejection sampling. Each round begins by deriving a vinegar vector $\mathbf{v} := \text{Expand}_{\text{v}}(\mu \parallel \text{salt} \parallel \text{seed}_{\text{sk}} \parallel \text{ctr}) \in \mathbb{F}_q^v$ from the message μ , the salt, the seed_{sk} for secret key and a one-byte counter ctr that is initially zero and increments after each round. Then we seek to compute a feasible $\mathbf{x} \in \mathbb{F}_q^m$ satisfying $\mathcal{P} \left(\begin{bmatrix} \mathbf{v} \\ \mathbf{0}_m \end{bmatrix} + \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix} \cdot \mathbf{x} \right) = \mathbf{t}$ by solving the system of linear equations $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$, where $\mathbf{L} \in \mathbb{F}_q^{m \times m}$ is a square matrix determined by $\mathbf{v}$ (as well as the secret key), and $\mathbf{y} = \mathcal{P}(\bar{\mathbf{v}})$ is the evaluation of $\mathcal{P}$ at $\bar{\mathbf{v}} = \begin{bmatrix} \mathbf{v} \\ \mathbf{0}_m \end{bmatrix}$. If $\mathbf{L}$ is invertible, we can efficiently find a unique solution $\mathbf{x}$, and hence a valid signature $\sigma = (\mathbf{s}, \text{salt})$ of the incoming message μ ; otherwise, we just increment the counter ctr and jump to the next round.

UOV.CompactKeyGen(params):

```

1: seedsk ← {0, 1}sk_seed_len
2: (seedpk, O) := Expandsk(seedsk)
3: {Pi(1), Pi(2)}i ∈ [m] := ExpandP(seedpk)
4: for i = 1 upto m do
5:   Pi(3) := Upper(−OTPi(1)O − OTPi(2))
6: cpk := (seedpk, {Pi(3)}i ∈ [m])
7: csk := seedsk
8: return (cpk, csk).

```

$\triangleright \mathbf{O} \in \mathbb{F}_q^{v \times m}$
 $\triangleright \mathbf{P}_i^{(1)} \in \mathbb{F}_q^{v \times v}$ upper-triangular
 $\triangleright \mathbf{P}_i^{(2)} \in \mathbb{F}_q^{v \times m}$

UOV.ExpandSK(csk): $\triangleright \text{csk} = \text{seed}_{\text{sk}}$

```

1: (seedpk, O) := Expandsk(seedsk)
2: {Pi(1), Pi(2)}i ∈ [m] := ExpandP(seedpk)
3: for i = 1 upto m do
4:   Si := (Pi(1) + Pi(1)T) O + Pi(2)
5: esk := (seedsk, O, {Pi(1), Si}i ∈ [m])
6: return esk.

```

UOV.ExpandPK(cpk): $\triangleright \text{cpk} = (\text{seed}_{\text{pk}}, \{\mathbf{P}_i^{(3)}\}_{i \in [m]})$

```

1: {Pi(1), Pi(2)}i ∈ [m] := ExpandP(seedpk)
2: for i = 1 upto m do
3:   Pi :=  $\begin{bmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ \mathbf{0} & \mathbf{P}_i^{(3)} \end{bmatrix}$ 
4: epk := {Pi}i ∈ [m]
5: return epk.

```

UOV.Sign(esk, μ): $\triangleright \text{esk} = (\text{seed}_{\text{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i \in [m]})$

```

1: salt ← {0, 1}salt_len
2: t ← Hash(μ || salt)
3: for ctr = 0 upto 255 do
4:   v := Expandv(μ || salt || seedsk || ctr)
5:   L := Om × m
6:   for i = 1 upto m do
7:     Set i-th row of L to vTSi.
8:   if L is invertible then
9:     y ←  $\left[ \mathbf{v}^T \mathbf{P}_i^{(1)} \mathbf{v} \right]_{i \in [m]}$ 
10:    Solve Lx = t − y for x
11:    s :=  $\begin{bmatrix} \mathbf{v} \\ \mathbf{0}_m \end{bmatrix} + \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix} \cdot \mathbf{x}$ 
12:    σ := (s, salt)
13:    return σ
14: return ⊥.

```

$\triangleright \mathbf{t} \in \mathbb{F}_q^m$
 $\triangleright \mathbf{v} \in \mathbb{F}_q^v$
 $\triangleright \mathbf{s} \in \mathbb{F}_q^n$

UOV.Verify(epk, μ, σ = (s, salt)): $\triangleright \text{epk} = \{\mathbf{P}_i\}_{i \in [m]}$

```

1: t ← Hash(μ || salt)
2: return (t ==  $[\mathbf{s}^T \mathbf{P}_i \mathbf{s}]_{i \in [m]}$ ).

```

Figure 2: The compact key generation, public/secret key expansion, signature generation and verification procedures for our UOV variants.

Public key expansion procedure $\text{UOV.ExpandPK}(\cdot)$. This procedure simply rederives $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i \in [m]}$ from seed_{pk} , and outputs $\text{epk} = \{\mathbf{P}_i\}_{i \in [m]}$, where $\mathbf{P}_i = \begin{bmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ \mathbf{0} & \mathbf{P}_i^{(3)} \end{bmatrix}$.

Verification procedure $\text{UOV.Verify}(\cdot)$. Given the expanded public key epk as well as the message-signature pair (μ, σ) where $\sigma = (\mathbf{s}, \text{salt})$, this procedure recomputes the salted hash digest $\mathbf{t} = \text{Hash}(\mu \parallel \text{salt})$, evaluates $\mathcal{P}$ on the input $\mathbf{s} \in \mathbb{F}_q^n$, and accepts the given message-signature pair if and only if $\mathbf{t} = \mathcal{P}(\mathbf{s})$.

3.2.4 Specification of the UOV variants

The key sizes of a digital signature scheme in MPKC are usually much larger than those of other types of post-quantum counterparts. Given the importance of key size in certain practical needs, the public/secret keys of our UOV digital signature scheme can be represented either in the expanded form or in the compressed form, and the associated key expansion operation is applied when necessary. The flexibility in the public/secret key representation enables us to design three, instead of one, variants of UOV, so as to accommodate different practical needs. It should be noted that these three variants are essentially the same, except with different representations of public/secret key pair; in particular, a signature generated with one variant can be verified by the other ones, and they achieve the same concrete hardness when instantiated with the same set of parameters.

In a nutshell, the three variants **classic**, **pkc** and **pkc+skc** of our UOV digital signature scheme are summarized as follows.

- **classic**: in this variant, the public/secret key pair is (epk, esk) , *i.e.*, both the public key expansion procedure and the secret key expansion procedure are carried out in its key generation algorithm $\text{UOV.classic.KeyGen}(\cdot)$. This means the key sizes are larger, but signing and verification are faster.
- **pkc**: in this public-key-compressed variant, the public/secret key pair is (cpk, esk) . Therefore, the secret key expansion procedure is carried out in its key generation algorithm $\text{UOV.pkc.KeyGen}(\cdot)$, and the public key expansion procedure is carried out in its verification algorithm $\text{UOV.pkc.Verify}(\cdot)$. This makes the public key much smaller (by a factor between 6 and 7), but makes verification slower.
- **pkc+skc**: in this doubly-compressed variant, the public/secret key pair is (cpk, csk) . Therefore, the secret key expansion procedure is carried out in its signing algorithm $\text{UOV.pkc+skc.Sign}(\cdot)$, and the public key expansion procedure is carried out in its verification algorithm $\text{UOV.pkc+skc.Verify}(\cdot)$. Compared with the compressed **pkc** variant, its key generation algorithm is faster, and the secret key becomes tiny (only sk_seed_len bits), but its signing algorithm becomes much slower.

The key generation, signing, and verification algorithms of these three variants are straightforward combinations of the foregoing five procedures, and are fully specified in Figure 3. Please refer to Table 3 for their qualitative comparisons.

Table 3: Qualitative comparisons of three UOV variants.

UOV variants	key pair	public key compressed	secret key compressed
classic	(epk, esk)	✗	✗
pkc	(cpk, esk)	✓	✗
pkc+skc	(cpk, csk)	✓	✓

UOV.classic.KeyGen(params):

```

1: seedsk ← {0, 1}sk_seed_len
2: (seedpk, O) := Expandsk(seedsk)
3: {Pi(1), Pi(2)}i∈[m] := ExpandP(seedpk)
4: for i = 1 upto m do
5:   Pi(3) := Upper(−OTPi(1)O − OTPi(2))
6:   Pi :=  $\begin{bmatrix} P_i^{(1)} & P_i^{(2)} \\ \mathbf{0} & P_i^{(3)} \end{bmatrix}$ 
7:   Si := (Pi(1) + Pi(1)T)O + Pi(2)
8: esk := (seedsk, O, {Pi(1), Si}i∈[m])
9: epk := {Pi}i∈[m]
10: return (epk, esk).

```

UOV.classic.Sign(esk, μ):

```

1: return UOV.Sign(esk, μ).

```

UOV.classic.Verify(epk, μ, σ):

```

1: return UOV.Verify(epk, μ, σ).

```

UOV.pkc.KeyGen(params):

```

1: seedsk ← {0, 1}sk_seed_len
2: (seedpk, O) := Expandsk(seedsk)
3: {Pi(1), Pi(2)}i∈[m] := ExpandP(seedpk)
4: for i = 1 upto m do
5:   Pi(3) := Upper(−OTPi(1)O − OTPi(2))
6:   Si := (Pi(1) + Pi(1)T)O + Pi(2)
7: cpk := (seedpk, {Pi(3)}i∈[m])
8: esk := (seedsk, O, {Pi(1), Si}i∈[m])
9: return (cpk, esk).

```

UOV.pkc.Sign(esk, μ):

```

1: return UOV.Sign(esk, μ).

```

UOV.pkc.Verify(cpk, μ, σ):

```

1: epk := UOV.ExpandPK(cpk)
2: return UOV.Verify(epk, μ, σ).

```

UOV.pkc+skc.KeyGen(params):

```

1: (cpk, csk) ← UOV.CompactKeyGen(params)
2: return (cpk, csk).

```

UOV.pkc+skc.Sign(csk, μ):

```

1: esk := UOV.ExpandSK(csk)
2: return UOV.Sign(esk, μ).

```

UOV.pkc+skc.Verify(cpk, μ, σ):

```

1: epk := UOV.ExpandPK(cpk)
2: return UOV.Verify(epk, μ, σ).

```

Figure 3: The key generation, signing, and verification algorithms of the **classic**, **pkc**, and **pkc+skc** variants of the UOV signature scheme.

3.3 Data Layout of Keys and Signature

Choices of finite fields. We use the following two finite fields in our UOV implementations:

- $\mathbb{F}_{256} := \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$;
- $\mathbb{F}_{16} := \mathbb{F}_2[x]/(x^4 + x + 1)$.

And each finite field element is represented by a polynomial over $\mathbb{F}_2$.

Each element in $\mathbb{F}_{256}$ is stored in one byte as its coefficient array with the most significant bit corresponding to x^7 . A vector in $\mathbb{F}_{256}^\ell$ is represented as an ℓ -byte string, the first entry of the vector corresponding to the first byte of the string.

For $\mathbb{F}_{16}$, we pack two field elements into one byte with the first element in the least significant nibble. The most significant bit of each nibble corresponds to x^3 . Each vector in $\mathbb{F}_{16}^\ell$ is encoded as its bit-packed representation in column-major order, and hence is represented as an $\ell/2$ -byte string (we only ever need to encode vectors of even length).

Encodings of matrices. For the matrices $\{\mathbf{P}_i^{(1)}\}_{i \in [m]}$, let $\mathbf{P}_i^{(1)} = [a_i^{(j,k)}]_{(j,k) \in [v] \times [v]} \in \mathbb{F}_q^{v \times v}$ for the moment. In our implementations, the bit string for $\{\mathbf{P}_i^{(1)}\}_{i \in [m]}$ is the bit-packed representation of the Macaulay matrix $\mathbf{M}_1$ of $\{\mathbf{P}_i^{(1)}\}_{i \in [m]}$ in column-major order, where

$$\mathbf{M}_1 = \begin{bmatrix} a_1^{(1,1)} & a_1^{(1,2)} & \cdots & a_1^{(1,v)} & a_1^{(2,2)} & a_1^{(2,3)} & \cdots & a_1^{(2,v)} & \cdots & a_1^{(v,v)} \\ a_2^{(1,1)} & a_2^{(1,2)} & \cdots & a_2^{(1,v)} & a_2^{(2,2)} & a_2^{(2,3)} & \cdots & a_2^{(2,v)} & \cdots & a_2^{(v,v)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_m^{(1,1)} & a_m^{(1,2)} & \cdots & a_m^{(1,v)} & a_m^{(2,2)} & a_m^{(2,3)} & \cdots & a_m^{(2,v)} & \cdots & a_m^{(v,v)} \end{bmatrix} \in \mathbb{F}_q^{m \times (v(v+1)/2)}.$$

Similarly, for the matrices $\{\mathbf{P}_i^{(3)}\}_{i \in [m]}$, let $\mathbf{P}_i^{(3)} = [c_i^{(j,k)}]_{(j,k) \in [m] \times [m]} \in \mathbb{F}_q^{m \times m}$ for the moment. In our implementations, the bit string for $\{\mathbf{P}_i^{(3)}\}_{i \in [m]}$ is the bit-packed representation of the Macaulay matrix $\mathbf{M}_3$ of $\{\mathbf{P}_i^{(3)}\}_{i \in [m]}$ in column-major order, where

$$\mathbf{M}_3 = \begin{bmatrix} c_1^{(1,1)} & c_1^{(1,2)} & \cdots & c_1^{(1,m)} & c_1^{(2,2)} & c_1^{(2,3)} & \cdots & c_1^{(2,m)} & \cdots & c_1^{(m,m)} \\ c_2^{(1,1)} & c_2^{(1,2)} & \cdots & c_2^{(1,m)} & c_2^{(2,2)} & c_2^{(2,3)} & \cdots & c_2^{(2,m)} & \cdots & c_2^{(m,m)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ c_m^{(1,1)} & c_m^{(1,2)} & \cdots & c_m^{(1,m)} & c_m^{(2,2)} & c_m^{(2,3)} & \cdots & c_m^{(2,m)} & \cdots & c_m^{(m,m)} \end{bmatrix} \in \mathbb{F}_q^{m \times (m(m+1)/2)}.$$

For the matrices $\{\mathbf{P}_i^{(2)}\}_{i \in [m]}$, let $\mathbf{P}_i^{(2)} = [b_i^{(j,k)}]_{(j,k) \in [v] \times [m]} \in \mathbb{F}_q^{v \times m}$ for the moment. In our implementations, the bit string for $\{\mathbf{P}_i^{(2)}\}_{i \in [m]}$ is the bit-packed representation of the matrix $\mathbf{M}_2$ in column-major order, where

$$\mathbf{M}_2 = \begin{bmatrix} b_1^{(1,1)} & \cdots & b_1^{(1,m)} & b_1^{(2,1)} & \cdots & b_1^{(2,m)} & \cdots & b_1^{(v,1)} & \cdots & b_1^{(v,m)} \\ b_2^{(1,1)} & \cdots & b_2^{(1,m)} & b_2^{(2,1)} & \cdots & b_2^{(2,m)} & \cdots & b_2^{(v,1)} & \cdots & b_2^{(v,m)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ b_m^{(1,1)} & \cdots & b_m^{(1,m)} & b_m^{(2,1)} & \cdots & b_m^{(2,m)} & \cdots & b_m^{(v,1)} & \cdots & b_m^{(v,m)} \end{bmatrix} \in \mathbb{F}_q^{m \times (vm)}.$$

Similarly, for the matrices $\{\mathbf{S}_i\}_{i \in [m]}$, let $\mathbf{S}_i = [d_i^{(j,k)}]_{(j,k) \in [v] \times [m]} \in \mathbb{F}_q^{v \times m}$ for the moment. In our implementations, the bit string for $\{\mathbf{S}_i\}_{i \in [m]}$ is the bit-packed representation of the matrix $\mathbf{M}_S$ in column-major order, where

$$\mathbf{M}_S = \begin{bmatrix} d_1^{(1,1)} & \cdots & d_1^{(1,m)} & d_1^{(2,1)} & \cdots & d_1^{(2,m)} & \cdots & d_1^{(v,1)} & \cdots & d_1^{(v,m)} \\ d_2^{(1,1)} & \cdots & d_2^{(1,m)} & d_2^{(2,1)} & \cdots & d_2^{(2,m)} & \cdots & d_2^{(v,1)} & \cdots & d_2^{(v,m)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ d_m^{(1,1)} & \cdots & d_m^{(1,m)} & d_m^{(2,1)} & \cdots & d_m^{(2,m)} & \cdots & d_m^{(v,1)} & \cdots & d_m^{(v,m)} \end{bmatrix} \in \mathbb{F}_q^{m \times (vm)}.$$

Finally comes the matrix $\mathbf{O} \in \mathbb{F}_q^{v \times m}$. In our implementations, the bit string for $\mathbf{O}$ is the bit-packed representation of the matrix $\mathbf{O}$ itself in column-major order.

Encoding of epk. In the *expanded* public key $\text{epk} = \{\mathbf{P}_i\}_{i \in [m]}$, each upper-triangular matrix $\mathbf{P}_i \in \mathbb{F}_q^{n \times n}$ corresponds to a homogeneous quadratic polynomial. Based on n, m as well as $v = n - m$, each matrix $\mathbf{P}_i$ is further divided into three blocks $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{v \times v}$, $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{v \times m}$, and $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{m \times m}$, such that

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

In our implementations, the bit string for epk is the concatenation of the bit-packed representation of $\{\mathbf{P}_i^{(1)}\}_{i \in [m]}$, the bit-packed representation of $\{\mathbf{P}_i^{(2)}\}_{i \in [m]}$, and finally the bit-packed representation of $\{\mathbf{P}_i^{(3)}\}_{i \in [m]}$. Concretely, the epk consists of $mn(n+1)/2$ field elements in $\mathbb{F}_q$: the first $mv(v+1)/2$ elements encode the matrices $\{\mathbf{P}_i^{(1)}\}_{i \in [m]}$, the next m^2v ones encode the matrices $\{\mathbf{P}_i^{(2)}\}_{i \in [m]}$, and the remaining $m^2(m+1)/2$ ones encode the matrices $\{\mathbf{P}_i^{(3)}\}_{i \in [m]}$. Thus, the size of the expanded public key is

$$|\text{epk}| = \frac{1}{16} \cdot mn(n+1) \cdot \log q \quad \text{bytes.}$$

Encoding of esk. In the *expanded* secret key esk , we have

$$\text{esk} = \left(\text{seed}_{\text{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i \in [m]} \right).$$

The bit string for esk is the concatenation of the binary representation of seed_{sk} , the bit-packed representation of $\mathbf{O}$, the bit-packed representation of $\{\mathbf{P}_i^{(1)}\}_{i \in [m]}$, and finally the bit-packed representation of $\{\mathbf{S}_i\}_{i \in [m]}$.

The number of bytes required to store the expanded secret key is therefore

$$|\text{esk}| = \frac{1}{8} \cdot \left[\underbrace{\text{sk_seed_len}}_{\text{seed}_{\text{sk}}} + \left(\underbrace{mv}_{\mathbf{O}} + \underbrace{mv(v+1)/2}_{\{\mathbf{P}_i^{(1)}\}_{i \in [m]}} + \underbrace{m^2v}_{\{\mathbf{S}_i\}_{i \in [m]}} \right) \log q \right].$$

Encodings of cpk and csk. In the *compressed* public/secret key pair (cpk, csk) , we have

$$\text{cpk} = \left(\text{seed}_{\text{pk}}, \{\mathbf{P}_i^{(3)}\}_{i \in [m]} \right), \quad \text{and} \quad \text{csk} = \text{seed}_{\text{sk}}.$$

The bit string cpk is the concatenation of the binary representation of seed_{pk} and the bit-packed representation of $\{\mathbf{P}_i^{(3)}\}_{i \in [m]}$. Conversely, the bit string for csk is trivially the binary representation of seed_{sk} .

Hence, the respective number of bytes to store cpk and csk are

$$|\text{cpk}| = \frac{1}{8} \cdot \left(\text{pk_seed_len} + \frac{1}{2} \cdot m^2(m+1) \cdot \log q \right), \quad \text{and} \quad |\text{csk}| = \frac{1}{8} \cdot \text{sk_seed_len}.$$

Encoding of the signature. A UOV signature $\sigma = (\mathbf{s}, \text{salt})$ consists of a vector $\mathbf{s} \in \mathbb{F}_q^n$ and a binary string $\text{salt} \in \{0, 1\}^{\text{salt_len}}$. The bit string for σ is the bit-packed representation of the vector $\mathbf{s}$ in column-major order and the binary representation of salt . Therefore, the number of bytes needed to store a signature is

$$|\sigma| = \left(\left\lceil \frac{n}{8} \cdot \log q \right\rceil + \frac{1}{8} \cdot \text{salt_len} \right).$$

3.4 Recommended Parameter Sets

To accommodate different security needs, we propose four recommended parameter sets for UOV. These four recommended parameter sets, together with their corresponding key/signature sizes, are presented in Table 4.

Table 4: Recommended parameter sets and the corresponding key/signature sizes for UOV variants. Note that in each parameter set, we have `salt_len` = 128, `pk_seed_len` = 128, and `sk_seed_len` = 256.

	NIST S.L.	n	m	q	$ \text{epk} $ (bytes)	$ \text{esk} $ (bytes)	$ \text{cpk} $ (bytes)	$ \text{csk} $ (bytes)	$ \sigma $ (bytes)
<code>uov-1p</code>	1	119	45	256	321 300	278 087	46 591	32	135
<code>uov-1s</code>	1	160	64	16	412 160	348 704	66 576	32	96
<code>uov-III</code>	3	193	72	256	1 347 912	1 167 440	189 232	32	209
<code>uov-V</code>	5	259	96	256	3 232 320	2 801 024	446 992	32	275

First, it should be stressed that each recommended parameter set in Table 4 is applicable to *every* UOV variant presented in Section 3.2.4. Moreover, among all four recommended parameter sets, we always have `salt_len` = `pk_seed_len` = 128 and `sk_seed_len` = 256. Finally, as shown in Table 4, their main differences lie in the choice of (n, m, q) :

- For NIST security level 1, we propose two recommended parameter sets: `uov-1p`, which is instantiated over $\mathbb{F}_{256}$ and gets slightly smaller keys, and `uov-1s`, which is instantiated over $\mathbb{F}_{16}$ and has shorter signatures;
- For NIST security level 3, we propose one recommended parameter set, *i.e.*, `uov-III`;
- Finally, we propose one recommended parameter set, *i.e.*, `uov-V`, for NIST security level 5.

In sum, given the three UOV variants presented in Figure 3 and the four recommended parameter sets presented in Table 4, this submission consists of $12 = 3 \times 4$ UOV instances, and they are *labeled* by concatenating the name of the variant with that of the recommended parameters set. For instance, `uov-1s-classic` refers to the UOV instance when we instantiate the `classic` UOV variant with the `uov-1s` parameter set, while `uov-V-pkc+skc` refers to the UOV instance when the `pkc+skc` UOV variant is instantiated with the `uov-V` parameter set.

Section 4 presents the concrete security analysis of UOV with these four parameter sets, since this analysis is independent of the choice of UOV variants; and Section 5 is devoted to the implementations of three UOV variants in combination with these four recommended parameter sets over various platforms.

4 Concrete Security Analysis

This section conducts the concrete security analysis of UOV. Similar to most of the cryptosystems in MPKC, researchers have not presented a formal security proof which reduces certain well-known “hard” mathematical problem(s) say, the MQ problem, to the security of UOV. The concrete security analysis for UOV is usually carried out by listing all the known critical attacks against UOV that may influence its concrete hardness estimate result. Our confidence in the security of UOV lies in the facts that UOV remains secure after more than two decades of cryptanalysis, and that there is a solid theoretical foundation on the concrete hardness estimate of practical attacks against MPKC such that the theoretical hardness estimate of UOV matches the experimental results consistently.

Historically, those attacks against UOV are usually classified into two types: (1) the *key-recovery attacks* that aim to recover the secret key from the given public key, *e.g.*, the Kipnis-Shamir attack [58], the intersection attack [14], the MinRank attack [7, 79, 87] and the wedge attack [80, 54, 44]; (2) the *forgery attacks* that aim to forge a message-signature pair passing the verification test, *e.g.*, the collision attacks against the hash function, and the direct attack. It should be noted that in the forgery attack, the hash function $\text{Hash}(\cdot)$ is usually modeled as a random oracle (RO) [23].

EUFCMA security of UOV. UOV can be designed to achieve the EUFCMA security requirement, *e.g.*, the salt-UOV proposed in 2011 [84]. As can be seen from Appendix A, the EUFCMA security of salt-UOV is readily based on the hardness of the UOV problem, an intermediate problem in the MQ realm.

Compared with salt-UOV, we prefer our recommended UOV scheme depicted in Section 3 for the following reasons:

- All the state-of-the-art attacks against our recommended UOV are applicable to salt-UOV, and vice versa. This means that when instantiated with the same set of parameters, they achieve the same security level according to the state-of-the-art cryptanalysis in MPKC.
- The signing algorithm of our recommended UOV runs faster than that of salt-UOV.

Bit-cost estimates. In this section we consider the best known (classical) attacks that are relevant to four recommended UOV parameter sets proposed in Section 3.4 under the usual RAM model. Moreover, for those attacks based on parallelized Wiedemann-based XL (PWXL) algorithm, their additional memory-aware costs under the MEM-2D model [35] presented in Section 4.7 are also considered. The bit-costs of the best known attacks against recommended UOV parameter sets under the standard RAM model are summarized in Table 5. we also report in Table 5 the *additional* memory-aware bit-costs of PWXL-related attacks under the MEM-2D model, and this provides additional evidence regarding the practical cost of the relevant attacks.

4.1 Collision Attack

The first attack we consider is a simple collision attack on the equality $\mathcal{P}(\mathbf{s}) = \text{Hash}(\mu \parallel \text{salt})$. An attacker can compute $\mathcal{P}(\mathbf{s}_i)$ for X inputs $\{\mathbf{s}_i\}_{i \in [X]}$ and compute $\text{Hash}(\mu \parallel \text{salt}_j)$ for Y salts $\{\text{salt}_j\}_{j \in [Y]}$. If $X \cdot Y = \alpha q^m$, then there is a collision $\mathcal{P}(\mathbf{s}_{i^*}) = \text{Hash}(\mu \parallel \text{salt}_{j^*})$ with probability $\approx 1 - e^{-\alpha}$, and the attacker can output the signature $(\mathbf{s}_{i^*}, \text{salt}_{j^*})$ for the message μ .

Computing hashes. For the concrete cost estimate, we assume that a Keccak-f 1600 permutation costs $2^{17.5}$ bit operations [86], so computing the list of hashes takes at least $Y \cdot 2^{17.5}$ bit operations.

Table 5: Bit-cost estimates (lower bound for the base-2 logarithm of the number of binary gates required to perform an attack) of state-of-the-art attacks against recommended UOV parameter sets. Note that for PWXL-related attacks (direct, intersection, wedge), both their bit-costs, *i.e.*, $C'_{\text{RAM}} = \log_2 C_{\text{RAM}}$ under the standard RAM model and their *additional* bit-costs δ_{2D} under the MEM-2D model are presented.

parameter set	uov-Ip (119, 45, 256)		uov-Is (160, 64, 16)		uov-III (193, 72, 256)		uov-V (259, 96, 256)	
costs	C'_{RAM}	δ_{2D}	C'_{RAM}	δ_{2D}	C'_{RAM}	δ_{2D}	C'_{RAM}	δ_{2D}
collision	194	–	143	–	303	–	399	–
direct	146	7	162	8	216	21	277	34
KS	258	–	153	–	420	–	565	–
intersection	201	31	176	24	314	60	413	85
wedge	144	17	158	22	213	34	272	48

Cost estimate. The lowest conceivable bit-cost of the collision attack (in the standard RAM model) is identified as

$$\frac{1}{(1 - e^{-\alpha})} (\varepsilon_{r,m} X + 2^{17.5} Y) ,$$

where $\varepsilon_{r,m}$ is the cost of evaluation. This is approximately equal to $2^{10.1} \sqrt{q^m \varepsilon_{r,m}}$ for optimally chosen X, Y and α^1 . We argue that this is at least 2^{143} because

- Suppose we assume the best known evaluation for MQ with m equations over $\mathbb{F}_{2^r}$: Gray-code enumeration [9]. We can evaluate a multivariate quadratic polynomial on a large number of inputs using only $3r$ bit operations per evaluation ($2r$ bit operations to compute the evaluation, and r bit operations to copy the evaluation to a list). We can optimize the attack by, evaluating only the first $m' = m/2 + o(m)$ polynomials of $\mathcal{P}$ (as opposed to all m of them) to look for partial collisions (*i.e.*, s_i and salt_j such that the first m' elements of $\mathcal{P}(s_i)$ matches the first $m' \log q$ bits of $\text{Hash}(\mu \parallel \text{salt}_j)$). Each time a partial collision is found, we use naive polynomial evaluation to check if it is a complete collision. For appropriately chosen m' this second step is cheap because the number of partial collisions is small, therefore we optimistically also ignore this step in our cost analysis. Then the cost is $2^{10.7} \sqrt{q^m m r}$, which is the formula we use in Table 5.

Note that there the uov-Is entry should compute to 142.7, but this fails to take into account that this method uses a lot of memory. This incurs a cost², which we also did not multiply into Table 5 because there is some dispute as to how much the large memory penalty is, and because we are ignoring possible tricks [81] resulting in a $\sqrt{q} \times$ speedup. Nevertheless, regardless of the precise value assigned to the large-memory penalty, the total cost will be at least 2^{143} .

- Realistically, an attacker would use a memoryless collision-finding algorithm such as *e.g.*, in [70]. However, algorithms like [70] have a small overhead in the number of function evaluations, and it would not be possible to take full advantage of Gray-code enumeration optimization (if you use Gray code to evaluate 2^k times, you typically lose about a factor of $2^{k/2}$).

¹We want to minimize $\sqrt{\alpha}/(1 - e^{-\alpha})$, which happens at $\alpha = -W_{-1}(-e^{-1/2}/2) - \frac{1}{2}$ or around $\alpha = 1.25$.

²Bernstein *et al.* [68]: “we estimate the cost of each access to a bit within N bits of memory as the cost of $\sqrt{N}/2^5$ ‘bit operations.’”

4.2 Direct Attack

The most straightforward attack against UOV, and even against most of the MPKC cryptosystems, is the direct attack that aims to solve an instance of the MQ problem associated with the public key $\mathcal{P}$. In the direct attack, the attacker first chooses a message $\mu^* \in \{0, 1\}^*$ and a salt $\text{salt}^* \in \{0, 1\}^{\text{salt_len}}$ adaptively, computes $\mathbf{t} = \text{Hash}(\mu^* \parallel \text{salt}^*)$, and then is devoted to the recovery of a preimage $\mathbf{s}$ for $\mathbf{t}$ under the public key $\mathcal{P}$ via the system-solving techniques.

At the heart of the attack lies solving a random system of m quadratic equations in n variables; the hybrid WiedemannXL algorithm [93] is used to solve this system. The estimated cost is

$$\text{MQ}(q; n, m) := \min_{k \in [0, n]} \left\{ q^k \cdot 3 \binom{n-k+2}{2} \binom{d_{n-k,m} + n-k}{d_{n-k,m}}^2 (2 \cdot \log^2 q + \log q) \right\},$$

where $d_{N,M}$ is the *solving degree* of XL, and is defined as the smallest $d > 0$ such that the coefficient of t^d in the power series expansion of

$$\frac{(1-t^2)^M}{(1-t)^{N+1}}$$

is non-positive.

Cost estimate. On the one hand, one can follow the idea of Thomae and Wolf [88] and exploit the underdetermined nature of the UOV system by first reducing it to a system of $m' = m - 1$ equations in $n' = m - 1$ variables, and then solving the new system using the hybrid WiedemannXL algorithm [93]. The estimated cost of this approach is $\text{MQ}(q; n', m')$. On the other hand, Hashimoto [51] proposed an improved approach for underdetermined MQ systems parameterized by a and c satisfying

$$n \geq \max\{(a+1)(m-c-a+1), a(m-c) - (a-1)^2 + c\},$$

whose cost is expressed in terms of smaller MQ instances as

$$\min_{a,c} \{(m-a-c+1) \cdot \text{MQ}(q; a, a) + q^c \cdot (\text{MQ}(q; a-1, a-1) + \text{MQ}(q; m-a-c, m-a))\}.$$

The cost of the direct attack (under the standard RAM model) is *identified* as the minimum of the two estimates above.

Note that the attacker might compute $\text{Hash}(\mu \parallel \text{salt})$ for a large number of message-salt pairs, and then solve a multi-target version of the system-solving problem. Nevertheless, our foregoing estimate is justified by the fact that there are no known algorithms that can take advantage of multiple targets (beyond the naive collision attacks introduced in Section 4.1).

4.3 Kipnis-Shamir Attack

The Kipnis-Shamir attack [29, 58] tries to recover the subspace O from the public map $\mathcal{P} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$. Historically, this attack was first proposed for the case $n = 2m$, where it runs in polynomial time and demonstrates the insecurity of the *original balanced* OV scheme proposed in [72]. Moreover, it can be generalized to the cases $n > 2m$, and in the literature its cost was identified as $\mathcal{O}(q^{n-2m}n^4)$, if n is even or q is odd.

However, it turns out that the foregoing formula overestimates the cost of the attack, as the following analysis indicates. First, the cost of finding a single vector in O is dominated by the cost of computing an average of q^{n-2m} characteristic polynomials of n -by- n matrices,

and solving the same number of linear systems in n variables; this takes $\mathcal{O}(q^{n-2m}n^\omega \log(n))$ field multiplications, where ω denotes the exponent of matrix multiplication. The n^4 factor in the literature was obtained by putting $\omega = 3$. Moreover, the foregoing attack should be repeated $m = \mathcal{O}(n)$ times so as to get a basis for O . Nevertheless, this does not contribute an m factor into the overall cost intuitively, because once the first vector in O is found, it could be fully utilized and the other vectors in O can be found more efficiently with other methods (*e.g.*, see [14]).

Cost estimate. We therefore identify the cost of the Kipnis–Shamir attack (under the RAM model) as

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

which is regarded as a conservative estimate of the attack cost.

4.4 Intersection Attack

The intersection attack [14] generalizes the ideas behind the Kipnis–Shamir attack, in combination with a system-solving approach such as in the reconciliation attack [39]. It tries to simultaneously find k linearly independent vectors in $O = \{\mathbf{u} \in \mathbb{F}_q^n \mid \mathcal{P}(\mathbf{u}) = \mathbf{0}_m\}$, by solving a system of quadratic equations for some vector(s) in the intersection $\cap_{i=1}^k \mathbf{M}_i O$, for some matrices $\mathbf{M}_i$. The attack only works if the intersection is nonempty, which is guaranteed if $n < \frac{2k-1}{k-1}m$. This attack is described in detail in [14].

The cost of the attack (in the standard RAM model) is dominated by the cost of solving a random system of $M = \binom{k+1}{2}m - 2\binom{k}{2}$ equations in $N = kn - (2k-1)m$ variables. For the uov-Is parameter set we use $k = 3$, even though $n = \frac{2k-1}{k-1}m$. This means that the intersection is not guaranteed to be nontrivial, and the attack is likely to fail. However, one can check that for these parameters the intersection is non-trivial with probability $1/(q-1)$, so on average we only need to repeat the attack $q-1 = 15$ times, which is still cheaper than running a single attack with $k = 2$.

4.5 MinRank Attack

In the MinRank attack, the attacker tries to find a linear combination of the matrices of minimal rank [15, 17, 48, 87]. And the MinRank problem can be formulated as: given the $k+1$ matrices $\mathbf{M}_0, \mathbf{M}_1, \dots, \mathbf{M}_k \in \mathbb{F}_q^{m' \times n'}$ as well as $r \in \{1, \dots, \min\{m', n'\}\}$, find $[c_i]_{i \in [k]}$ such that the rank of their linear combination $\mathbf{M}_0 + \sum_i c_i \mathbf{M}_i$ is not larger than r . Historically, many different approaches have been proposed to solve the MinRank problem, including the Kipnis–Shamir modeling method [59], the minors modeling method [43], and the support minors modeling method [7].

The MinRank attack can be applied to estimate the concrete security of UOV: if $\mathbf{O}_1 \in \mathbb{F}_q^{n \times k}$ is a submatrix of $\overline{\mathbf{O}} = \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix} \in \mathbb{F}_q^{n \times m}$ that consists of $k = \lceil \frac{n-m+1}{m} \rceil$ columns,

then the matrix $\begin{bmatrix} \mathbf{O}_1^\top \mathbf{P}_1 \\ \vdots \\ \mathbf{O}_1^\top \mathbf{P}_m \end{bmatrix} \in \mathbb{F}_q^{km \times n}$ has rank no more than $n-m$. Therefore, the MinRank

attack can be used to recover the secret key of UOV. However, in regard to our four recommended parameter sets depicted in Section 3.4, the cost estimate of MinRank attack is much larger than those of the other types of attack.

4.6 Wedge Attack

In 2025, Lars Ran [80] first proposed the wedge attack that is applicable to UOV as well as its variants, provided that the underlying finite field is of characteristic $p = 2$. The

attack follows from the observation that when $p = 2$, each polar form of UOV can be naturally represented as a 2-vector in $\wedge \mathbb{F}_q^n$, and the orthogonal complement of the oil space can be represented, via the Plücker embedding, as a decomposable $(n - m)$ -vector. Jin *et al.* [54] generalized the wedge attack so that it is applicable to UOV instance with arbitrary characteristic. More recently, Furue and Ikematsu [44] further extended this attack by combining the intersection attack with computations in p^ℓ -truncated polynomial rings, leading to *improved* concrete hardness estimates for UOV and its variants.

Recall that we have $\mathcal{P} = \mathcal{F} \circ \mathcal{T}$ in UOV. Here, entries of $\mathcal{P} = (p_1, \dots, p_m)$ and $\mathcal{F} = (f_1, \dots, f_m)$ could be seen as quadratic polynomials in $\mathbb{F}_q[x_1, \dots, x_n]$; $\mathcal{T}$ is an invertible linear transformation in $\mathbb{F}_q^n$ represented by an invertible matrix $\mathbf{T} \in \text{GL}_n(\mathbb{F}_q)$ and can naturally induce a ring automorphism on $\mathbb{F}_q[x_1, \dots, x_n]$. Let $t_j \in \mathbb{F}_q[x_1, \dots, x_n]$ denote the linear polynomial represented by the j -th row of the matrix $\mathbf{T}$. It follows from the fact $(f_1, \dots, f_m) \subseteq (x_1, \dots, x_{n-m})$ that

$$\tilde{\mathcal{T}}(f_1, \dots, f_m) \subseteq \tilde{\mathcal{T}}(x_1, \dots, x_{n-m}),$$

or equivalently,

$$(p_1, \dots, p_m) \subseteq (t_1, \dots, t_{n-m}).$$

Since this relation still holds when passing to p^ℓ -truncated ring $\mathbb{F}_q[x_1, \dots, x_n]/(x_1^{p^\ell}, \dots, x_n^{p^\ell})$, the degree- d components of quotient spaces always satisfy the following inequality:

$$\dim \left(\frac{\mathbb{F}_q[x_1, \dots, x_n]}{(x_1^{p^\ell}, \dots, x_n^{p^\ell}, t_1, \dots, t_{n-m})} \right)_d \leq \dim \left(\frac{\mathbb{F}_q[x_1, \dots, x_n]}{(x_1^{p^\ell}, \dots, x_n^{p^\ell}, p_1, \dots, p_m)} \right)_d.$$

The key observation is that when the foregoing equality holds for some degree d , then the hidden linear forms $t_1, \dots, t_{n-m}$ can be recovered from the public polynomials $p_1, \dots, p_m$. Since these linear forms determine the oil space, recovering them allows one to solve the UOV central map.

Furthermore, Furue and Ikematsu [44] generalized this framework using the intersection attack. The main idea is to convert a standard UOV instance into another instance with a smaller oil space and more polynomials. Such a conversion makes the truncated-ring attack more effective.

Cost estimate. Let $r = \log_2 q$ and $e = \log_p q$. Then the cost of wedge attack against UOV (in the standard RAM model) is identified as

$$\min_{\substack{\ell \in [1, e] \\ k \in [0, m-1]}} 3 \binom{n-k+1}{2} (T(n-k, p^\ell, D) - T(m-k, p^\ell, D) + 1)^2 (2r^2 + r).$$

Here,

$$T(n, p^\ell, d) := \sum_{i=0}^{\min\{n, \lfloor d/p^\ell \rfloor\}} (-1)^i \binom{n}{i} \binom{d - ip^\ell + n - 1}{n-1},$$

and the solving degree D is chosen as the smallest integer $d \in [2, (m-k)(p^\ell - 1)]$ such that

$$\left[\frac{(1+z+\dots+z^{p^\ell-1})^{n-k}}{(1+z^2+\dots+z^{2(p^\ell-1)})^m} \right]_d \leq T(m-k, p^\ell, d),$$

where the notation $[g(z)]_d$ denotes the coefficient of the d -th power in $g(z)$.

When the intersection attack is applied in the truncated-ring setting, as is suggested in [44], the concrete hardness result can be improved further: for the case $n < 3m$, the *improved* estimate for cost of the wedge attack against UOV is

$$\min_{\substack{\ell \in [1, e] \\ k \in [0, 3m-n-1]}} 3 \binom{n-k+1}{2} (T(n-k, p^\ell, D) - T(3m-n-k, p^\ell, D) + 1)^2 (2r^2 + r),$$

where the solving degree D is chosen to be the smallest integer $d \in [2, (3m - n - k)(p^\ell - 1)]$ such that

$$\left\lceil \frac{(1 + z + \dots + z^{p^\ell - 1})^{n-k}}{(1 + z^2 + \dots + z^{2(p^\ell - 1)})^{3m-2}} \right\rceil_d \leq T(3m - n - k, p^\ell, d);$$

conversely, for the case $n \geq 3m$, the *improved* concrete hardness estimate of the wedge attack against UOV is

$$\min_{\ell \in [1, e]} 3q^{n-3m+1} \binom{n+1}{2} T(n, p^\ell, D)^2 (2r^2 + r),$$

where D is chosen to be the smallest integer $d \in [2, p^\ell - 1]$ such that

$$\left\lceil \frac{(1 + z + \dots + z^{p^\ell - 1})^n}{(1 + z^2 + \dots + z^{2(p^\ell - 1)})^{3m-2}} \right\rceil_d \leq T(1, p^\ell, d) = 1.$$

Remarks on wedge attack. Compared with other classical attacks against UOV, the strength of the wedge attack can be illustrated from the fact that for the four recommended UOV parameter sets, three of them attain their minimal bit-cost estimates (in the standard RAM model) when wedge attack is employed, as can be seen from Table 5. Nevertheless, for all variants of wedge attack [80, 54, 44], the correctness analysis of their concrete hardness estimates relies on certain heuristic conjectures, and counterexamples have been found to demonstrate that these conjectures do not always hold [80, 37]. Further investigation is needed to establish confidence in the correctness of these estimates.

4.7 Memory-Aware Costs of PWXL-related Attacks

In general, the cost of a cryptanalytic attack is primarily determined by its running time, which is commonly estimated in the standard RAM model. However, for some attacks against cryptographic primitives [68, 30], the actual running time of key subprocedures can be substantially affected by memory-access operations, resulting in additional computational overhead that is not adequately captured by the RAM model. A more practical cost model is therefore useful for obtaining more realistic estimates of the running time of such attacks. In particular, several attacks against UOV, *i.e.*, direct FXL attack, intersection attack and wedge attack, rely heavily on the parallelized Wiedemann-based XL (PWXL) algorithm [35]. The PWXL algorithm employs the Wiedemann algorithm to solve the large sparse linear systems arising in the XL procedure, and its practical running time can be strongly influenced by memory-access costs.

In this subsection, taking into account the specific characteristics of PWXL and drawing on existing memory-aware cost models, we present the MEM-2D model [35] to capture the additional cost associated with memory access. We use the MEM-2D model together with the conventional RAM-model cost of PWXL to obtain more realistic cost estimates for the direct FXL, intersection, and wedge attacks. The resulting additional MEM-2D costs, reported in Table 5, provide an additional sensitivity analysis supporting the practical cost assessment of the recommended UOV parameter sets.

We use the usual gate-count model (RAM) as the primary metric and report a two-dimensional nearest-neighbor, *i.e.*, MEM-2D, model only as a sensitivity analysis. In MEM-2D, a random access to an M -bit memory costs $C_2 \sqrt{M}$, where $C_2 = 2^{-5}$ as suggested in NTRU Prime [68]. The convention is motivated by the area-time literature and prior cryptanalytic cost studies [20, 90, 12, 30]. The same distinction between gate counts and memory-access costs is made in the Kyber specification [61] and in the NIST assessment [1]. Thus MEM-2D is model-dependent, not a replacement for RAM or a lower bound [21].

Costs of PWXL-related attacks. The direct FXL attack, intersection attack and wedge attack all use a regular quadratic-system PWXL stage [26, 14, 80, 44]. And their cost estimates are determined by that of PWXL. Let N be the sparse Wiedemann matrix dimension, w its row weight, E the outer enumeration factor, and $r = \log_2 q$. With $C_{\text{arith}} = 2r^2 + r$, the Wiedemann/Block-Wiedemann count is about $3N$ products [89, 34], and the cost of PWXL under the standard RAM model and its additional cost under the MEM-2D model are

$$C_{\text{RAM}} = E 3wN^2 C_{\text{arith}} \quad \text{and} \quad \Delta C_{2D} = E 3wN^2 \frac{C_{\text{gen}} + \bar{C}_{\text{acc}}}{n'_{\text{blk}}},$$

respectively, where the tall-and-thin row-survival and block-reuse conventions follow the parallel-XL treatment [26, 65]. For support buckets of sizes V_S , retaining $R_{\text{keep}} = \min(N, R)$ of R candidate rows, our attacker-favorable access estimate is

$$\bar{C}_{\text{acc}} = \frac{C_2(r+L)}{N} \sum_S V_S^{3/2}, \quad L = \lceil \log_2(2Nr) \rceil, \quad n'_{\text{blk}} = \min\left(s, \frac{sR_{\text{keep}}}{R}\right).$$

The corresponding logarithmic overhead is thus

$$\delta_{2D} = \log_2 \left(1 + \frac{\Delta C_{2D}}{C_{\text{RAM}}} \right) = \log_2 \left(1 + \frac{C_{\text{gen}} + \bar{C}_{\text{acc}}}{n'_{\text{blk}} C_{\text{arith}}} \right).$$

The $V_S^{3/2}$ term is the access-weighted partition moment used in memory-aware XL models [8]. Here C_{gen} is on-demand column generation and s is the shared-support block height. The $3/2$ moment charges each bucket by its square-root size; output writes and row selection are free. The numerical scan enumerates the radius-two buckets and minimizes direct FXL, wedge, and intersection costs separately under both models, with the usual semi-regular Hilbert-series test [16].

Beyond PWXL. First, for the collision attack discussed in Section 4.1, PWXL is not invoked and hence no such MEM-2D increment is assigned. Similarly, the Kipnis-Shamir attack specified in Section 4.3 aims to enumerate oil subspaces and use dense linear algebra [58], which is beyond the regular PWXL schedule as well. Finally, the Support-Minors MinRank attack discussed in Section 4.5 is bilinear and uses XL in bi-degree $(b, 1)$ [7, 8]; its radius-one star support differs from regular XL, and no compatible access schedule is specified. We therefore do not apply the regular-PWXL MEM-2D estimate to these attacks; instead, we retain their RAM-model cost estimates until attack-specific access schedules have been specified.

4.8 Quantum Attacks

To date, the known quantum attacks against UOV are obtained by applying Grover's algorithm to exhaustive-search components of classical attacks, and no quantum attack is known that provides more than a quadratic speedup over the corresponding classical attack. Since Grover's algorithm provides at most a quadratic speedup for such searches, these attacks do not reduce the estimated concrete hardness of the recommended UOV parameter sets below their claimed security levels.

The NIST security levels 1, 3, and 5 are defined relative to the computational cost of key search against block ciphers with 128-, 192-, and 256-bit keys, respectively, such as AES-128, AES-192, and AES-256. In particular, Grover's algorithm reduces a classical exhaustive search with complexity 2^k to approximately $2^{k/2}$ quantum operations. Hence, an attack based solely on Grover's algorithm would need to achieve an additional speedup beyond this quadratic factor in order to violate the corresponding NIST security target. No such quantum speedup is currently known for UOV.

5 Implementations and Performance

We have presented *three* UOV variants as well as *four* recommended parameter sets in Section 3. This section specifies the implementations of these $12 = 3 \times 4$ UOV instances over various platforms, as well as their performance, so as to evaluate the practical performance of UOV across several representative platforms. Please refer to [11] for the comprehensive and detailed information regarding our implementations.

5.1 Common Implementation Techniques

First, we describe our implementation techniques for linear equation solving in signing and for verification, which are shared among our software implementations.

5.1.1 Solving linear equations

The UOV signing algorithm needs to solve the system of linear equations $\mathbf{Ax} = \mathbf{b}$ for the m -dimensional vector $\mathbf{x} \in \mathbb{F}_q^m$, where $\mathbf{A} = \mathbf{L}$ and $\mathbf{b} = \mathbf{t} - \mathbf{y}$. For this we apply Algorithm 1, a constant-time Gaussian elimination algorithm with back-substitution. As the first step (line 3) in the outer loop, we conditionally add all following rows to make sure the pivot element $a'_{i,i}$ is non-zero. This has to be performed in constant time, *i.e.*, the addition has to be performed for all following rows. In case it is still zero, we return $\perp$ (line 8) as the matrix is not invertible or the system of linear equations has no unique solution. Leaking that the matrix is not invertible via a timing side-channel is not an issue as the matrix is discarded if it is not invertible. Then, we invert the pivot element (line 9) and multiply the current row by the inverse (line 10). We then add multiples of that row to the remainder of the matrix (line 12), and back-substitute the variables into the system of equations to obtain the solution (line 15). Note that in the previous works [32, 83], the solution $\mathbf{x}$ is obtained by first computing the inverse of the matrix $\mathbf{A}$, followed by a matrix multiplication, *i.e.*, $\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b}$. This approach is less efficient, as pointed out in [11].

Reducing the number of conditional additions. For Algorithm 1, we have to perform a large number of conditional additions in lines 3-6 to achieve constant-time behavior. In practice, most of these additions will not actually be performed as the pivot element is already nonzero. Instead, we propose an optional performance optimization to limit the additions to a small number of rows. We propose to add at most 15 rows for $\mathbb{F}_{16}$ and at most 7 rows for $\mathbb{F}_{256}$. This results in a probability of at most $m \cdot 16^{-16} = 2^{-58}$ and $m \cdot 256^{-8} \leq 2^{-57.4}$ to wrongly abort for the $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ parameters, respectively, and is therefore negligible for the target security levels. While this trick does result in faster signing speed, it complicates formal verification of code. Hence, this optimization trick is disabled in our implementations by default.

5.1.2 Verification

For UOV verification, we evaluate the public map represented by a Macaulay matrix at the variables given by the signature $\mathbf{s}$ and verify that the output equals the hash digest of the message. Note that UOV verification is exactly the same as that of Rainbow [38] and, thus, the same techniques apply. We make use of a technique first introduced by Chou, Kannwischer, and Yang [32]: instead of multiplying the monomials $s_i s_j$ by the corresponding column of the Macaulay matrix and accumulating it into a single accumulator, we use multiple accumulators and do not perform any multiplication while passing through the matrix. At the end of verification, each accumulator is multiplied by the corresponding field element to obtain the final result. This allows for delaying all multiplications to the end and, hence, substantially reducing the number of required multiplications. This results in a substantial speed-up. In the case of $\mathbb{F}_{16}$, we use 15 accumulators: one for each

Algorithm 1 Constant-time linear equation solving using Gaussian elimination directly

Input: Linear equation $\mathbf{Ax} = \mathbf{b}$ **Output:** Solution $\mathbf{x} \in \mathbb{F}_q^m$ or $\perp$

```

1:  $\mathbf{A}' := [\mathbf{A} \mid \mathbf{b}] \in \mathbb{F}_q^{m \times (m+1)}$   $\triangleright \mathbf{A}' = [a'_{i,j}]$ 
2: for  $i = 0$  upto  $m - 1$  do
3:   for  $j = i + 1$  upto  $m - 1$  do
4:     if  $a'_{i,i} == 0$  then
5:       for  $k = i$  upto  $m$  do
6:          $a'_{i,k} := a'_{i,k} + a'_{j,k}$ 
7:     if  $a'_{i,i} == 0$  then
8:       return  $\perp$ 
9:      $p_i^{-1} := (a'_{i,i})^{-1}$ 
10:    for  $k = i$  upto  $m$  do
11:       $a'_{i,k} := p_i^{-1} \cdot a'_{i,k}$ 
12:    for  $j = i + 1$  upto  $m - 1$  do
13:      for  $k = i$  upto  $m$  do
14:         $a'_{j,k} := a'_{j,k} + a'_{j,i} \cdot a'_{i,k}$ 
15:  for  $i = m - 1$  downto  $1$  do
16:    for  $j = 0$  upto  $i - 1$  do
17:       $a'_{j,m} := a'_{j,m} + a'_{j,i} \cdot a'_{i,m}$ 
18: return last column of  $\mathbf{A}'$ 

```

possible value of $s_i s_j$ except for zero as those columns can be discarded straight away. In the case of $\mathbb{F}_{256}$, we use two sets of accumulators, each with 15 ones: one set for the four least significant bits, and the other set for the four most significant bits. Each column gets added to the corresponding accumulator of each set. By using different accumulators for the high and low bits, we keep the memory requirements for this approach reasonable while still substantially reducing the number of required costly field multiplications. Note that this approach results in signature-dependent memory access patterns which may be problematic in case signatures are secret and the targeted device leaks memory addresses, *e.g.*, through cache timing side channels. For the majority of cases, however, the signature is public and this approach should be used for verification speed.

Skipping parts of the public key. As already pointed out by Chou, Kannwischer, and Yang [32], the verification can be further sped-up by observing that for the zero monomials, their corresponding columns in the Macaulay matrix do not affect the result. Hence, we skip ahead if either s_i or s_j is zero. This is particularly significant when working with $\mathbb{F}_{16}$ as 1/16 of variables are expected to be zero, which means 31/256 of the products $s_i s_j$ are expected to be zero.

“Lazy sampling”. Recall that in the verification algorithm of UOV, if the public key is in its compressed representation, the matrices $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(2)}$ are first sampled pseudo-randomly by computing $\text{Expand}_{\mathbf{P}}(\text{seed}_{\text{pk}})$. Although the direct implementation is to first sample the entire pseudo-random part before conducting the usual verification operation, careful analysis shows that speed-up is possible if some variables in the signature are zero. To be precise, we can simply advance the state of the PRNG (through a function `prng_skip`) by increasing the counter of `aes128ctr` state. This optimization technique is referred to as “*lazy sampling*”. Note that this optimization is made possible by choosing a PRNG construction that allows sampling output at arbitrary positions. This was not

possible with previous constructions, *e.g.*, used within Rainbow which requires sampling all the output sequentially. It would also not be possible when using a sponge-based extendable-output function (XOF) like `shake256` which may have appeared to be a natural choice for seed expansion. Benchmarking results show that lazy sampling can result in a significant speed-up, especially for those UOV instances with $q = 16$.

5.1.3 Signing with a compressed secret key

In the signing algorithm, it builds the matrix $\mathbf{L} \in \mathbb{F}_q^{m \times m}$ whose i -th row is $\mathbf{v}^\top \mathbf{S}_i$, where

$$\mathbf{S}_i = \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top} \right) \mathbf{O} + \mathbf{P}_i^{(2)}.$$

Forming the $\mathbf{S}_i$'s explicitly costs $\Theta(v^2 m^2)$ field operations, as each of the m matrices is a $v \times v$ by $v \times m$ product. In the `pkc+skc` variant, the secret key is stored as a seed only. Therefore, the $\mathbf{S}_i$'s are not readily available, and `Expandsk(·)` must recompute them for every signature. The direct implementation first computes the m matrices $\mathbf{S}_i$, and then multiplies each matrix by the vinegar vector $\mathbf{v}$.

Following Amagasa, Ueno, and Homma [5], since the signing algorithm only requires the product $\mathbf{v}^\top \mathbf{S}_i$ rather than the explicitly formed $\mathbf{S}_i$, this computation can be re-ordered as

$$\mathbf{v}^\top \mathbf{S}_i = \mathbf{v}^\top \mathbf{P}_i^{(2)} + \left(\mathbf{v}^\top \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top} \right) \right) \mathbf{O}.$$

In the re-ordered computation, $\mathbf{v}^\top \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top} \right)$ is a vector-matrix product costing $\Theta(v^2)$ and its multiplication by $\mathbf{O}$ costs $\Theta(vm)$, for a total of $\Theta(v^2 m + vm^2)$ over all $i \in [m]$.

The technique is transparent to the specification, producing identical signatures, and applies specifically to the `pkc+skc` variant, which is precisely the case where the $\mathbf{S}_i$'s would otherwise be built and discarded for every signing operation. However, this optimization introduces a performance trade-off regarding singular matrices. In the straightforward approach, the $\mathbf{S}_i$'s are computed once outside the retry loop. If $\mathbf{L}$ turns out to be singular (with probability roughly $1/q$), sampling a new $\mathbf{v}$ only requires one $\mathbf{v}^\top \mathbf{S}_i$ re-computation. With the re-ordered computation, the entire $\Theta(v^2 m + vm^2)$ evaluation must be performed for every failed attempt.

5.2 x86 AVX2 Implementation

In this subsection we present our optimization of UOV implementations for x86-64 platforms, which is designated as the reference platform in NIST PQC standardization [67]. More precisely, we focus on the optimization for the AVX2 instruction set, which is arguably the most useful instruction set for its availability on modern x86 platforms. In addition to the implementations of UOV on the recommended Intel Haswell microarchitecture, we also investigate those on the Intel Skylake microarchitecture.

Symmetric primitives. For the instantiations of the four symmetric primitives `Hash(·)`, `Expandv(·)`, `Expandsk(·)`, and `ExpandP(·)`, we call the OpenSSL library when relating to standard cryptographic primitives, *e.g.*, `shake256` and `aes128`. Specifically, for the instantiation of `ExpandP(·)` with round-reduced AES, we adapt the `aes128ctr` implementation in [50], which utilizes x86 AES instructions, to implement only 4 AES rounds.

Native hardware field operations. Since our chosen $\mathbb{F}_{256}$ representation aligns with the standard AES format, we leverage native hardware support for field operations through the new GFNI [53] instruction set as one of our supported architectures. For the $\mathbb{F}_{256}$ parameter sets, the `GF2P8MULB` instruction multiplies two vectors of field elements directly. The $\mathbb{F}_{16}$ parameter set benefits from GFNI as well. Multiplication by a fixed $b \in \mathbb{F}_{16}$ is an

Table 6: Benchmarking results of AVX2 implementations. Numbers are the median CPU cycles of 1000 signing and verification operations, and of 100 key generations.

	KeyGen	Haswell Sign	Verify	KeyGen	Skylake Sign	Verify
<code>uov-Ip-classic</code>	3 587 232	120 984	83 724	3 275 886	117 422	74 252
<code>uov-Ip-pkc</code>	3 537 640		354 036	3 189 084		252 876
<code>uov-Ip-pkc+skc</code>	3 523 032	573 328		3 218 816	471 258	
<code>uov-Is-classic</code>	4 895 116	136 712	66 408	4 253 780	120 526	60 660
<code>uov-Is-pkc</code>	4 846 220		409 428	4 314 950		286 888
<code>uov-Is-pkc+skc</code>	4 362 000	678 528		3 978 980	520 234	
Dilithium 2 [†] [62]	97 621*	281 078*	108 711*	70 548	194 892	72 633
Falcon-512 [77]	19 189 801*	792 360*	103 281*	26 604 000	948 132	81 036
SPHINCS+ [‡] [52]	1 334 220	33 651 546	2 150 290	1 510 712*	50 084 397*	2 254 495*
<code>uov-III-classic</code>	22 512 648	326 116	245 868	18 459 512	299 814	216 772
<code>uov-III-pkc</code>	22 337 004		1 383 436	18 360 852		967 100
<code>uov-III-pkc+skc</code>	23 996 620	2 111 064		18 851 832	1 628 832	
<code>uov-V-classic</code>	66 626 076	673 972	463 424	53 227 288	648 574	433 102
<code>uov-V-pkc</code>	65 580 064		3 183 152	51 692 258		2 229 086
<code>uov-V-pkc+skc</code>	59 292 764	4 499 944		48 217 758	3 467 808	

[†] Security level II. [‡] Sphincs+-SHA2-128f-simple. * Data from SUPERCOP [40].

$\mathbb{F}_2$ -linear map and is therefore a 4×4 matrix over $\mathbb{F}_2$. Since two $\mathbb{F}_{16}$ elements are packed into a byte, the induced map on a byte is the block-diagonal 8×8 matrix, which is exactly what `GF2P8AFFINEQB` applies. A single instruction thus multiplies by b the 64 field elements in a 256-bit register, and the matrix is derived from b by a short sequence that again begins with a `GF2P8AFFINEQB`. The matrix is built once, as when building a multiplication table, and reused throughout a matrix-vector product. Finally, `GF2P8AFFINEINVQB` yields the inverse of the pivot element for the $\mathbb{F}_{256}$ Gaussian elimination in Algorithm 1.

Target platform. We benchmark our AVX2 optimization of UOV on the Intel Haswell and the Intel Skylake architectures. The C source code is compiled with `gcc` version 15.2.0 under Ubuntu 24.04 LTS, and the performance numbers are measured on Intel Xeon E3-1230L v3 1.80GHz (Haswell) and Intel Xeon E3-1275 v5 3.60GHz (Skylake) with turbo boost and hyper-threading disabled. For benchmarking UOV with GFNI, we compare AVX2 and GFNI optimizations on an AMD EPYC 9754 2.25 GHz, using the same compiler and the same settings as Haswell and Skylake.

Results. Table 6 reports the performance of our AVX2 implementations and comparisons to other standard PQC schemes, whereas Table 7 shows similar results with round-reduced AES. Table 8 reports the comparisons between the AVX2 and GFNI implementations. In Table 6, we merge the numbers for the signing algorithms of the `classic` and `pkc` variants as well as those for the verification algorithms of the `pkc` and `pkc+skc` variants, indicating that they share the same implementations.

It can be seen from Table 6 that `uov-Ip` has the fastest signing speed. And `uov-Is` has the fastest verification speed although its public key is larger than `uov-Ip`, which stems from the fact that `uov-Ip` uses more XOR operations for the 2 sets of accumulators while evaluating $\mathbb{F}_{256}$ public polynomials (see Section 5.1.2).

For verification with compressed keys, the computation of `ExpandP(·)`, *i.e.*, `aes128ctr`, dominates the execution time, which can be seen by comparing with the results of 4-round AES in Table 7. The round-reduced AES improves the verification time by 41% to 49%, depending on the parameter set and the microarchitecture.

Table 7: Benchmarking results of AVX2 implementations using 4-round AES for public-key expansion. Numbers are the median CPU cycles of 1000 signing and verification operations, and of 100 key generations.

	KeyGen	Haswell Sign	Verify	KeyGen	Skylake Sign	Verify
uov-Ip-pkc	3 420 684	120 844	199 984	3 124 796	117 146	150 196
uov-Ip-pkc+skc	3 392 256	425 496		3 098 444	372 004	
uov-Is-pkc	4 552 276	136 692	213 804	4 099 510	121 118	156 784
uov-Is-pkc+skc	4 312 048	473 652		3 769 210	391 738	
uov-III-pkc	21 594 440	325 428	732 072	17 602 694	296 626	531 830
uov-III-pkc+skc	23 351 324	1 465 752		18 883 308	1 202 812	
uov-V-pkc	64 156 388	675 220	1 619 680	50 718 114	640 168	1 175 602
uov-V-pkc+skc	57 938 632	2 936 044		47 395 648	2 412 678	

Table 8: Comparisons between AVX2 and GFNI implementations. Numbers are the median CPU cycles of 1000 signing and verification operations, and of 100 key generations.

	AVX2 on EPYC 9754			GFNI on EPYC 9754		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify
uov-Ip-classic	2 250 945	80 550	59 265	1 293 255	51 930	59 040
uov-Ip-pkc	2 228 602		163 620	1 275 570		163 260
uov-Ip-pkc+skc	2 230 920	351 990		1 275 300	318 690	
uov-Is-classic	2 778 030	75 262	46 373	1 897 403	64 575	46 058
uov-Is-pkc	2 739 263		178 627	1 859 445		178 132
uov-Is-pkc+skc	2 593 552	353 970		1 646 955	313 493	
uov-III-classic	12 285 810	220 973	170 145	6 261 053	144 720	171 652
uov-III-pkc	12 215 610		614 633	6 154 605		616 410
uov-III-pkc+skc	12 561 592	1 156 117		6 968 227	1 028 160	
uov-V-classic	34 319 002	428 220	307 575	19 811 498	294 660	305 617
uov-V-pkc	34 088 242		1 374 008	19 591 898		1 370 700
uov-V-pkc+skc	33 715 462	2 242 058		15 494 220	1 767 915	

For the signing algorithm of the **pkc+skc** variant, the main computation is spent on expanding the compressed public/secret keys, although the re-ordering of Section 5.1.3 mitigates the expensive secret key expansion compared to the **classic** variant.

For signing and key generation algorithms that involve intensive field multiplications, native support for field multiplication significantly boosts performance. This holds for both $\mathbb{F}_{256}$ and $\mathbb{F}_{16}$ fields. Verification gains little in either field, since the technique of Section 5.1.2 has already removed the multiplications from its inner loop.

5.3 Arm Neon Implementation

In this subsection we present our optimization of UOV for the Armv8-A architecture.

Symmetric primitives. For those three symmetric primitives instantiated with **shake256** function, *i.e.*, **Hash**($\cdot$), **Expand_v**($\cdot$), and **Expand_{sk}**($\cdot$), we also call the OpenSSL library since it is generally available on most platforms.

In regard to the instantiation of **Expand_p**($\cdot$), we present two different Neon implementations for **aes128ctr** depending on the availability of Arm AES instructions. On platforms supporting AES instructions, *e.g.*, Apple M1, we implement the standard and

Table 9: Benchmarking results of our Neon implementations. Numbers are the median CPU cycles of 1000 signing and verification operations, and of 100 key generations.

	KeyGen	Cortex-A72 Sign	Verify	KeyGen	Apple M1 Sign	Verify
uov-Ip-classic	15 077 049	306 615	186 088	1 967 954	65 181	56 228
uov-Ip-pkc	14 956 930		4 757 803	1 949 002		129 561
uov-Ip-pkc+skc	14 963 282	5 332 289		1 949 300	276 774	
uov-Is-classic	30 433 505	545 080	151 045	3 368 233	91 886	46 887
uov-Is-pkc	30 122 976		5 635 489	3 352 707		139 068
uov-Is-pkc+skc	29 638 581	6 796 861		3 334 542	328 381	
Dilithium 2 [†] [18]	269 724	649 230	272 824	71 061	224 125	69 792
Falcon-512 [64]	—	1 044 600	59 900	—	459 200	22 700
uov-III-classic	85 389 052	1 677 905	648 004	10 775 236	195 425	204 193
uov-III-pkc	83 278 149		21 043 833	10 693 440		508 557
uov-III-pkc+skc	83 131 314	22 651 588		10 690 920	804 582	
uov-V-classic	383 892 736	3 814 851	1 471 972	33 327 979	412 237	419 622
uov-V-pkc	376 752 508		49 273 587	31 193 346		1 148 596
uov-V-pkc+skc	376 410 153	54 317 859		31 202 269	1 732 567	

[†] Security level II.

round-reduced `aes128ctr` with AES instructions. On platforms without AES instructions, *e.g.*, Raspberry Pi4b, we port the bitsliced implementation for 32-bit platforms in [3], which runs four parallelized 32-bit bitsliced instances, to the Neon instruction set, since Biesheuvel [19] reported bitsliced implementations outperform TBL-based implementations in the Linux kernel setting.

Target platform. We benchmark our Neon implementations of UOV on Raspberry Pi4b and Apple’s 2020 MacBook Air, both supporting 64-bit Armv8-A instruction set. The Raspberry Pi4b equips a Broadcom BCM2711 CPU (Arm Cortex-A72 CPU [6]) running at 1.8 GHz without Arm AES instructions. Its compiler is `clang` version 19.1.7 (`Debian 3~deb12u1`). The Macbook has an Apple M1 CPU running at 3.2 GHz with Arm AES instruction support. Its compiler is Apple `clang` version 21.0.0 (`clang-2100.1.1.101`).

Results. Table 9 reports the results of Neon UOV implementation and comparison with other PQC signatures on the two Armv8-A platforms, whereas Table 10 shows similar

Table 10: Benchmarking results of Neon implementations using 4-round AES for public-key expansion. Numbers are the median CPU cycles of 1000 signing and verification operations, and of 100 key generations.

	KeyGen	Cortex-A72 Sign	Verify	KeyGen	Apple M1 Sign	Verify
uov-Ip-pkc	12 144 991	298 126	1 927 674	1 916 034	65 151	94 689
uov-Ip-pkc+skc	12 133 435	2 485 690		1 914 795	241 738	
uov-Is-pkc	26 615 434	530 721	2 267 445	3 299 713	91 935	98 258
uov-Is-pkc+skc	26 107 546	3 263 817		3 292 148	284 705	
uov-III-pkc	71 306 157	1 655 393	9 178 208	10 534 110	195 636	361 876
uov-III-pkc+skc	71 203 869	10 654 573		10 547 013	656 363	
uov-V-pkc	347 213 415	3 813 727	20 603 385	30 848 339	412 336	794 958
uov-V-pkc+skc	353 429 716	26 029 873		30 845 668	1 370 311	

Table 11: Cortex-M4F cycle counts for our M4 implementations in comparison to the fastest implementations of the winners of the NIST PQC competition. For signing and verification we report the average of 1 000 executions.

		speed (clock cycles)		
	variant	KeyGen	Sign	Verify
uov-<i>Ip</i> (This work)	classic	178 018k	3 050k	1 433k
	pkc	229 685k		13 853k
	pkc+skc	229 685k	20 300k	(12 776k)
uov-<i>Is</i> (This work)	classic	195 769k	2 409k	616k
	pkc	203 364k		16 043k
	pkc+skc	296 200k	21 435k	(15 162k)
Dilithium-2 [2]		1 598k	4 083k	1 572k
Falcon-512 [77, 78]		163 994k	39 014k	473k
sphincs-sha256-128f-simple [78]		16 112k	400 443k	22 548k
sphincs-sha256-128s-simple [78]		1 031 755k	7 848 131k	7 711k

results when $\text{Expand}_{\mathbf{P}}(\cdot)$ is instantiated with round-reduced AES.

First, it can be seen from Tables 9 and 10 that **uov-*Ip*** has the best signing time which is consistent with the results of AVX2 implementation (Table 6). However, on the Neon platform, the performance lead of **uov-*Ip*** over **uov-*Is*** is greater than on the AVX2 platform. This is caused by the mismatch between the sizes of registers and vectors. When processing line 9 and 10 of the signature generation procedure in Figure 2, the vectors are of length 45 or 46 bytes for **uov-*Ip***. These vectors are actually processed as 16×3 bytes on Neon but 32×2 bytes on AVX2 due to their 128-bit or 256-bit registers. It is clear that the AVX2 implementation wastes more computations than Neon.

Moreover, for verification, due to the fewer accumulators on $\mathbb{F}_{16}$ (see Section 5.1.2), **uov-*Is*** outperforms **uov-*Ip*** in spite of its larger public key size. On the other hand, the verification time is proportional to the public key sizes for the **pkc** and **pkc+skc** variants, where $\text{Expand}_{\mathbf{P}}(\cdot)$ dominates the computation time.

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

Finally, the 4-round AES makes for an efficient $\text{Expand}_{\mathbf{P}}(\cdot)$ function such that the verification time of **pkc** variant is of the same order as other PQC schemes on Apple M1.

Table 12: For the *Is* parameter sets the keys are too large to fully fit in RAM, we write them to flash during key generation. Cycles in Table 11 exclude those required for flashing. This table contains cycles required for flashing and the total key generation cycles.

	variant	key generation w/o flashing (cc)	flashing (cc)	key generation w/ flashing (cc)
uov-<i>Is</i>	classic	195 769k	202 293k	398 062k
	pkc	203 364k	110 741k	314 105k
	pkc+skc	296 200k	18 284k	314 484k

5.4 Arm Cortex-M4 Implementation

This subsection presents our implementations of UOV for the Arm Cortex-M4. Due to the stack limitations of available Cortex-M4 cores, we restrict our implementations in this

Table 13: Cortex-M4F cycle counts when using 4-round AES for expanding the public key. This change primarily affects the verification procedure providing a $1.9\times$ speed-up for **uov-Ip** and a $2.1\times$ speed-up for **uov-Is**.

		speed (clock cycles)		
	variant	KeyGen	Sign	Verify
uov-Ip	pkc	222 969k	3 084k	7 131k
	pkc+skc	222 969k	13 563k	
uov-Is	pkc	194 918k	2 466k	7 593k
	pkc+skc	287 754k	13 042k	

subsection to the two recommended parameter sets for NIST security level 1, *i.e.*, **uov-Ip** and **uov-Is**.

Symmetric primitives. For the instantiations of $\text{Hash}(\cdot)$, $\text{Expand}_v(\cdot)$, and $\text{Expand}_{sk}(\cdot)$, we use `shake256` as implemented in `pqm4` [78] which integrates the Keccak permutation in `Armv7-M` assembly from the `XKCP` [36]. For the instantiation of $\text{Expand}_p(\cdot)$, we use the t-table AES implementation by Schwabe and Stoffelen [85]. We also modify said implementation to implement a round-reduced AES with only 4 rounds. We present results both for the 10-round and 4-round AES.

Target platform. We use the ST NUCLEO-L4R5ZI development board featuring a STM32L4R5ZI ultra-low-power Arm Cortex-M4F core with 640 KB of RAM, and 2048 KB of flash memory. It runs at a frequency of up to 120 MHz. However, we clock the device at 16 MHz allowing for zero wait-states when fetching instructions and data from flash. For benchmarking, we use the `pqm4` [78] benchmarking framework.

Keys exceeding RAM size. For the **uov-Is** parameter sets, the total size of the expanded secret key and the expanded public key is 743 KB which exceeds the RAM of our target platform. To still be able to benchmark all primitives, we split up key generation into secret key and public key computation. We then write the keys to flash memory as was previously proposed by Chen and Chou for Classic McEliece [25]. This requires minimal code modification while still being able to provide benchmarks for all parts of the scheme. Higher security levels, however, are out of reach for running on the Cortex-M4.

Table 11 contains the performance benchmarks for Arm Cortex-M4. We present in Table 11 cycle counts for all six instances for NIST security level 1. Due to timing variations (depending only on public data) in signing and verification, we perform 1 000 measurements and report the average. Note that public key compression does not affect signing performance, while secret key compression does not affect verification performance. For the **uov-Is**, the key generation cycles exclude the writing of keys to flash. We report the flashing cycles separately in Table 12. And Table 13 presents the cycle counts when using a round-reduced AES (4, instead of 10, rounds) for expanding the public key; it results in significantly faster verification, *i.e.*, $1.9\times$ for **uov-Ip** and $2.1\times$ for **uov-Is**.

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

5.5 FPGA Implementation

In this subsection, we present our FPGA design for the four UOV parameter sets presented in Table 4 and report their performance on three popular Xilinx platforms. We evaluate 32 configurations covering the supported key storage modes, the 4-round and full-round AES options, and the low-area and fully pipelined AES designs.

Since our design supports multiple UOV instances, we adopt a processor design that provides a custom instruction set dedicated to the computation of UOV functions. This way, we support the key generation, signing, and verification algorithms in Figure 3 with pre-loaded firmware using the proposed instructions. Our hardware modules can be roughly divided into the following three categories according to their functionalities: (1) an instruction memory for storing firmware and a decoder for decoding user code and sending control signals to other hardware modules for computation; (2) data memory responsible for storing UOV keys and data movement from and to the computation modules; and (3) the modules for performing the actual computations.

Target platforms. We evaluate `uov-1p` and `uov-1s` on the Xilinx ZedBoard, which contains a Zynq-7000™ Z-7020 device, and `uov-III` and `uov-V` on the Nexys Video, which contains an Artix-7 XC7A200T device [92]. Artix-7 is a hardware target platform recommended by NIST [4] for the PQC standardization effort. Consequently, other PQC schemes have also been implemented on Artix-7, allowing comparison to our implementation.

The `uov-V` parameters require more key memory than the XC7A200T can provide. We therefore also target the Xilinx ZCU106, which contains a Zynq UltraScale+ XCZU7EV device and provides UltraRAM. This platform allows all four parameter sets to be implemented using the same processor architecture and provides a common target for comparing their resource and timing characteristics. All designs are synthesized and implemented with Xilinx Vivado 2022.1 and constrained to a 10 ns clock period.

RTL validation and cycle counts. All 32 configurations pass RTL simulation for key generation, signing, and verification. In addition to checking that verification accepts a valid signature and rejects a modified message, we compare every generated signature with an independent C implementation. The reported cycle counts are obtained from these RTL simulations.

The signing cycle count in the `classic` variant is mainly determined by preparing and solving the linear system. The signing cycle count in the `pkc+skc` variant is dominated by the $\text{Expand}_{\text{sk}}(\cdot)$ function, specifically the calculation of $\mathbf{S}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\top})\mathbf{O} + \mathbf{P}_i^{(2)}$. This calculation takes approximately $(n - m)m(n - m + 15)$ cycles, where the 15 includes flow control and other operations such as loading from and storing to temporary storage.

The cycle count of verification in the `pkc` variant is limited by the throughput of the $\text{Expand}_{\text{p}}(\cdot)$ function. The AES module of our low-area design generates one 128-bit block every 12 cycles. Replacing the full-round AES with a 4-round AES therefore reduces the verification cycle count. In the fully pipelined design, both AES variants generate one 128-bit block per cycle, and their cycle counts are consequently close.

For the ZCU106 implementations, key generation in the `pkc` and `pkc+skc` modes includes a prologue that initializes the $\mathbf{P}^{(3)}$ accumulator before it is mapped to UltraRAM. UltraRAM does not provide the same initialization behavior as block RAM. The prologue adds 11 521 cycles for `uov-1p`, 23 073 cycles for `uov-1s`, 29 125 cycles for `uov-III`, and 51 505 cycles for `uov-V`. The `classic` mode initializes the accumulator as part of its ordinary computation and does not require this prologue.

Implementation results on the ZedBoard. The ZedBoard results for `uov-1p` and `uov-1s` are reported in Table 14, Table 15, and Table 16. The 4-round and full-round versions of the low-area AES have the same resource utilization. The `uov-1s-classic` implementation occupies all 140 block RAM tiles and uses 44 974 LUTs, whereas the compressed-key variants reduce the LUT requirement to 29 385 and 28 947, respectively. The estimated maximum frequencies of the reported ZedBoard implementations range from 90.3 to 96.83 MHz.

Implementation results on the Nexys Video. The Nexys Video results for `uov-III` and `uov-V` are reported in Table 17, Table 18, and Table 19. The successfully routed `uov-III` implementations have estimated maximum frequencies from 90.98 to 98.31 MHz. The

Table 14: The FPGA results on the ZedBoard (Zynq-7000 Z-7020) using the low-area, non-pipelined full-round AES design.

	LUTs	Utilization			Timing Freq. (MHz)	Cycle Count		
		FFs	BRAM	DSP		KeyGen	Sign	Verify
uov- <i>Ip-classic</i>	34 663	24 113	135.5	2	92.58	4 219 103	8 292	7 241
uov- <i>Ip-pkc</i>	33 123	23 018	93.5	2	92.69	4 939 730	8 292	230 425
uov- <i>Ip-pkc+skc</i>	30 887	23 236	57.5	2	96.83	4 465 662	459 308	230 425
uov- <i>Is-classic</i>	44 974	27 433	140	2	92.2	9 916 182	13 070	12 986
uov- <i>Is-pkc</i>	29 385	25 328	110	2	94.8	11 922 375	13 070	284 379
uov- <i>Is-pkc+skc</i>	28 947	24 444	66	2	90.8	11 072 933	843 885	284 379

Table 15: Cycle counts on the ZedBoard using the low-area 4-round AES design. The resource utilization and timing are the same as those of the corresponding full-round designs in Table 14.

	Cycle Count		
	KeyGen	Sign	Verify
uov- <i>Ip-classic</i>	4 048 691	8 292	7 241
uov- <i>Ip-pkc</i>	4 825 538	8 292	121 075
uov- <i>Ip-pkc+skc</i>	4 416 702	410 348	121 075
uov- <i>Is-classic</i>	9 746 742	13 070	12 986
uov- <i>Is-pkc</i>	11 814 183	13 070	176 859
uov- <i>Is-pkc+skc</i>	11 026 181	797 133	176 859

Table 16: The FPGA results on the ZedBoard using the fully pipelined AES design.

AES rounds		LUTs	Utilization			Timing Freq. (MHz)	Cycle Count		
			FFs	BRAM	DSP		KeyGen	Sign	Verify
10	uov- <i>Ip-pkc</i>	39 710	25 509	93.5	2	94.12	4 730 047	8 292	39 501
	uov- <i>Ip-pkc+skc</i>	37 369	25 685	57.5	2	95.43	4 376 387	370 033	39 501
	uov- <i>Is-pkc</i>	35 786	27 856	110	2	95.5	11 773 796	13 070	115 258
	uov- <i>Is-pkc+skc</i>	34 208	26 974	66	2	90.3	11 008 802	779 754	115 258
4	uov- <i>Ip-pkc</i>	35 131	23 998	93.5	2	91.80	4 729 693	8 292	38 937
	uov- <i>Ip-pkc+skc</i>	34 268	24 148	57.5	2	94.44	4 376 177	369 823	38 937
	uov- <i>Is-pkc</i>	31 227	26 327	110	2	94.2	11 772 350	13 070	113 914
	uov- <i>Is-pkc+skc</i>	31 006	25 444	66	2	92.4	11 008 124	779 076	113 914

low-area uov-*III-pkc* implementation uses 348.5 of the 365 available block RAM tiles, showing that the device is already close to its memory capacity.

The uov-*V-pkc+skc* design does not fit the XC7A200T. In the low-area design, Vivado reports 67 269 LUTs used as memory, whereas the device provides 46 200 compatible sites. The design also requires 368 RAMB36 blocks while 365 are available, and 139 387 Slice LUTs while 134 600 are available. Consequently, the utilization design-rule checks fail before placement. We retain its RTL cycle counts in the Nexys Video tables, but do not report utilization or routed timing results for this device.

Implementation results on the ZCU106. The ZCU106 results for all four parameter sets are reported in Table 20, Table 21, and Table 22. All 32 configurations complete routing and meet the 100 MHz target frequency. Their reported setup slack is positive, and their estimated maximum frequencies range from 101.64 to 147.56 MHz.

The UltraRAM resources of the XCZU7EV provide a suitable mapping for the key memories. For uov-*Is-classic*, moving part of the key memory to 16 UltraRAM blocks reduces the LUT count from 44 974 on the ZedBoard to 28 226 on the ZCU106. It

Table 17: The FPGA results on the Nexys Video (Artix-7 XC7A200T) using the low-area, non-pipelined full-round AES design. The **uov-V-pkc+skc** design does not fit the device; its cycle counts are obtained from RTL simulation.

	LUTs	Utilization			Timing Freq. (MHz)	Cycle Count		
		FFs	BRAM	DSP		KeyGen	Sign	Verify
uov-III-pkc	41 108	30 633	348.5	4	96.81	20 484 600	21 113	944 953
uov-III-pkc+skc	38 438	29 261	204.5	4	90.98	18 660 604	1 781 993	944 953
uov-V-pkc+skc	–	–	–	–	–	44 687 389	4 136 227	2 191 474

uses 30 block RAM tiles and reaches an estimated maximum frequency of 114.89 MHz. The **uov-Is-pkc** and **uov-Is-pkc+skc** implementations reach 130.04 and 116.67 MHz, respectively. The fully pipelined **uov-Is** designs reach estimated maximum frequencies between 125.71 and 147.56 MHz.

The non-pipelined **uov-V-pkc+skc** design uses 52 UltraRAM blocks and 39 762 LUTs. It completes routing with 1.848 ns positive slack and an estimated maximum frequency of 122.67 MHz. The fully pipelined full-round and 4-round AES designs also complete routing, with estimated maximum frequencies of 112.16 and 112.76 MHz, respectively. These results show that the processor architecture supports all four parameter sets when a device with sufficient memory capacity is used.

Table 18: Cycle counts for the low-area 4-round AES design targeted at the Nexys Video. The resource utilization is the same as that of the corresponding full-round design; **uov-V-pkc+skc** does not fit the device.

	Cycle Count		
	KeyGen	Sign	Verify
uov-III-pkc	20 035 392	21 113	496 121
uov-III-pkc+skc	18 460 372	1 581 761	496 121
uov-V-pkc+skc	44 202 685	3 651 523	1 142 030

Table 19: The FPGA results on the Nexys Video using the fully pipelined AES design. The **uov-V-pkc+skc** designs do not fit the device; their cycle counts are obtained from RTL simulation.

AES rounds		Utilization				Timing Freq. (MHz)	Cycle Count		
		LUTs	FFs	BRAM	DSP		KeyGen	Sign	Verify
10	uov-III-pkc	47 492	33 143	348.5	4	98.31	19 663 128	21 113	162 515
	uov-III-pkc+skc	44 910	31 805	204.5	4	92.92	18 295 588	1 416 977	162 515
	uov-V-pkc+skc	–	–	–	–	–	43 801 185	3 250 023	361 732
4	uov-III-pkc	43 096	31 676	348.5	4	95.83	19 661 292	21 113	160 997
	uov-III-pkc+skc	40 658	30 344	204.5	4	95.18	18 294 688	1 416 077	160 997
	uov-V-pkc+skc	–	–	–	–	–	43 800 201	3 249 039	359 914

Table 20: The FPGA results on the ZCU106 (Zynq UltraScale+ XCZU7EV) using the low-area, non-pipelined full-round AES design.

	LUTs	FFs	Utilization				DSP	Timing Freq. (MHz)	Cycle Count		
			BRAM	URAM	LUTMEM				KeyGen	Sign	Verify
uov- Ip -classic	32 491	23 167	37	12	671	2	104.83	4 219 103	8 292	7 241	
uov- Ip -pkc	31 603	23 180	43	6	664	2	112.51	4 951 251	8 292	230 425	
uov- Ip -pkc+skc	31 776	23 268	55	0	664	2	110.55	4 477 183	459 308	230 425	
uov- Is -classic	28 226	24 696	30	16	904	2	114.89	9 916 182	13 070	12 986	
uov- Is -pkc	28 571	24 688	14	12	900	2	130.04	11 945 448	13 070	289 756	
uov- Is -pkc+skc	28 173	24 354	34	4	313	2	116.67	11 096 006	843 885	289 756	
uov- III -pkc	35 839	29 289	60	36	416	4	127.05	20 513 725	21 113	944 953	
uov- III -pkc+skc	36 130	29 451	60	18	407	4	122.00	18 689 729	1 781 993	944 953	
uov- V -pkc+skc	39 762	34 345	67	52	435	4	122.67	44 738 894	4 136 227	2 191 474	

Table 21: Cycle counts on the ZCU106 using the low-area 4-round AES design. The resource utilization and timing are the same as those of the corresponding full-round designs in Table 20.

	Cycle Count		
	KeyGen	Sign	Verify
uov- Ip -classic	4 048 691	8 292	7 241
uov- Ip -pkc	4 837 059	8 292	121 075
uov- Ip -pkc+skc	4 428 223	410 348	121 075
uov- Is -classic	9 746 742	13 070	12 986
uov- Is -pkc	11 837 256	13 070	181 596
uov- Is -pkc+skc	11 049 254	797 133	181 596
uov- III -pkc	20 064 517	21 113	496 121
uov- III -pkc+skc	18 489 497	1 581 761	496 121
uov- V -pkc+skc	44 254 190	3 651 523	1 142 030

Table 22: The FPGA results on the ZCU106 using the fully pipelined AES design.

AES rounds		LUTs	FFs	Utilization				DSP	Timing Freq. (MHz)	Cycle Count		
				BRAM	URAM	LUTMEM				KeyGen	Sign	Verify
10	uov- Ip -pkc	37 660	25 809	43	6	729	2	105.47	4 741 568	8 292	39 501	
	uov- Ip -pkc+skc	37 584	25 698	55	0	721	2	115.62	4 387 908	370 033	39 501	
	uov- Is -pkc	34 771	27 201	14	12	961	2	125.71	11 796 869	13 070	28 820	
	uov- Is -pkc+skc	33 418	26 910	34	4	373	2	145.62	11 031 875	779 754	28 820	
	uov- III -pkc	41 701	31 905	60	36	472	4	124.44	19 692 253	21 113	162 515	
	uov- III -pkc+skc	42 317	31 806	60	18	467	4	111.62	18 324 713	1 416 977	162 515	
	uov- V -pkc+skc	46 709	36 863	67	52	499	4	112.16	43 852 690	3 250 023	361 732	
4	uov- Ip -pkc	34 589	24 233	43	6	719	2	101.64	4 741 214	8 292	38 937	
	uov- Ip -pkc+skc	33 462	24 166	55	0	720	2	103.94	4 387 698	369 823	38 937	
	uov- Is -pkc	30 580	25 671	14	12	957	2	144.86	11 795 423	13 070	27 662	
	uov- Is -pkc+skc	30 382	25 372	34	4	373	2	147.56	11 031 197	779 076	27 662	
	uov- III -pkc	37 769	30 360	60	36	464	4	125.58	19 690 417	21 113	160 997	
	uov- III -pkc+skc	37 645	30 370	60	18	479	4	110.57	18 323 813	1 416 077	160 997	
	uov- V -pkc+skc	42 698	35 225	67	52	499	4	112.76	43 851 706	3 249 039	359 914	

6 Summary: Advantages and Limitations

In this section we summarize the advantages and the limitations of our UOV in this submission.

In comparison with other post-quantum digital signature schemes, the main advantages of the UOV signature scheme are:

Performance. The computationally dominant part of UOV signing consists of linear-algebraic operations over small finite fields, such as matrix vector multiplication and solving linear systems over small finite fields. Therefore, the UOV scheme can be implemented very efficiently and achieves competitive signing performance in the benchmark environments evaluated in this document.

Signature Size. The signatures produced by the recommended UOV instances range from 96 to 275 bytes, and are much shorter than standardized post-quantum digital signature schemes such as Dilithium [62], Falcon [75], and SPHINCS+ [52] of RSA and those of other post-quantum signature schemes.

Resource Requirements. Since UOV only requires simple linear algebra operations over a small finite field, it can be efficiently implemented on low cost devices, without the need of a cryptographic coprocessor.

Security Considerations. Since its invention in 1999, no efficient attack against UOV has been found. Moreover, based on the attack analyses and concrete cost models considered in this document, no currently known attack is estimated to reduce the recommended UOV parameter sets in Table 4 below their claimed security levels. It is worth emphasizing that in contrast to some other post-quantum schemes, the theoretical complexities of the known attacks against UOV match the experimental data very well. Based on the attack analyses and concrete cost models considered in this document, we assign the recommended parameter sets the security levels reported in Table 4.

Implementation Simplicity. The UOV construction is relatively simple and relies primarily on standard linear-algebraic operations over finite fields. This simplicity also implies that there are not many structures of the scheme which could be fully utilized.

On the other hand, the main disadvantage of UOV is the large size of the public keys. The public key sizes of UOV are, for security levels beyond 128 bit, much larger than those of classical schemes such as RSA and ECC, and some other post-quantum schemes. And the practical impact of the large public-key size depends on the target application and available memory. Furthermore, to mitigate this disadvantage, we propose the **pkc** and **pkc+skc** variants of UOV with smaller keys to support different trade-offs between persistent storage, RAM usage, and computation.

References

- [1] Gorjan Alagic, Daniel Apon, David Cooper, Quynh Dang, Thinh Dang, John Kelsey, Jacob Lichtinger, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, Daniel Smith-Tone and Yi-Kai Liu. NIST IR 8413-upd1, status report on the third round of the NIST post-quantum cryptography standardization process, September 2022. Available at <https://csrc.nist.gov/publications/detail/nistir/8413/final>, last accessed on August 20, 2026.
- [2] Amin Abdulrahman, Vincent Hwang, Matthias J. Kannwischer and Daan Sprenkels. Faster Kyber and Dilithium on the Cortex-M4. In *ACNS 22*, LNCS vol. 13269, pp. 853–871. Springer, 2022.
- [3] Alexandre Adomnicai and Thomas Peyrin. Fixslicing AES-like ciphers: New bitsliced AES speed records on Arm-Cortex M and RISC-V. In *IACR TCHES*, vol. 2021(1), pp. 402–425, 2021.
- [4] Daniel Apon. NIST assignments of platforms on implementation efforts to PQC teams. Available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/cJxMq0_90gU. Online February 7, 2019, last accessed on August 20, 2026.
- [5] Hiroshi Amagasa, Rei Ueno and Naofumi Homma. AVX2 Implementation of QR-UOV for modern x86 processors. In *IACR TCHES 2026(2)*, pp. 325–346, 2026.
- [6] Arm Ltd. Arm Cortex-A72 software optimization guide, 2015. Available at <https://developer.arm.com/documentation/uan0016/a/>, last accessed on August 20, 2026.
- [7] Magali Bardet, Maxime Bros, Daniel Cabarcas, Philippe Gaborit, Ray Perlner, Daniel Smith-Tone, Jean-Pierre Tillich and Javier Verbel. Improvements of algebraic attacks for solving the rank decoding and MinRank problems. In *Asiacrypt 2020*, LNCS vol. 12491, pp. 507–536. Springer, 2020.
- [8] John Baena, Pierre Briaud, Daniel Cabarcas, Ray Perlner, Daniel Smith-Tone and Javier Verbel. Improving support-minors rank attacks: Applications to GeMSS and Rainbow. IACR Cryptology ePrint Archive, Report 2021/1677.
- [9] Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir and Bo-Yin Yang. Fast exhaustive search for polynomial systems in $\mathbb{F}_2$. In *CHES 2010*, LNCS vol. 6225, pp. 203–218. Springer, 2010.
- [10] Ward Beullens, Fabio Campos, Sofía Celi, Basil Hess and Matthias J. Kannwischer. MAYO: Round 1 submission document in NIST’s PQC Project for Additional Digital Signature Schemes. Available at <https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures>, last accessed on August 20, 2026.
- [11] 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 Cryptology ePrint Archive, Report 2023/059.
- [12] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange and Christine van Vredendaal. NTRU Prime: Reducing attack surface at low cost. Available at <https://ntruprime.cr.yp.to/ntruprime-20170816.pdf>, last accessed on August 20, 2026.
- [13] Johannes Buchmann, Jintai Ding, Mohamed Saied Emam Mohamed and Wael Said Abd Elmageed Mohamed. MutantXL: solving multivariate polynomial equations for cryptanalysis. In *Symmetric Cryptography*, vol. 9031, pp. 1–7, 2009.

- [14] Ward Beullens. Improved cryptanalysis of UOV and Rainbow. In *EUROCRYPT 2021(1)*, LNCS vol. 12696, pp. 348–373. Springer, 2021.
- [15] Ward Beullens. Breaking Rainbow takes a weekend on a laptop. In *CRYPTO 2022(2)*, LNCS vol. 13508, pp. 464–479. Springer, 2022.
- [16] Magali Bardet, Jean-Charles Faugère and Bruno Salvy. Complexity of Gröbner basis computation for semi-regular overdetermined sequences over $\mathbb{F}_2$ with solutions in $\mathbb{F}_2$. Research Report RR-5049, INRIA, December 2003.
- [17] Olivier Billet and Henri Gilbert. Cryptanalysis of Rainbow. In *SCN 2006*, LNCS vol. 4116, pp. 336–347. Springer, 2006.
- [18] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang and Shang-Yi Yang. Neon NTT: faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. In *IACR TCHES*, vol. 2022(1), pp. 224–244, 2022.
- [19] Ard Biesheuvel. Accelerated AES for Arm64 Linux kernel, 2017. Available at <https://web.archive.org/web/20240422014047/https://www.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/>, last accessed on August 20, 2026.
- [20] Richard P. Brent and H. T. Kung. The area-time complexity of binary multiplication. In *Journal of the ACM*, vol. 28, pp. 521–534, 1981.
- [21] Charles Bouillaguet. Nice attacks - but what is the cost computational models for cryptanalysis. IACR Cryptology ePrint Archive, Report 2022/197.
- [22] Ward Beullens, Bart Preneel, Alan Szepieniec and Frederik Vercauteren. LUOV signature scheme proposal for NIST PQC project (Round 2 version), 2019. Available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions>, last accessed on August 20, 2026.
- [23] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS 1993*, ACM, pp. 62–73.
- [24] An Braeken, Christopher Wolf and Bart Preneel. A study of the security of Unbalanced Oil and Vinegar signature schemes. In *CT-RSA 2005*, LNCS vol. 3376, pp. 29–43. Springer, 2005.
- [25] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. In *IACR TCHES*, vol. 2021(3), pp. 125–148, 2021.
- [26] Chen-Mou Cheng, Tung Chou, Ruben Niederhagen and Bo-Yin Yang. Solving quadratic equations with XL on parallel architectures - extended version. IACR Cryptology ePrint Archive, Report 2016/412.
- [27] Antoine Casanova, Jean-Charles Faugère, Gilles Macario-Rat, Jacques Patarin, Ludovic Perret and Jocelyn Ryckeghem. GeMSS 3rd round submission, NIST submission document and technical report, October 2020. Available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>, last accessed on August 20, 2026.
- [28] Nicolas T. Courtois, Louis Goubin, Willi Meier and Jean-Daniel Tacier. Solving underdefined systems of multivariate quadratic equations. In *PKC 2002*, LNCS vol. 2274, pp. 211–227. Springer, 2002.

- [29] Weiwei Cao, Lei Hu, Jintai Ding and Zhijun Yin. Kipnis-Shamir attack on Unbalanced Oil-Vinegar scheme. In *ISPEC 2011*, LNCS vol. 6672, pp. 168–180. Springer, 2011.
- [30] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson and Wen Wang. Round-4 supporting documentation submitted to the NIST Post-Quantum Cryptography Standardization Process. Available at <https://classic.mceliece.org/mceliece-security-20221023.pdf>, last accessed on August 20, 2026.
- [31] Nicolas T. Courtois, Alexander Klimov, Jacques Patarin and Adi Shamir. Efficient algorithms for solving overdefined systems of multivariate polynomial Equations. In *EUROCRYPT 2000*, LNCS vol. 1807, pp. 392–407. Springer, 2000.
- [32] Tung Chou, Matthias J. Kannwischer and Bo-Yin Yang. Rainbow on Cortex-M4. In *IACR TCHES*, vol. 2021(4), pp. 650–675, 2021.
- [33] Jintai Ding, Ming-Shing Chen, Matthias J. Kannwischer, Albrecht Petzoldt, Jacques Patarin, Dieter Schmidt and Bo-Yin Yang. Rainbow 3rd round submission, NIST submission document and technical report, October 2020. Available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>, last accessed on August 20, 2026.
- [34] Don Coppersmith. Solving homogeneous linear equations over $GF(2)$ via block Wiedemann algorithm. In *Mathematics of Computation*, Vol. 62(205), pp. 333–350. AMS, 1994.
- [35] Jintai Ding, Hao Guo and Bo-Yin Yang. On memory effects in PWXL variants. IACR Cryptology ePrint Archive, Report 2026/1819.
- [36] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche and Ronny Van Keer. eXtended keccak code package. Available at <https://github.com/XKCP/XKCP>, last accessed on August 20, 2026.
- [37] Jintai Ding, Peigen Li and Siyong Tao. Revisiting the wedge attack on UOV problem. IACR Cryptology ePrint Archive, Report 2026/1548.
- [38] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *ACNS 2005*, LNCS vol. 3531, pp. 164–175. Springer, 2005.
- [39] Jintai Ding, Bo-Yin Yang, Chia-Hsin Owen Chen, Ming-Shing Chen and Chen-Mou Cheng. New differential-algebraic attacks and reparametrization of Rainbow. In *ACNS 2008*, LNCS vol. 5037, pp. 242–257. Springer, 2008.
- [40] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. Available at <https://bench.cr.yp.to>, last accessed on August 20, 2026.
- [41] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). In *Journal of Pure and Applied Algebra*, vol. 139(1–3), pp. 61–88, 1999.
- [42] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases without reduction to zero (F_5). In *ISSAC 2002*, pp. 75–83. ACM, 2002.

- [43] Jean-Charles Faugère, Mohab Safey El Din and Pierre-Jean Spaenlehauer. Computing loci of rank defects of linear matrices using Gröbner bases and applications to cryptography. In *ISSAC 2010*, pp. 257–264. ACM, 2010.
- [44] Hiroki Furue and Yasuhiko Ikematsu. Key recovery attacks on UOV using p^ℓ -truncated polynomial rings. In *CRYPTO 2026*, LNCS vol. 16803, pp. 289–318. Springer, 2026.
- [45] FIPS PUB 197 – Advanced Encryption Standard (AES), 2001. Available at <https://doi.org/10.6028/NIST.FIPS.197>, last accessed on August 20, 2026.
- [46] FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, 2015. Available at <https://doi.org/10.6028/NIST.FIPS.202>, last accessed on August 20, 2026.
- [47] Hiroki Furue, Shuhei Nakamura and Tsuyoshi Takagi. Improving Thomae-Wolf algorithm for solving underdetermined multivariate quadratic polynomial problem. In *PQCrypto 2021*, LNCS vol. 12841, pp. 65–78. Springer, 2021.
- [48] Louis Goubin and Nicolas T. Courtois. Cryptanalysis of the TTM cryptosystem. In *Asiacrypt 2000*, LNCS vol. 1976, pp. 44–57. Springer, 2000.
- [49] Michael R. Garey and David S. Johnson. Computers and intractability: A guide to the theory of NP-Completeness. W. H. Freeman, 1979.
- [50] Shay Gueron. Intel advanced encryption standard (AES) new instructions set, 2010. Available at <https://www.intel.com.bo/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf>, last accessed on August 20, 2026.
- [51] Yasufumi Hashimoto. An improvement of algorithms to solve under-defined systems of multivariate quadratic equations. IACR Cryptology ePrint Archive, Report 2021/1045.
- [52] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas Westerbaan and Ward Beullens. SPHINCS+. Technical report, National Institute of Standards and Technology, 2020. Available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>, last accessed on August 20, 2026.
- [53] Intel. Galois field new instructions (GFNI) technology guide. Available at https://cdrdv2.intel.com/v1/dl/getContent/644497?fileName=GFNI_TechGuide_644497v2.pdf, last accessed on August 20, 2026.
- [54] Yi Jin, Yuansheng Pan, Xiaou He, Boru Gong and Jintai Ding. Security analysis on UOV families with odd characteristics: Using symmetric algebra. In *PKC 2026*, LNCS vol. 16551, pp. 428–454. Springer, 2026.
- [55] Jonathan Katz. Digital signatures. Springer, 2010.
- [56] Jonathan Katz and Yehuda Lindell. Introduction to modern cryptography, 3rd Edition. CRC Press, 2020.

- [57] Aviad Kipnis, Jacques Patarin and Louis Goubin. Unbalanced Oil and Vinegar signature schemes. In *EUROCRYPT 1999*, LNCS vol. 1592, pp. 206–222. Springer, 1999.
- [58] Aviad Kipnis and Adi Shamir. Cryptanalysis of the Oil and Vinegar signature scheme. In *CRYPTO 1998*, LNCS vol. 1462, pp. 257–266. Springer, 1998.
- [59] Aviad Kipnis and Adi Shamir. Cryptanalysis of the HFE public key cryptosystem by relinearization. In *CRYPTO 1999*, LNCS vol. 1666, pp. 19–30. Springer, 1999.
- [60] Liam Keliher and Jiayuan Sui. Exact maximum expected differential and linear probability for two-round advanced encryption standard. In *IET Information Security*, vol. 1(2), pp. 53–57, 2007.
- [61] Roberto Avanzi, Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler and Damien Stehlé. CRYSTALS-Kyber algorithm specifications and supporting documentation, 2020. Third-round submission to the NIST Post-Quantum Cryptography Standardization Process. Available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>, last accessed on August 20, 2026.
- [62] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. Available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>, last accessed on August 20, 2026.
- [63] Hiroyuki Miura, Yasufumi Hashimoto and Tsuyoshi Takagi. Extended algorithm for solving underdefined multivariate quadratic equations. In *PQCrypto 2013*, LNCS vol. 7932, pp. 118–135. Springer, 2013.
- [64] Duc Tri Nguyen and Kris Gaj. Fast Falcon signature generation and verification using Armv8 NEON instructions, 2022. Available at <https://csrc.nist.gov/csrc/media/Events/2022/fourth-pqc-standardization-conference/documents/papers/fast-falcon-signature-generation-and-verification-pqc2022.pdf>, last accessed on August 20, 2026.
- [65] Ruben Niederhagen. Parallel Cryptanalysis. PhD thesis, Eindhoven University of Technology, 2012. Available at <http://www.polycephaly.org/thesis/index>, last accessed on August 20, 2026.
- [66] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process, December 2016. Available at <https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf>, last accessed on August 20, 2026.
- [67] National Institute of Standards and Technology. Call for additional digital signature schemes for the post-quantum cryptography standardization process, September 2022. Available at <https://csrc.nist.gov/csrc/media/Projects/pqc-dig-sig/documents/call-for-proposals-dig-sig-sept-2022.pdf>, last accessed on August 20, 2026.

- [68] NTRU Prime Team. NTRU Prime: NIST Round 3 submission document. Available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>, last accessed on August 20, 2026.
- [69] Horace Ong, Claus-Peter Schnorr and Adi Shamir. An efficient signature scheme based on quadratic equations. In *STOC 1984*, pp. 208–216. ACM, 1984.
- [70] Paul C. van Oorschot and Michael J. Wiener. Improving implementable meet-in-the-middle attacks by orders of magnitude. In *CRYPTO 1996*, LNCS vol. 1109, pp. 229–236. Springer, 1996.
- [71] Jacques Patarin. Cryptanalysis of the Matsumoto and Imai public key scheme of Eurocrypt’88. In *CRYPTO 1995*, LNCS vol. 963, pp. 248–261. Springer, 1995.
- [72] Jacques Patarin. The oil and vinegar signature scheme. Presented at the *Dagstuhl Workshop on Cryptography*, September 1997.
- [73] Albrecht Petzoldt, Stanislav Bulygin and Johannes Buchmann. CyclicRainbow – A multivariate signature scheme with a partially cyclic public key. In *INDOCRYPT 2010*, LNCS vol. 6498, pp. 33–48. Springer, 2010.
- [74] Albrecht Petzoldt. Efficient key generation for Rainbow. In *PQCrypto 2020*, LNCS vol. 12100, pp. 92–107. Springer, 2020.
- [75] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Ricosset, Gregor Seiler, William Whyte and Zhenfei Zhang. Falcon. Technical report, National Institute of Standards and Technology, 2020. Available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions>, last accessed on August 20, 2026.
- [76] Jacques Patarin and Louis Goubin. Trapdoor one-way permutations and multivariate polynomials. In *ICICS 1997*, LNCS vol. 1334, pp. 356–368. Springer, 1997.
- [77] Thomas Pornin. New efficient, constant-time implementations of Falcon. IACR Cryptology ePrint Archive, Report 2019/893.
- [78] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe and Ko Stoffelen. PQM4: Post-quantum crypto library for the Arm Cortex-M4. Available at <https://github.com/mupq/pqm4>, last accessed on August 20, 2026.
- [79] The Rainbow Team. Modified parameters of Rainbow in response to a refined analysis of the Rainbow Band Separation attack by the NIST Team and the recent new MinRank attacks. In September 2020, available at <http://precision.moscito.org/by-publ/recent/rainbow-pars.pdf>, last accessed on August 20, 2026.
- [80] Lars Ran. Wedges, oil, and vinegar - An analysis of UOV in the exterior algebra. In *EUROCRYPT 2026*, LNCS vol. 16544, pp. 363–388. Springer, 2026.
- [81] Markku-Juhani O. Saarinen and Ward Beullens. Bit security of UOV 1.0 (and MAYO). Available at https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/S1Bu7_oITN4/m/1MW-0VyfBgAJ, last accessed on August 20, 2026.
- [82] Sophie Schmieg. Comments for round second on ramp candidate UOV. Available at <https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/1CrLRB4RzFU/m/t2WibzZrAAAAJ>, last accessed on August 20, 2026.

- [83] Kyung-Ah Shim, Sangyub Lee and Namhun Koo. Efficient implementations of Rainbow and UOV using AVX2. In *IACR TCHES 2022(1)*, pp. 245–269, 2022.
- [84] Koichi Sakumoto, Taizo Shirai and Harunaga Hiwatari. On provable security of UOV and HFE signature schemes against chosen-message attack. In *PQCrypto 2011*, LNCS vol. 7071, pp. 68–82. Springer, 2011.
- [85] Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In *SAC 2016*, LNCS vol. 10532, pp. 180–194. Springer, 2016.
- [86] Nigel Smart. ‘Bristol Fashion’ MPC Circuits. Available at <https://nigelsmart.github.io/MPC-Circuits/>, last accessed on August 20, 2026.
- [87] Chengdong Tao, Albrecht Petzoldt and Jintai Ding. Efficient key recovery for all HFE signature variants. In *CRYPTO 2021(1)*, LNCS vol. 12825, pp. 70–93. Springer, 2021.
- [88] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In *PKC 2012*, LNCS vol. 7293, pp. 156–171. Springer, 2012.
- [89] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. In *IEEE Transactions on Information Theory*, Vol. 32, pp. 54–62, 1986.
- [90] Michael J. Wiener. The full cost of cryptanalytic attacks. In *Journal of Cryptology*, Vol. 17, pp. 105–124, 2004.
- [91] Christopher Wolf and Bart Preneel. Equivalent keys in Multivariate Quadratic public key systems. In *Journal of Mathematical Cryptology*, vol. 4, no. 4, pp. 375–415, 2011.
- [92] Xilinx, Inc. XMP100: cost-optimized portfolio product selection guide, 2.1 edition, November 2022. Available at <https://docs.amd.com/v/u/en-US/cost-optimized-product-selection-guide>, last accessed on August 20, 2026.
- [93] Bo-Yin Yang, Chia-Hsin Owen Chen, Daniel J. Bernstein and Jiun-Ming Chen. Analysis of QUAD. In *FSE 2007*, LNCS vol. 4593, pp. 290–308. Springer, 2007.

A The salt-UOV and Its EUF-CMA Security

In this appendix we review the salt-UOV scheme and its security argument.

The salt-UOV was proposed in 2011 [84] and is very close to our recommended UOV depicted in Section 3. It turns out that salt-UOV is *less efficient* than our recommended UOV. In regard to security, it can be shown that the EUF-CMA security of salt-UOV is readily based on the hardness of the UOV problem, an intermediate problem in the MQ realm that is firmly related to UOV. All the state-of-the-art attacks against our recommended UOV are applicable to salt-UOV, and vice versa.

The salt-UOV scheme. The salt-UOV is very similar to our recommended UOV depicted in Section 3, and the *only* difference lies in the design of the signing algorithm. In the signing algorithm of salt-UOV, it first picks (and fixes) a random vinegar vector $\mathbf{v} \in \mathbb{F}_q^v$, and chooses multiple salts uniformly and independently, until the system $\mathcal{F} \left(\begin{bmatrix} \mathbf{v} \\ \cdot \end{bmatrix} \right) = \text{Hash}(\mu \parallel \text{salt})$ of linear equations is solvable. Please refer to Figure 4 for the full detail of salt-UOV.

UOV problem and UOV assumption. We first describe the UOV problem and its associated UOV assumption.

Definition 2 (UOV problem). The UOV problem is parameterized by $\text{params} = (n, m, q)$. Its input is $(\text{params}, \mathcal{P}, \mathbf{t})$, where the target vector $\mathbf{t} \leftarrow \mathbb{F}_q^m$ is uniformly sampled, $\mathcal{P} = \mathcal{F} \circ \mathcal{T} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ is a multivariate quadratic map, $\mathcal{F} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^m$ is a set of m OV-polynomials chosen uniformly at random, and $\mathcal{T} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^n$ is an invertible linear transformation chosen uniformly at random. It asks to find a preimage $\mathbf{s} \in \mathbb{F}_q^n$ such that $\mathcal{P}(\mathbf{s}) = \mathbf{t}$.

The associated *UOV assumption* states that for every (even quantum) efficient algorithm, its success probability in solving the UOV problem is negligibly small.

Security proof of salt-UOV. The relation between the UOV problem and the salt-UOV scheme is summarized by the following theorem.

Theorem 1 ([84]). *Let the hash function $\text{Hash} : \{0, 1\}^* \rightarrow \mathbb{F}_q^m$ in salt-UOV be modeled as a random oracle. Assume there exists an attacking algorithm A , that runs in time $t = t(\lambda)$ and, after making $q_h = \text{poly}_1(\lambda)$ hash queries and $q_s = \text{poly}_2(\lambda)$ signing queries, wins in the EUF-CMA game of salt-UOV with probability $\varepsilon = \varepsilon(\lambda)$. Then we can construct an algorithm $B = B^A$ that runs in time $t' = t'(\lambda)$ and solves the UOV problem with probability $\varepsilon' = \varepsilon'(\lambda)$, where*

$$t' \leq t + (q_s + q_h + 1) \cdot (T + \mathcal{O}(1)), \quad \varepsilon' \geq \varepsilon \cdot \frac{1 - (q_h + q_s) \cdot q_s \cdot 2^{-\text{salt_len}}}{q_s + q_h + 1},$$

and $T = T(\lambda) = \text{poly}_3(\lambda)$ denotes the running time of the evaluation operation associated with the UOV function. $\square$

```

KeyGen(params = (n, m, q, salt_len)):
1: Choose OV-polynomials  $f^{(1)}(x_1, \dots, x_n), \dots, f^{(m)}(x_1, \dots, x_n)$  uniformly at random
2:  $\mathcal{F} := (f^{(1)}, \dots, f^{(m)})$ 
3: Choose an invertible linear transformation  $\mathcal{T} : \mathbb{F}_q^n \rightarrow \mathbb{F}_q^n$  uniformly at random
4:  $\mathcal{P} := \mathcal{F} \circ \mathcal{T}$ 
5:  $\text{pk} := \mathcal{P}$ 
6:  $\text{sk} := (\mathcal{F}, \mathcal{T})$ 
7: return (pk, sk)

Sign(params, sk = ( $\mathcal{F}, \mathcal{T}$ ),  $\mu \in \{0, 1\}^*$ ):
1:  $\mathbf{v} \leftarrow \mathbb{F}_q^{n-m}$ 
2: repeat                                     ▷ Beginning of rejection-sampling phase
3:    $\text{salt} \leftarrow \{0, 1\}^{\text{salt\_len}}$ 
4:    $\mathbf{t} \leftarrow \text{Hash}(\mu \parallel \text{salt})$                                      ▷  $\text{Hash} : \{0, 1\}^* \rightarrow \mathbb{F}_q^m$ 
5:    $\Delta_{\mathbf{t}} := \left\{ \begin{bmatrix} \mathbf{v} \\ \mathbf{u} \end{bmatrix} \in \mathbb{F}_q^n \mid \mathcal{F} \left( \begin{bmatrix} \mathbf{v} \\ \mathbf{u} \end{bmatrix} \right) = \mathbf{t} \right\}$                                      ▷  $\mathbf{u} \in \mathbb{F}_q^m$ 
6: until  $\Delta_{\mathbf{t}} \neq \emptyset$                                      ▷ End of rejection-sampling phase
7:  $\mathbf{w} \leftarrow \Delta_{\mathbf{t}}$ 
8:  $\mathbf{s} := \mathcal{T}^{-1}(\mathbf{w})$ 
9:  $\sigma := (\mathbf{s}, \text{salt})$                                      ▷  $\sigma \in \mathbb{F}_q^n \times \{0, 1\}^{\text{salt\_len}}$ 
10: return  $\sigma$ 

Verify(params, pk =  $\mathcal{P}$ , ( $\mu, \sigma = (\mathbf{s}, \text{salt})$ )):
1:  $\mathbf{t} \leftarrow \text{Hash}(\mu \parallel \text{salt})$ 
2:  $\mathbf{t}' := \mathcal{P}(\mathbf{s})$ 
3: return ( $\mathbf{t} == \mathbf{t}'$ )

```

Figure 4: The key generation, signing and verification algorithms of salt-UOV.