UOV: Unbalanced Oil and Vinegar

Algorithm Specifications and Supporting Documentation
Version 2.0

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

February 5, 2025

Changelog (Round 1 → Round 2)

Changes to the UOV scheme. There is only one change of UOV from Round 1 to Round 2:

- In the round-1 submission of UOV, $\mathsf{seed}_{\mathsf{sk}}$ and $\mathsf{seed}_{\mathsf{pk}}$ were drawn uniformly and *independently*. Instead, $\mathsf{seed}_{\mathsf{pk}}$ is now derived pseudo-randomly from $\mathsf{seed}_{\mathsf{sk}}$. This is achieved by adjusting $\mathsf{UOV.ExpandSK}$ to output both $\mathbf{O}$ and $\mathsf{seed}_{\mathsf{pk}}$. Due to this change, the size of *compressed* secret key has been reduced by 16 bytes, *i.e.*, from 48 bytes to 32 bytes.

Changes to this document.

- A new optimized implementation with GFNI support has been added in Subsection 5.2, and Table 8 compares its performance with the AVX2 implementation.
- The benchmark numbers in Table 2, Table 6, Table 7, Table 9, and Table 10, along with the result descriptions, have been updated to reflect the latest implementations.

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 23
4.2 Direct Attack 23
4.3 Kipnis-Shamir Attack 24
4.4 Intersection Attack 24
4.5 MinRank Attack 25
4.6 Quantum Attacks 25

5 Implementations and Performance **26**
5.1 Common Implementation Techniques 26
5.2 x86 AVX2 Implementation 28
5.3 Arm Neon Implementation 30
5.4 Arm Cortex-M4 Implementation 32
5.5 FPGA Implementation 33

6 Summary: Advantages and Limitations **36**

References **37**

A The salt-UOV and Its EUF-CMA Security **43**

1 Introduction

This document introduces **U**nbalanced **O**il and **V**inegar (UOV), a digital signature scheme using the hash-and-sign paradigm from a trapdoored multivariate quadratic map. Since its initial proposal in 1999 [35], UOV has successfully withstood more than twenty years of rigorous cryptanalysis, demonstrating its remarkable security robustness and long-term reliability.

UOV excels in time efficiency and signature size. Specifically, when it comes to NIST security level 1 [40], UOV outshines other post-quantum digital signature candidates such as Dilithium [28], Falcon [54], and SPHINCS+ [25] 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` version of UOV is at least twice as fast as Dilithium, Falcon, and SPHINCS+ when it comes to generating signatures on most platforms, 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+, marking a notable advantage in the verification speed.

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` version of UOV has a public key of 272 KB, which is significantly larger than those of Dilithium, Falcon and SPHINCS+. To address this challenge, we have proposed new versions of UOV with smaller keys (*e.g.*, 43 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 [18] and GeMSS [16], 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 [19, 31, 42, 35, 16, 12, 29]. Take the UOV signature scheme [35] as an example. Generally speaking, the secret key in UOV is $(\mathcal{F}, \mathcal{T})$, where $\mathcal{F} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ is a *specific* quadratic map and is usually called *central map* due to its critical role in UOV, and the invertible linear transformation $\mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ is used to "hide" the structure of the central map in the public key; moreover, the associated public key is $\mathcal{P} = \mathcal{F} \circ \mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ that consists of a set of multivariate *quadratic* polynomials, *i.e.*,

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

where

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

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 "high" probability.

Given the public/secret key pair $(\mathcal{P}, (\mathcal{F}, \mathcal{T}))$, it is straightforward to build the original UOV scheme [35] 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 versions, *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 sets of recommended parameters 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 versions 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 associated sets of recommended parameters of 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 sets of recommended parameters 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	112	44	256	278 432	237 896	43 576	32	128
`uov-Is`	1	160	64	16	412 160	348 704	66 576	32	96
`uov-III`	3	184	72	256	1 225 440	1 044 320	189 232	32	200
`uov-V`	5	244	96	256	2 869 440	2 436 704	446 992	32	260

§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 estimation of UOV matches the experimental results consistently.

We present in Section 4 those well-known attacks against UOV, including the collision attack, the direct attack, the Kipnis-Shamir attack [36], the intersection attack [8], as well as an improved MinRank attack [59]. Taking all those attacks under consideration, we choose the foregoing four sets of recommended parameters presented in Table 1 so that they can achieve the desired NIST security levels in practice, respectively.

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 executions each.

	Haswell			Skylake		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify
`uov-Ip-classic`	3 242 676	115 420	102 680	2 857 092	109 328	80 342
`uov-Ip-pkc`	3 223 128		321 100	2 789 330		235 006
`uov-Ip-pkc+skc`	3 276 280	2 231 296		2 763 234	1 828 230	
`uov-Is-classic`	5 917 200	141 528	67 104	4 836 380	128 972	60 916
`uov-Is-pkc`	5 555 436		406 224	4 948 572		282 842
`uov-Is-pkc+skc`	5 734 164	3 583 860		4 893 010	2 763 582	
`uov-III-classic`	22 520 100	336 240	314 184	17 438 370	302 728	282 514
`uov-III-pkc`	22 151 812		1 345 384	17 728 338		963 800
`uov-III-pkc+skc`	20 485 784	11 634 928		16 397 898	10 000 580	
`uov-V-classic`	62 045 248	656 104	619 956	46 725 606	591 144	530 468
`uov-V-pkc`	60 507 344		2 865 736	46 434 404		2 017 472
`uov-V-pkc+skc`	53 558 376	27 570 816		42 985 134	22 963 090	
Dilithium 2† [28]	97 621*	281 078*	108 711*	70 548	194 892	72 633
Falcon-512 [44]	19 189 801*	792 360*	103 281*	26 604 000	948 132	81 036
SPHINCS+‡ [25]	1 334 220	33 651 546	2 150 290	1 510 712*	50 084 397*	2 254 495*

† Security level II. ‡ Sphincs+-SHA2-128f-simple. * Data from SUPERCOP [20].

§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 [7] for the comprehensive and detailed information regarding our implementations.

First come several implementation techniques shared among all platforms introduced in Section 5. Then we present our optimization for x86-64 platforms, *i.e.*, the reference platform in NIST PQC standardization [41]. 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 $\{1, 2, ..., k\}$. 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* in essence, 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 $(\cdot)^\mathsf{T}$ denotes the matrix transposition operation; in particular, $\mathbf{0}_k$ denotes the k-dimensional zero vector $[0, 0, ..., 0]^\mathsf{T}$, 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 $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ usually consists of three probabilistic polynomial-time algorithms:

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda, \mathsf{params})$. 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.
- $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$. 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.
- $b := \mathsf{Verify}(\mathsf{pk}, \mu, \sigma)$. Verify is the *deterministic* verification algorithm that, on input the public key pk and the message/signature pair (μ, σ), outputs $b \in \{\mathsf{True}, \mathsf{False}\}$, indicating whether it accepts the signature σ as a valid signature on μ for the public key pk (*i.e.*, $b = \mathsf{True}$) or not (*i.e.*, $b = \mathsf{False}$).

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

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

where the probability is taken over the randomness of the key generation and signing algorithms, and $\mathsf{negl}(\lambda)$ denotes a function that is negligible in the security parameter λ. Moreover, the *standard* security definition [26] 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 $\mathsf{Sign}(\mathsf{sk}, \mu)$ (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 [19, 31, 42, 35, 16, 12, 29]. However, the linearization equations attack proposed by Jacques Patarin [30] 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

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

where

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

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

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

In general, the MQ problem is proven to be NP-complete on every finite field $\mathbb{F}_q$ [62], and the proof is particularly simple and direct when $q = 2$ [23]. Concretely, efficient algorithms are known [38, 67, 68, 63, 64, 38, 65, 66] 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 [73]. Conversely, what interests us most is that 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 [35], as well as its variants, could be traced back to Patarin's linearization equations attack [30] 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 [31] in 1997. After the *balanced* version of this scheme, *i.e.*, $m = n$, was broken by an invariant subspace attack [36] in 1998, Kipnis, Patarin and

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

Sign$\big(\mathsf{sk} = (\mathcal{F}, \mathcal{T}), \mu \in \{0,1\}^*\big)$:
1: $\mathbf{t} \leftarrow \mathsf{Hash}(\mu)$ ▷ $\mathsf{Hash} : \{0,1\}^* \to \mathbb{F}_q^m$
2: Find a random preimage $\mathbf{u} \in \mathbb{F}_q^n$ of $\mathbf{t}$ such that $\mathcal{F}(\mathbf{u}) = \mathbf{t}$
3: $\mathbf{s} := \mathcal{T}^{-1}(\mathbf{u})$
4: $\sigma := \mathbf{s}$
5: **return** σ

Verify$\big(\mathsf{pk} = \mathcal{P}, \mu, \sigma = \mathbf{s} \in \{0,1\}^* \times \mathbb{F}_q^n\big)$:
1: $\mathbf{t} \leftarrow \mathsf{Hash}(\mu)$
2: $\mathbf{t}' := \mathcal{P}(\sigma)$
3: **return** $(\mathbf{t} == \mathbf{t}')$

Figure 1: The key generation, signing, and verification algorithms of the original UOV [35].

Goubin proposed the Unbalanced Oil and Vinegar (UOV) digital signature scheme [35] in 1999, where $m < n$. The simplicity in the UOV design, and the fact no fundamental improvement on attacks against UOV has been made after more than twenty years of cryptanalysis gives us the confidence in the security of the UOV digital signature scheme.

Original UOV. When the multivariate quadratic map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ 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 $(\mathcal{P}, \mathsf{td})$, 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 $(\mathsf{pk} = \mathcal{P}, \mathsf{sk} = \mathsf{td})$.
- For the signing algorithm, given the secret key $\mathsf{sk} = \mathsf{td}$ as well as a message $\mu \in \{0,1\}^*$ to be signed, it first computes $\mathbf{t} \leftarrow \mathsf{Hash}(\mu)$, 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 $\mathsf{Hash} : \{0,1\}^* \to \mathbb{F}_q^m$ denotes a hash function.
- For the verification algorithm, given the public key $\mathsf{pk} = \mathcal{P}$ 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 $\mathbf{t}' = \mathsf{Hash}(\mu)$ holds; otherwise, it returns False, indicating that (μ, σ) is deemed an invalid message/signature pair.

Similar to FDH [13], its security clearly is firmly connected to the design of the key pair $(\mathcal{P}, \mathsf{td})$ as well as the choice of parameters. For instance, in the original UOV digital signature scheme [35] depicted in Figure 1, we have $n > 2m$, $\mathsf{td} = (\mathcal{F}, \mathcal{T})$, and the public key $\mathcal{P}$ is the composite of the maps $\mathcal{F}$ and $\mathcal{T}$, where:

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

$f^{(k)}$ is in the form of

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

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

- $\mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ 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 $\mathcal{F} : \mathbb{F}_q^n \to \mathbb{F}_q^m$, every fixed vector $\mathbf{u} \in \mathbb{F}_q^v$ induces 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 [14]. Similarly, in the central map $\mathcal{F}$, every OV-polynomial $f^{(k)}$ depicted previously is *homogeneous* in essence, 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 [31], 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 $\mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^n$, 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 [28, 54, 25]. 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 versions of UOV, as the following summarizes.

- In the `classic` version 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` version 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` version of UOV, the doubly-compressed version `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 versions are *interoperable*, in the sense that a signature produced by the signing algorithm of one version can be verified by the verification algorithm of the other versions.

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 versions of UOV. After discussing the data layout of UOV in Section 3.3, we conclude in Section 3.4 by proposing four sets of recommended parameters for UOV specified in Table 4. Jumping ahead, the $12 = 3 \times 4$ UOV instances implemented in Section 5 correspond precisely to the three UOV versions 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 [35].

About the trapdoor. Recall that 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 $\left[\mathcal{T}^{-1}(\mathbf{e}_{v+1}), ..., \mathcal{T}^{-1}(\mathbf{e}_n)\right] \in \mathbb{F}_q^{n\times m}$ that is often referred to as the *oil space* in the literature, *i.e.*, $O = \mathsf{span}\left(\mathcal{T}^{-1}(\mathbf{e}_{v+1}), ..., \mathcal{T}^{-1}(\mathbf{e}_n)\right)$. 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 subspaces of $\mathbb{F}_q^n$ could be seen as the column spaces 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 $\mathbf{O}\in\mathbb{F}_q^{v\times m}$ [34]. 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 $p_1,...,p_m$ that vanish on the oil space O determined by $\mathbf{O}$. Nevertheless, inspired by the design of the CyclicRainbow scheme [42, 43], 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.

Recall that 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 $p_i(\mathbf{x})=\mathbf{x}^\mathsf{T}\mathbf{P}_i\mathbf{x}$. 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

$$\begin{bmatrix}\mathbf{O}^\mathsf{T} & \mathbf{I}_m\end{bmatrix}\mathbf{P}_i\begin{bmatrix}\mathbf{O}\\ \mathbf{I}_m\end{bmatrix}=\mathbf{O}^\mathsf{T}\mathbf{P}_i^{(1)}\mathbf{O}+\mathbf{O}^\mathsf{T}\mathbf{P}_i^{(2)}+\mathbf{P}_i^{(3)}\in\mathbb{F}_q^{m\times m}$$

is skew-symmetric.

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 $\mathcal{P}=(p_1,...,p_m)$, 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

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

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

About the inversion operation. For the multivariate quadratic map $\mathcal{P}=(p_1,...,p_m)$ 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

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

where $y_i = \mathbf{v}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{v}$ is the evaluation of p_i at $\overline{\mathbf{v}} = \begin{bmatrix}\mathbf{v}\\ \mathbf{0}_m\end{bmatrix}$, and $\mathbf{S}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O} + \mathbf{P}_i^{(2)} \in \mathbb{F}_q^{v\times m}$. Let $\mathbf{L} \in \mathbb{F}_q^{m\times m}$ be the square matrix with the i-th row being $\mathbf{v}^{\mathsf{T}}\mathbf{S}_i$, and $\mathbf{y} = \mathcal{P}(\overline{\mathbf{v}}) = [y_i]_{i\in[m]} \in \mathbb{F}_q^m$. 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 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 $\mathsf{seed}_{\mathsf{pk}}$ that is used to expand the bulk of the public key deterministically via the PRNG $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ (to be defined) as follows:

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

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

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

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

$$\mathsf{params} = (n, m, q, \mathsf{salt\_len}, \mathsf{pk\_seed\_len}, \mathsf{sk\_seed\_len}),$$

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 $\mathbb{F}_q$;
- $\mathsf{salt\_len}$ denotes the bit length of a binary string $\mathsf{salt} \in \{0,1\}^{\mathsf{salt\_len}}$;
- $\mathsf{pk\_seed\_len}$ denotes the bit length of the binary string $\mathsf{seed}_{\mathsf{pk}} \in \{0,1\}^{\mathsf{pk\_seed\_len}}$ used to expand (the bulk of) the public key; and
- $\mathsf{sk\_seed\_len}$ denotes the bit length of the binary string $\mathsf{seed}_{\mathsf{sk}} \in \{0,1\}^{\mathsf{sk\_seed\_len}}$ used to expand the secret key.

For the recommended parameter sets proposed in this submission, we always have $q \in \{16, 256\}$, $\mathsf{salt\_len} = 128$, $\mathsf{pk\_seed\_len} = 128$, and $\mathsf{sk\_seed\_len} = 256$. 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, as the following indicates.

- $\mathsf{Hash} : (\mu, \mathsf{salt}) \mapsto \mathbf{t}$. Given a message $\mu \in \{0,1\}^*$ and a $\mathsf{salt} \in \{0,1\}^{\mathsf{salt\_len}}$, it maps to the target vector $\mathbf{t} \in \mathbb{F}_q^m$.
- $\mathsf{Expand}_{\mathbf{v}} : (\mu, \mathsf{salt}, \mathsf{seed}_{\mathsf{sk}}, \mathsf{ctr}) \mapsto \mathbf{v}$. Given a message μ, a salt, the $\mathsf{seed}_{\mathsf{sk}} \in \{0,1\}^{\mathsf{sk\_seed\_len}}$ for the secret key and a counter $\mathsf{ctr} \in \{0,1\}^8$, it samples a vinegar vector $\mathbf{v} \in \mathbb{F}_q^v$.
- $\mathsf{Expand}_{\mathsf{sk}} : \mathsf{seed}_{\mathsf{sk}} \mapsto (\mathsf{seed}_{\mathsf{pk}}, \mathbf{O})$. It expands the $\mathsf{seed}_{\mathsf{sk}}$ for the secret key to the $\mathsf{seed}_{\mathsf{pk}} \in \{0,1\}^{\mathsf{pk\_seed\_len}}$ for the public key, as well as the matrix $\mathbf{O} \in \mathbb{F}_q^{v \times m}$ in column-major order.
- $\mathsf{Expand}_{\mathbf{P}} : \mathsf{seed}_{\mathsf{pk}} \mapsto (\{\mathbf{P}_i^{(1)}\}_{i \in [m]}, \{\mathbf{P}_i^{(2)}\}_{i \in [m]})$. It expands the $\mathsf{seed}_{\mathsf{pk}}$ 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, $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ is instantiated with `aes128ctr` using the seed as the key and a zero nonce.

For the first three ones $\mathsf{Hash}(\cdot)$, $\mathsf{Expand}_{\mathbf{v}}(\cdot)$, and $\mathsf{Expand}_{\mathsf{sk}}(\cdot)$, their performances are not critical, and hence we instantiate them with `shake256` [22].

For the last PRNG $\mathsf{Expand}_{\mathbf{P}}(\cdot)$, its performance 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 $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ with `aes128` [21] because this results in much faster implementations. Note that we do not require $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ 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} [37] 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. As we shall see, these procedures serve as the building blocks of three versions of UOV.

- UOV.CompactKeyGen: $\mathsf{params} \mapsto (\mathsf{cpk}, \mathsf{csk})$. Given the parameter params, it outputs a key pair $(\mathsf{cpk}, \mathsf{csk})$, where cpk and csk are compact representations of a UOV public key and its associated secret key, respectively.
- UOV.ExpandSK: $\mathsf{csk} \mapsto \mathsf{esk}$. 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 *invertible* in nature.
- UOV.ExpandPK:$\mathsf{cpk} \mapsto \mathsf{epk}$. 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 *deterministic* in nature as well.
- UOV.Sign:$(\mathsf{esk}, \mu) \mapsto \sigma$. 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:$(\mathsf{epk}, \mu, \sigma) \mapsto b \in \{\mathsf{True}, \mathsf{False}\}$. It takes as input an expanded public key epk, a message/signature pair (μ, σ), and outputs either $b = \mathsf{True}$ or $b = \mathsf{False}$ according to the given message/signature pair (μ, σ) is deemed valid or invalid, respectively.

As specified in the Changelog, we have updated in this submission the generation of the $\mathsf{seed_{pk}}$ that corresponds to the bulk of the public key, mostly driven by the design of MAYO [74] and [75].

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

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

Furthermore, to determine the m multivariate quadratic polynomials $p_1, \ldots, p_m$ 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 $\mathsf{seed_{pk}} \in \{0,1\}^{\mathsf{pk\_seed\_len}}$ using the PRNG $\mathsf{Expand_{P}}(\cdot)$, and

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

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

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

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

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

Signature generation procedure UOV.Sign(·). Given the message $\mu \in \{0,1\}^*$ to be signed, this *randomized* procedure first computes the hash digest $\mathbf{t} = \mathsf{Hash}(\mu \| \mathsf{salt})$, where $\mathsf{salt} \leftarrow \{0,1\}^{\mathsf{salt\_len}}$ 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 $\mathbf{v} := \mathsf{Expand_{v}}(\mu \| \mathsf{salt} \| \mathsf{seed_{sk}} \| \mathsf{ctr}) \in \mathbb{F}_q^v$ from the message μ, the salt, the $\mathsf{seed_{sk}}$ 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 $\mathbf{y} = \mathcal{P}(\overline{\mathbf{v}})$ is the evaluation of $\mathcal{P}$ at $\overline{\mathbf{v}} = \begin{bmatrix}\mathbf{v}\\ \mathbf{0}_m\end{bmatrix}$. If $\mathbf{L}$ is invertible, we can efficiently find a unique solution $\mathbf{x}$, and hence a valid signature $\sigma = (\mathbf{s}, \mathsf{salt})$ of the incoming message μ; otherwise, we just increment the counter ctr and jump to the next round.

UOV.CompactKeyGen(params):

1: $\mathsf{seed_{sk}} \leftarrow \{0,1\}^{\mathsf{sk\_seed\_len}}$
2: $(\mathsf{seed_{pk}}, \mathbf{O}) := \mathsf{Expand_{sk}}(\mathsf{seed_{sk}})$ ▷ $\mathbf{O} \in \mathbb{F}_q^{v \times m}$
3: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand_P}(\mathsf{seed_{pk}})$ ▷ $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{v \times v}$ upper-triangular
4: **for** $i = 1$ upto m **do** ▷ $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{v \times m}$
5: $\quad \mathbf{P}_i^{(3)} := \mathsf{Upper}(-\mathbf{O}^\mathsf{T}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^\mathsf{T}\mathbf{P}_i^{(2)})$
6: $\mathsf{cpk} := (\mathsf{seed_{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]})$
7: $\mathsf{csk} := \mathsf{seed_{sk}}$
8: **return** $(\mathsf{cpk}, \mathsf{csk})$.

UOV.ExpandSK(csk): ▷ $\mathsf{csk} = \mathsf{seed_{sk}}$

1: $(\mathsf{seed_{pk}}, \mathbf{O}) := \mathsf{Expand_{sk}}(\mathsf{seed_{sk}})$
2: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand_P}(\mathsf{seed_{pk}})$
3: **for** $i = 1$ upto m **do**
4: $\quad \mathbf{S}_i := \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}\right)\mathbf{O} + \mathbf{P}_i^{(2)}$
5: $\mathsf{esk} := \left(\mathsf{seed_{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right)$
6: **return** esk.

UOV.ExpandPK(cpk): ▷ $\mathsf{cpk} = \left(\mathsf{seed_{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]}\right)$

1: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand_P}(\mathsf{seed_{pk}})$
2: **for** $i = 1$ upto m **do**
3: $\quad \mathbf{P}_i := \begin{bmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ \mathbf{0} & \mathbf{P}_i^{(3)} \end{bmatrix}$
$\quad \mathsf{epk} := \{\mathbf{P}_i\}_{i\in[m]}$
4: **return** epk.

UOV.Sign(esk, μ): ▷ $\mathsf{esk} = \left(\mathsf{seed_{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right)$

1: $\mathsf{salt} \leftarrow \{0,1\}^{\mathsf{salt\_len}}$
2: $\mathbf{t} \leftarrow \mathsf{Hash}(\mu \| \mathsf{salt})$ ▷ $\mathbf{t} \in \mathbb{F}_q^m$
3: **for** $\mathsf{ctr} = 0$ upto 255 **do**
4: $\quad \mathbf{v} := \mathsf{Expand_v}(\mu \| \mathsf{salt} \| \mathsf{seed_{sk}} \| \mathsf{ctr})$ ▷ $\mathbf{v} \in \mathbb{F}_q^v$
5: $\quad \mathbf{L} := \mathbf{0}_{m\times m}$
6: $\quad$ **for** $i = 1$ upto m **do**
7: $\quad\quad$ Set i-th row of $\mathbf{L}$ to $\mathbf{v}^\mathsf{T}\mathbf{S}_i$.
8: $\quad$ **if** $\mathbf{L}$ is invertible **then**
9: $\quad\quad \mathbf{y} \leftarrow \left[\mathbf{v}^\mathsf{T}\mathbf{P}_i^{(1)}\mathbf{v}\right]_{i\in[m]}$
10: $\quad\quad$ Solve $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$ for $\mathbf{x}$
11: $\quad\quad \mathbf{s} := \begin{bmatrix} \mathbf{v} \\ \mathbf{0}_m \end{bmatrix} + \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix} \cdot \mathbf{x}$ ▷ $\mathbf{s} \in \mathbb{F}_q^n$
12: $\quad\quad \sigma := (\mathbf{s}, \mathsf{salt})$
13: $\quad\quad$ **return** σ
14: **return** $\perp$.

UOV.Verify(epk, $(\mu, \sigma = (\mathbf{s}, \mathsf{salt}))$): ▷ $\mathsf{epk} = \{\mathbf{P}_i\}_{i\in[m]}$

1: $\mathbf{t} \leftarrow \mathsf{Hash}(\mu \| \mathsf{salt})$
2: **return** $\left(\mathbf{t} == \left[\mathbf{s}^\mathsf{T}\mathbf{P}_i\mathbf{s}\right]_{i\in[m]}\right)$.

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

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

Verification procedure UOV.Verify(·). Given the expanded public key epk as well as the message/signature pair (μ, σ) where $\sigma = (\mathbf{s}, \mathsf{salt})$, this procedure recomputes the salted hash digest $\mathbf{t} = \mathsf{Hash}(\mu\|\mathsf{salt})$, 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 versions

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, versions of UOV, so as to accommodate different practical needs. It should be noted that these three versions are essentially the same, except with different representations of public/secret key pair; in particular, a signature generated with one version 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 versions `classic`, `pkc` and `pkc+skc` of our UOV digital signature scheme are summarized as follows.

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

The key generation, signing, and verification algorithms of these three versions 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 versions.

UOV versions	key pair	public key compressed	secret key compressed
`classic`	$(\mathsf{epk}, \mathsf{esk})$	×	×
`pkc`	$(\mathsf{cpk}, \mathsf{esk})$	✓	×
`pkc+skc`	$(\mathsf{cpk}, \mathsf{csk})$	✓	✓

UOV.`classic`.KeyGen(params):

1: $\mathsf{seed}_{\mathsf{sk}} \leftarrow \{0,1\}^{\mathsf{sk\_seed\_len}}$
2: $(\mathsf{seed}_{\mathsf{pk}}, \mathbf{O}) := \mathsf{Expand}_{\mathsf{sk}}(\mathsf{seed}_{\mathsf{sk}})$
3: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$
4: **for** $i = 1$ upto m **do**
5: $\quad \mathbf{P}_i^{(3)} := \mathsf{Upper}(-\mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(2)})$
6: $\quad \mathbf{P}_i := \begin{bmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ \mathbf{0} & \mathbf{P}_i^{(3)} \end{bmatrix}$
7: $\quad \mathbf{S}_i := \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}\right)\mathbf{O} + \mathbf{P}_i^{(2)}$
8: $\mathsf{esk} := \left(\mathsf{seed}_{\mathsf{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right)$
9: $\mathsf{epk} := \{\mathbf{P}_i\}_{i\in[m]}$
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: $\mathsf{seed}_{\mathsf{sk}} \leftarrow \{0,1\}^{\mathsf{sk\_seed\_len}}$
2: $(\mathsf{seed}_{\mathsf{pk}}, \mathbf{O}) := \mathsf{Expand}_{\mathsf{sk}}(\mathsf{seed}_{\mathsf{sk}})$
3: $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$
4: **for** i from 1 to m **do**
5: $\quad \mathbf{P}_i^{(3)} := \mathsf{Upper}(-\mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(2)})$
6: $\quad \mathbf{S}_i := \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}\right)\mathbf{O} + \mathbf{P}_i^{(2)}$
7: $\mathsf{cpk} := (\mathsf{seed}_{\mathsf{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]})$
8: $\mathsf{esk} := \left(\mathsf{seed}_{\mathsf{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right)$
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` versions 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 $\mathbf{M}_\mathbf{S}$ in column-major order, where

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

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. Recall that in the *expanded* public key $\mathsf{epk} = \{\mathbf{P}_i\}_{i\in[m]}$, 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

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

Encoding of esk. Recall that in the *expanded* secret key esk, we have

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

The bit string for esk is the concatenation of the binary representation of $\mathsf{seed}_{\mathsf{sk}}$, 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

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

Encodings of cpk and csk. Recall that in the *compressed* public/secret key pair $(\mathsf{cpk}, \mathsf{csk})$,

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

The bit string cpk is the concatenation of the binary representation of $\mathsf{seed}_{\mathsf{pk}}$ 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 $\mathsf{seed}_{\mathsf{sk}}$.

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

$$|\mathsf{cpk}| = \frac{1}{8} \cdot \left(\mathsf{pk\_seed\_len} + \frac{1}{2} \cdot m^2(m+1) \cdot \log q\right), \text{ and } |\mathsf{csk}| = \frac{1}{8} \cdot \mathsf{sk\_seed\_len}.$$

Encoding of the signature. A UOV signature $\sigma = (\mathbf{s}, \mathsf{salt})$ consists of a vector $\mathbf{s} \in \mathbb{F}_q^n$ and a binary string $\mathsf{salt} \in \{0,1\}^{\mathsf{salt\_len}}$. 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

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

3.4 Recommended Parameter Sets

To accommodate different security needs, we propose four sets of recommended parameters for UOV. These four sets of recommended parameters, 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 versions. 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	|epk| (bytes)	|esk| (bytes)	|cpk| (bytes)	|csk| (bytes)	$\|\sigma\|$ (bytes)
uov-Ip	1	112	44	256	278 432	237 896	43 576	32	128
uov-Is	1	160	64	16	412 160	348 704	66 576	32	96
uov-III	3	184	72	256	1 225 440	1 044 320	189 232	32	200
uov-V	5	244	96	256	2 869 440	2 436 704	446 992	32	260

First, it should be stressed that each set of recommended parameters in Table 4 is applicable to *every* UOV version presented in Section 3.2.4. Moreover, recall that among all four sets of recommended parameters, 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 sets of recommended parameters: `uov-Ip`, which works over $\mathbb{F}_{256}$ and gets slightly smaller keys, and `uov-Is`, which works over $\mathbb{F}_{16}$ and has shorter signatures;
- For NIST security level 3, we propose one set of recommended parameters, *i.e.*, `uov-III`;
- Finally, we propose one set of parameters, *i.e.*, `uov-V`, for NIST security level 5.

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

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

4 Concrete Security Analysis

In this section we present the state-of-the-art attacks against UOV scheme, and analyze the concrete security of the four sets of recommended parameters proposed in Section 3.4. Also we present in Table 5 lower bounds for the bit-complexity of the state-of-the-art attacks against UOV, and clarify how these lower bounds in Table 5 are obtained.

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 analys for UOV is usually carried out by listing all the known critical attacks against UOV that may influence its concrete hardness estimation result. Our confidence in the security of UOV lies in the facts that UOV remains secure after more than twenty years of cryptanalysis, and that there is a solid theoretical foundation on the concrete hardness estimation of practical attacks against MPKC such that the theoretical hardness estimation of UOV matches the experimental results consistently.

Historically, those attacks against UOV are usually classified into two types:

- The *key-recovery attacks* that aim to recover the secret key from the given public key, *e.g.*, the Kipnis-Shamir attack [36], the intersection attack [8] and the MinRank attack [46, 59, 49].

- 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 $\mathsf{Hash}(\cdot)$ is usually modeled as a random oracle (RO) [13].

EUF-CMA security of UOV. UOV can be designed to achieve the EUF-CMA security requirement, *e.g.*, the salt-UOV proposed in 2011 [32]. As can be seen from Appendix A, the EUF-CMA 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.

- It is easy to see our recommended UOV is *more efficient* than salt-UOV in terms of the signing speed.

Table 5: Bit-complexity estimates (lower bound for the base-2 logarithm of the number of binary gates required to perform an attack) of state-of-the-art attacks against our proposed parameter sets. The Kipnis-Shamir and intersection attacks are key-recovery attacks, and the collision and direct attacks are forgery attacks.

Parameter set	Collision	Direct		KS	Intersection	
(n, m, q)	$\log_2$	k	$\log_2$	$\log_2$	k	$\log_2$
`uov-Ip` $(112, 44, 256)$	191	2	**145**	218	2	166
`uov-Is` $(160, 64, 16)$	**143**	12	165	154	3	176
`uov-III` $(184, 72, 256)$	303	4	**218**	348	2	250
`uov-V` $(244, 96, 256)$	399	6	**278**	445	2	312

4.1 Collision Attack

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

Computing hashes. For the sake of concreteness, we say that the cost of a Keccak-f 1600 permutation is $2^{17.5}$ bit operations [48], so computing the list of hashes takes at least $Y \cdot 2^{17.5}$ bit operations.

Concrete Hardness The lowest conceivable bit-cost of the total attack is then

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

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 most optimal known evaluation for MQ with m equations over $\mathbb{F}_{2^r}$: Gray-code enumeration [6]. 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.*, $\mathbf{s}_i$ and salt_j such that the first m' elements of $\mathcal{P}(\mathbf{s}_i)$ matches the first $m' \log q$ bits of $\mathsf{Hash}(\mu\|\mathsf{salt}_j)$). 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 $2^{10.7}\sqrt{q^m mr}$, 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 cost[2], 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 [76] resulting in a $\sqrt{q}\times$ speedup. It is clear on the other hand that whatever rational value we place on this large-memory penalty, the total cost will be at least 2^{143}.

- Realistically, an attacker would use a memoryless collision-finding algorithm such (*e.g.*, in [51]). However, algorithms like [51] 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 $2^{k/2}$).

4.2 Direct Attack

The most straightforward attack against UOV, (and even against most of the MPKC cryptosystems) is the direct attack, where the attacker 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 $\mathsf{salt}^* \in \{0,1\}^{\mathsf{salt\_len}}$ on his will, computes

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

[2] Bernstein *et al.* [60]: "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'."

$\mathbf{t} = \mathsf{Hash}(\mu^* \| \mathsf{salt}^*)$, 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 is to solve a random system of m quadratic equations in n variables; and the state-of-the-art approach is to first take advantage of the underdeterminedness of the system by reducing to the problem of solving a system of $m' = m - 1$ equations in $n' = m - 1$ variables with the approach of Thomae and Wolf [50], and then using the hybrid WiedemannXL algorithm [72] to solve the new system. The estimated cost of this state-of-the-art approach is

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

and is *identified* as the cost of the direct attack against UOV. Here, $d_{N,M}$ is the *operating degree* of XL, and is defined to be 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.

Note that the attacker might compute $\mathsf{Hash}(\mu \| \mathsf{salt})$ for a large number of message/salt pairs, and then solve a multi-target version of the system-solving problem. Nevertheless, our foregoing estimation 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 [36, 69] tries to recover the subspace O from the public map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$. 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 [31]. 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 [8]).

With this in mind, in this submission the cost of Kipnis-Shamir attack is identified as

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

which we believe is an underestimate of the cost of the attack for our proposed parameters.

4.4 Intersection Attack

The intersection attack [8] generalizes the ideas behind the Kipnis-Shamir attack, in combination with a system-solving approach such as in the reconciliation attack [61]. It tries to simultaneously find k linearly independent vectors in $O = \left\{\mathbf{u} \in \mathbb{F}_q^n \,\middle|\, \mathcal{P}(\mathbf{u}) = \mathbf{0}_m\right\}$, 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 $n < \frac{2k-1}{k-1}m$. Please refer to [8] for the full detail.

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

4.5 MinRank Attack

In the *MinRank attack*, the attacker tries to find a linear combination of the public polynomials of minimal rank [56, 57, 9, 49]. And the *MinRank problem* can be formulated as: given the m matrices $\mathbf{P}_1, ..., \mathbf{P}_m \in \mathbb{F}_q^{n\times n}$ representing the quadratic polynomials $p_1, ..., p_m$ in the public key $\mathcal{P}$, find a linear combination $\mathbf{Q} = \sum c_i \cdot \mathbf{P}_i$ with rank no more than r. Historically, there exist many different approaches to solve the MinRank problem, including the linera algebra approach, the Kipnis-Shamir method [70], the Minors Modeling method [71], and the Support Minors Modeling method [59].

Let $\mathbf{A}_i$ be the submatrix of $\mathbf{P}_i$ which consists of the last $k = \lceil \frac{n-m}{m} \rceil$ rows. Then $\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \vdots \\ \mathbf{A}_m \end{bmatrix} \in \mathbb{F}_q^{km\times n}$ has rank no more than $n - m$. In this way, the MinRank attacker can be used to recover the secret key of UOV.

Although this attack works for UOV scheme, in regard to our four sets of recommended parameters, its cost estimate is much larger than those of the other attackers.

4.6 Quantum Attacks

All the known quantum attacks against UOV are obtained by speeding some part of a classical attack up with Grover's algorithm. Therefore, they outperform the classical attacks by at most a square root factor, and they do not threaten our security claims. Indeed, the NIST security levels 1,3, and 5 are defined with respect to the hardness of a key search against a block cipher such as the AES with 128, 192, and 256-bit keys respectively. Grover speeds up a key search by almost a square root factor, so, for a quantum attack to break the NIST security targets it needs to improve on classical attacks by more than a square root factor, which is not possible by relying on Grover's algorithm alone.

5 Implementations and Performance

Recall that in Section 3 we have presented *three* UOV versions as well as *four* sets of recommended parameters. This section specifies the implementations of these $12 = 3 \times 4$ UOV instances over various platforms, as well as their performance, so as to fully demonstrate the strengths of our UOV digital signature in practice. Please refer to [7] 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 all platforms under consideration.

5.1.1 Solving linear equations

Recall that the UOV signing algorithm needs to solve the system of linear equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ 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 [47, 17], 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 [7].

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, which we deem is sufficiently small. 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 [19] and, thus, the same techniques apply. We make use of a technique first introduced by Chou, Kannwischer, and Yang [17]: 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

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 $\perp$

$\mathbf{A}' := [\mathbf{A} \mid \mathbf{b}] \in \mathbb{F}_q^{m \times (m+1)}$ ▷ $\mathbf{A}' = [a'_{i,j}]$
for $i = 0$ upto $m - 1$ **do**
  for $j = i + 1$ upto $m - 1$ **do**
    if $a'_{i,i} == 0$ **then**
      for $k = i$ upto m **do**
        $a'_{i,k} := a'_{i,k} + a'_{j,k}$
  if $a'_{i,i} == 0$ **then**
    return $\perp$
  $p_i^{-1} := (a'_{i,i})^{-1}$
  for $k = i$ upto m **do**
    $a'_{i,k} := p_i^{-1} \cdot a'_{i,k}$
  for $j = i + 1$ upto $m - 1$ **do**
    for $k = i$ upto m **do**
      $a'_{j,k} := a'_{j,k} + a'_{j,i} \cdot a'_{i,k}$
for $i = m - 1$ downto 1 **do**
  for $j = 0$ upto $i - 1$ **do**
    $a'_{j,m} := a'_{j,m} + a'_{j,i} \cdot a'_{i,m}$
return last column of $\mathbf{A}'$

end and, hence, vastly reducing the number of required multiplications. This results in a substantial speed-up. In the case of $\mathbb{F}_{16}$, we use 15 accumulators: one for each possible value of $s_i s_j$ except for zero as those columns can be discarded straight away. In the case of $\mathbb{F}_{256}$, we use 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 vastly reducing the number of required costly field multiplications. Note that this approach results in signature-dependent memory access patterns which may be problematic in case signatures are secret and 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 [17], 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 $s_i s_j$ is 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 $\mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$. Althought the straightforward 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

Table 6: Benchmarking results of AVX2 implementations. Numbers are the median CPU cycles of 1000 executions each.

	Haswell			Skylake		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify
`uov-Ip-classic`	3 242 676	115 420	102 680	2 857 092	109 328	80 342
`uov-Ip-pkc`	3 223 128		321 100	2 789 330		235 006
`uov-Ip-pkc+skc`	3 276 280	2 231 296		2 763 234	1 828 230	
`uov-Is-classic`	5 917 200	141 528	67 104	4 836 380	128 972	60 916
`uov-Is-pkc`	5 555 436		406 224	4 948 572		282 842
`uov-Is-pkc+skc`	5 734 164	3 583 860		4 893 010	2 763 582	
Dilithium 2† [28]	97 621*	281 078*	108 711*	70 548	194 892	72 633
Falcon-512 [44]	19 189 801*	792 360*	103 281*	26 604 000	948 132	81 036
SPHINCS+‡ [25]	1 334 220	33 651 546	2 150 290	1 510 712*	50 084 397*	2 254 495*
`uov-III-classic`	22 520 100	336 240	314 184	17 438 370	302 728	282 514
`uov-III-pkc`	22 151 812		1 345 384	17 728 338		963 800
`uov-III-pkc+skc`	20 485 784	11 634 928		16 397 898	10 000 580	
`uov-V-classic`	62 045 248	656 104	619 956	46 725 606	591 144	530 468
`uov-V-pkc`	60 507 344		2 865 736	46 434 404		2 017 472
`uov-V-pkc+skc`	53 558 376	27 570 816		42 985 134	22 963 090	

† Security level II. ‡ Sphincs+-SHA2-128f-simple. * Data from SUPERCOP [20].

`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 $q = 16$.

5.2 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 [41]. 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 $\mathsf{Hash}(\cdot)$, $\mathsf{Expand}_{\mathbf{v}}(\cdot)$, $\mathsf{Expand}_{\mathsf{sk}}(\cdot)$, and $\mathsf{Expand}_{\mathbf{P}}(\cdot)$, we call the OpenSSL library when relating to standard cryptographic primitives, *e.g.*, `shake256` and `aes128`. Specifically, for the instantiation of $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ with round-reduced AES, we adapt the `aes128ctr` implementation in [24], 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 [77] instruction set as one of our supported architectures. The GFNI instruction set offers native arithmetic instructions for field multiplication, significantly enhancing

Table 7: Benchmarking results of AVX2 implementations using 4-round AES for public-key expansion. Numbers are median CPU cycles of 1000 executions.

	Haswell			Skylake		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify
`uov-Ip-pkc`	3 068 424	115 384	204 304	2 704 742	108 772	146 234
`uov-Ip-pkc+skc`	3 105 224	2 128 032		2 681 140	1 739 984	
`uov-Is-pkc`	5 816 588	141 092	212 668	4 632 826	128 494	153 210
`uov-Is-pkc+skc`	5 553 688	3 372 588		4 433 596	2 600 024	
`uov-III-pkc`	21 892 612	334 304	791 636	17 011 172	303 270	553 182
`uov-III-pkc+skc`	20 134 272	11 567 888		15 987 562	9 627 218	
`uov-V-pkc`	58 967 004	669 800	1 540 068	45 542 444	588 118	1 107 108
`uov-V-pkc+skc`	52 285 484	26 766 088		42 173 848	22 040 178	

Table 8: Comparisons between AVX2 and GFNI implementations. Numbers are the median CPU cycles of 1000 executions each.

	AVX2 on Sapphire Rapids			GFNI on Sapphire Rapids		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify
`uov-Ip-classic`	2 487 484	77 506	73 358	1 369 930	54 090	86 990
`uov-Ip-pkc`	2 472 242		160 086	1 353 750		160 986
`uov-Ip-pkc+skc`	2 485 266	1 473 552		1 350 776	1 004 064	
`uov-Is-classic`	4 415 672	83 114	54 936	4 417 746	83 216	51 482
`uov-Is-pkc`	4 389 066		167 868	4 365 356		169 336
`uov-Is-pkc+skc`	4 406 518	2 245 916		4 364 576	2 226 750	
`uov-III-classic`	15 035 550	270 462	239 322	7 814 890	172 698	222 992
`uov-III-pkc`	14 937 048		640 266	7 693 778		636 358
`uov-III-pkc+skc`	13 888 072	8 099 638		7 057 630	5 044 940	
`uov-V-classic`	39 891 484	526 664	423 164	18 745 178	413 920	420 184
`uov-V-pkc`	39 754 340		1 362 146	18 276 498		1 330 336
`uov-V-pkc+skc`	35 316 226	19 198 672		15 913 010	8 426 464	

performance compared to the standard AVX2 implementation, which relies on table lookup techniques for field multiplication.

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 Ubuntu `clang` version `18.1.3 (1ubuntu1)` 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 Intel Xeon Platinum 8488C (Sapphire Rapids) 2.1 GHz in AWS EC2, with Turbo Boost and Hyper-Threading disabled, using `gcc` version `13.3.0`.

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` versions as well as those for the verification algorithms of the `pkc` and `pkc+skc` versions,

indicating that they share the same implementations. It can be seen from Table 6 that

- `uov-Ip` has the fastest signing speed;
- `uov-Is` has the fastest verification speed although its public key is larger than `uov-Ip`. This 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 $\mathsf{Expand}_{\mathbf{P}}(\cdot)$, *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 around 40%;
- For the signing algorithm of the `pkc+skc` version, the main computation is spent on expanding the compressed public/secret keys.
- For signing and key generation algorithms that involve intensive $\mathbb{F}_{256}$ multiplications, native support for field multiplication significantly boosts performance.

5.3 Arm Neon Implementation

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

Table 9: Benchmarking results of our Neon implementations. Numbers are median CPU cycles of 1000 executions.

	Cortex-A72			Apple M1		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify
`uov-Ip-classic`	11 184 344	242 940	146 685	1 666 693	60 551	49 257
`uov-Ip-pkc`	11 131 516		4 108 895	1 646 625		112 557
`uov-Ip-pkc+skc`	11 136 554	7 862 072		1 658 789	1 060 675	
`uov-Is-classic`	28 639 886	532 316	157 286	3 353 769	92 166	46 366
`uov-Is-pkc`	28 367 829		5 655 593	3 329 499		138 842
`uov-Is-pkc+skc`	26 434 420	16 471 394		3 349 631	2 095 931	
Dilithium 2† [10]	269 724	649 230	272 824	71 061	224 125	69 792
Falcon-512 [39]	—	1 044 600	59 900	—	459 200	22 700
`uov-III-classic`	70 654 623	1 634 516	663 105	9 573 288	185 700	184 615
`uov-III-pkc`	71 626 169		19 179 875	9 500 736		458 693
`uov-III-pkc+skc`	66 149 624	44 498 639		9 515 409	6 259 053	
`uov-V-classic`	318 633 305	3 625 405	1 458 819	28 228 123	393 047	371 702
`uov-V-pkc`	318 012 194		43 930 901	28 143 066		1 006 448
`uov-V-pkc+skc`	324 840 210	113 198 818		26 668 781	16 051 661	

† Security level II.

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

In regard to the instantiation of $\mathsf{Expand}_{\mathbf{P}}(\cdot)$, 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 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],

Table 10: Benchmarking results of Neon implementations using 4-round AES for public-key expansion. Numbers are median CPU cycles of 1000 executions.

	Cortex-A72			Apple M1		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify
`uov-Ip-pkc`	8 919 141	253 248	1 906 022	1 616 346	60 606	82 587
`uov-Ip-pkc+skc`	8 685 240	5 668 785		1 622 300	1 030 357	
`uov-Is-pkc`	25 109 983	548 013	2 599 386	3 290 909	92 210	97 835
`uov-Is-pkc+skc`	23 202 073	13 227 111		3 313 696	2 051 666	
`uov-III-pkc`	60 927 075	1 675 403	9 242 424	9 376 139	185 666	326 898
`uov-III-pkc+skc`	57 810 441	37 228 110		9 350 857	6 136 255	
`uov-V-pkc`	297 219 084	3 374 750	21 091 068	27 900 801	383 246	700 126
`uov-V-pkc+skc`	302 614 530	91 100 712		26 463 880	15 739 151	

which runs four parallelized 32-bit bitsliced instances, to the Neon instruction set, since Biesheuvel [11] 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 [5]) running at 1.8 GHz without Arm AES instructions. The source code is compiled with `gcc (Debian 12.2.0-14) 12.2.0`. The Macbook has an Apple M1 CPU running at 3.2 GHz with Arm AES instruction support. Its compiler is Apple `clang` version `16.0.0 (clang-1600.0.26.6)`.

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 results when $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ is instantiated with round-reduced AES. 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 44 or 45 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.
- 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` versions, where $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ dominates the computation time.
- For `pkc` and `pkc+skc` versions, the symmetric primitives play an important role in the performance. By comparing the performance impact of key compressed versions to the `classic` version, 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.
- The 4-round AES makes for an efficient $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ function such that the verification time of `pkc` version is of the same order as other PQC schemes on Apple M1.

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)		
	version	KeyGen	Sign	Verify
`uov-Ip` (This work)	`classic`	138 757k	2 522k	995k
	`pkc`	174 978k		11 548k (10 716k)
	`pkc+skc`	174 978k	88 810k	
`uov-Is` (This work)	`classic`	195 793k	2 478k	616k
	`pkc`	203 363k		16 043k (15 168k)
	`pkc+skc`	296 199k	113 509k	
Dilithium-2 [2]		1 598k	4 083k	1 572k
Falcon-512 [44, 45]		163 994k	39 014k	473k
sphincs-sha256-128f-simple [45]		16 112k	400 443k	22 548k
sphincs-sha256-128s-simple [45]		1 031 755k	7 848 131k	7 711k

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 the cycles required for flashing. This table contains the cycles required for flashing and the total key generation cycles.

	version	key generation w/o flashing (cc)	flashing (cc)	key generation w/ flashing (cc)
`uov-Is`	`classic`	195 793k	202 293k	398 086k
	`pkc`	203 363k	110 741k	314 104k
	`pkc+skc`	296 199k	18 284k	314 483k

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

Symmetric primitives. For the instantiations of $\mathsf{Hash}(\cdot)$, $\mathsf{Expand}_{\mathbf{v}}(\cdot)$, and $\mathsf{Expand}_{\mathsf{sk}}(\cdot)$, we use `shake256` as implemented in pqm4 [45] which integrates the Keccak permutation in `Armv7-M` assembly from the XKCP [53]. For the instantiation of $\mathsf{Expand}_{\mathbf{P}}(\cdot)$, we use the t-table AES implementation by Schwabe and Stoffelen [33]. 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.

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

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 [45] 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

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

		speed (clock cycles)		
	version	KeyGen	Sign	Verify
`uov-Ip`	`pkc`	169 238k	2 548k	5 801k
	`pkc+skc`	169 239k	83 072k	
`uov-Is`	`pkc`	194 917k	2 478k	7 592k
	`pkc+skc`	287 753k	105 083k	

previously proposed by Chen and Chou for Classic McEliece [15]. 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.

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.

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

5.5 FPGA Implementation

In this subsection, we present our field-programmable gate array (FPGA) design for the UOV instances and report their performances on popular platforms. Since our design supports multiple UOV instances, we adopt a processor design that provides a custom instruction set dedicated for 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.

Target platform. We test our design on two Xilinx Artix-7 platforms: Zynq-7000™ Z-7020 and Artix-7 XC7A200T. We target Artix-7 as it is the hardware target platform recommended by NIST [4] for the PQC standardization effort. Consequently, other PQC schemes have also been implemented on Artix-7 allowing comparison to our implementation. Although we report our results with a setting tailoring for the Artix-7 platforms, it can be easily adapted to other parameter sets and ported to other FPGAs.

Since we use a processor design for performing UOV in hardware, 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/to the computation modules; and (3) the modules for performing actual computations.

Results. We evaluate the FPGA design by measuring the resource utilization and cycle counts for key generation, signing, and verification. All of the designs are synthesized

Table 14: The FPGA results with full-round AES for our low-area (no pipelined AES) design.

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

Table 15: Results of UOV with 4-round AES for our low-area design. The resource information is the same as that of full-round AES.

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

and done implementation with Xilinx Vivado 2022.1 edition. The designs for `uov-Ip` and `uov-Is` are evaluated on Xilinx Zynq-7000 Z-7020 and `uov-III` and `uov-V` are evaluated on XC7A200T. We set the target frequency to 100MHz for both.

We report the resource utilization for UOV with non-pipelined AES and the cycle counts in full-round AES mode in Table 14. The utilization of LUTs and Slices of the versions with the same security level are similar, except `uov-Is` and `uov-V-pkc+skc`. Their requirements for key storage exceed the limit of the BRAM on their target boards, resulting in an increase in LUTs. The utilization of BRAMs is close to what we expect, whereas the utilization of DSP and FF resources is low.

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

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

The signing cycle count in the `pkc+skc` version is dominated by the $\mathsf{Expand}_{\mathsf{sk}}(\cdot)$ function, specifically, the calculation of the $\mathbf{S}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O} + \mathbf{P}_i^{(2)}$. This calculation takes $(n-m)\cdot m\cdot(n-m+15)$ cycles, where the 15 includes flow control and other operations such as loading from and storing to temporary storage. In the case of `uov-Ip-pkc+skc`,

Table 16: The performance results using pipelined AES.

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

$\mathsf{Expand}_{\mathsf{sk}}(\cdot)$ takes 248 336 cycles. The remaining computation includes 7 515 cycles for tasks such as Gaussian elimination and polynomial evaluation, and 189 618 cycles for expanding $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$ from $\mathsf{seed}_{\mathsf{pk}}$. In the end, with savings from overlapping these computations, it results in 78 262 cycles in `uov-Ip-pkc+skc`.

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

We also report the cycle counts when using a 4-round AES for $\mathsf{Expand}_{\mathbf{P}}(\cdot)$ in Table 15. It shows a reduction in cycles for verification in the `pkc` version and signing in the `pkc+skc` version. The saving for verification matches our expectation, which can be estimated by the difference in rounds multiplied by the number of calls to the AES module. It is $(8 \cdot 44 \cdot ((112+44)(112-44)/2)/128) \cdot 6 = 87\,516$ cycles in the case of `uov-Ip`. For the signing algorithm of UOV `pkc+skc`, the saving is less significant because computing the $\mathbf{S}_i$'s in the expanded secret key dominates the cycle count.

Finally, we present the results for our high-performance design using a fully pipelined AES in Table 16. We show only the results for `pkc` and `pkc+skc` as only those are majorly affected in signing and verification by the faster AES. Comparing to the results using the no-pipelined AES, verification improves by a factor of 3. As AES now generates one block per cycle, it requires $(8 \cdot 44 \cdot ((112+44)(112-44)/2)/128) = 14\,586$ cycles to generate $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$. The overhead $61\,499 - (14\,586 + 6\,435) = 40\,478$ cycles comes again from waiting for quadratic terms $\mathbf{s}_i^{\mathsf{T}}\mathbf{s}_j$. For the signing algorithm of the `pkc+skc` version, the cycle count slightly improves since the bottleneck is the computation of the $\mathbf{S}_i$. The cycles for 4-round and 10-round AES are similar since both are pipelined, generating 128-bits per cycle.

For the case of `uov-Ip-pkc+skc`, the pipelined versions use 16% and 3% more LUTs than the non-pipelined version for 10- and 4-round AES, respectively.

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:

Efficiency. The signature generation process of UOV consists of simple linear algebra operations such as matrix vector multiplication and solving linear systems over small finite fields. Therefore, the UOV scheme can be implemented very efficiently and is one of the fastest available signature schemes.

Short Signatures. The signatures produced by the UOV signature scheme are only a few hundred bits in size and therefore much shorter than those of RSA and those of other post-quantum signature schemes.

Modest Computational 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. Since its invention in 1999, no efficient attack against UOV has been found. Moreover, despite rigorous cryptanalysis, no fundamental attack improvement against UOV has been developed. 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. Therefore, overall we are confident in the security of the UOV signature scheme.

Simplicity. The design of the UOV schemes is extremely simple. Therefore, it requires only minimum knowledge in algebra to understand and implement the scheme. 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. However, due to increasing memory capabilities even of medium devices (*e.g.*, smartphones), we do not think that this will be a major problem. Furthermore, to mitigate this disadvantage, we propose the `pkc` and `pkc+skc` versions of UOV with smaller keys to accommodate different practical needs.

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 Jan 5, 2025.

[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 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/m/qbGEs3TXGwAJ`. Online February 7, 2019, last accessed on Jan 5, 2025.

[5] Arm Ltd. Arm Cortex-A72 software optimization guide, 2015. Available at `https://developer.arm.com/documentation/uan0016/a/`, last accessed on Jan 5, 2025.

[6] 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.

[7] 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.

[8] Ward Beullens. Improved cryptanalysis of UOV and Rainbow. In *EUROCRYPT 2021(1)*, LNCS vol. 12696, pp. 348–373. Springer, 2021.

[9] Ward Beullens. Breaking Rainbow takes a weekend on a laptop. In *CRYPTO 2022(2)*, LNCS vol. 13508, pp. 464–479. Springer, 2022.

[10] 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 2022(1)*, pp. 224–244, 2022.

[11] Ard Biesheuvel. Accelerated AES for Arm64 linux kernel, 2017. Available at `https://old.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/`, last accessed on Jan 5, 2025.

[12] 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 Jan 5, 2025.

[13] Mihir Bellare and Phillip Rogaway. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *CCS 1993*, ACM, pp. 62–73.

[14] 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.

[15] Ming-Shing Chen and Tung Chou. Classic McEliece on the ARM Cortex-M4. In *IACR TCHES 2021(3)*, pp. 125–148, 2021.

[16] 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 Jan 5, 2025.

[17] Tung Chou, Matthias J. Kannwischer and Bo-Yin Yang. Rainbow on Cortex-M4. In *IACR TCHES, 2021(4)*, pp. 650–675, 2021.

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

[19] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *ACNS 2005*, LNCS vol. 3531, pp. 164–175. Springer, 2005.

[20] Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. Available at `https://bench.cr.yp.to`, last accessed on Jan 5, 2025.

[21] FIPS PUB 197 – Advanced Encryption Standard (AES), 2001. Available at `https://doi.org/10.6028/NIST.FIPS.197`, last accessed on Jan 5, 2025.

[22] 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 Jan 5, 2025.

[23] Michael R. Garey and David S. Johnson. Computers and intractability: A guide to the theory of NP-Completeness. W. H. Freeman, 1979.

[24] 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 Jan 5, 2025.

[25] Andreas Hulsing, 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 Jan 5, 2025.

[26] Jonathan Katz. Digital Signatures. Springer, 2010.

[27] Juliane Krämer, Mirjam Loiero. Fault Attacks on UOV and Rainbow In *COSADE 2019*, LNCS vol. 11421, pp. 193–214. Springer, Heidelberg, April 2019.

[28] Vadim Lyubashevsky, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Peter Schwabe, Gregor Seiler, Damien Stehlé, and Shi Bai. CRYSTALS-DILITHIUM. Technical report, National Institute of Standards and Technology, 2020. Available at `https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions`, last accessed on Jan 5, 2025.

[29] H. Ong, C.P. Scnorr and A. Shamir. 1984. An Efficient Signature Scheme Based on Quadratic Equations. In *STOC 1984*, ACM, pp. 208–216.

[30] 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.

[31] Jacques Patarin. The oil and vinegar signature scheme. Presented at the *Dagstuhl Workshop on Cryptography*, September 1997.

[32] 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.

[33] 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.

[34] Christopher Wolf and Bart Preneel. 2011. Equivalent keys in Multivariate Quadratic public key systems. Journal of Mathematical Cryptology 4.4 (2011): 375–415.

[35] Aviad Kipnis, Jacques Patarin and Louis Goubin. Unbalanced Oil and Vinegar signature schemes. In *EUROCRYPT 1999*, LNCS vol. 1592, pp. 206–222. Springer, 1999.

[36] Aviad Kipnis and Adi Shamir. Cryptanalysis of the Oil and Vinegar signature scheme. In *CRYPTO 1998*, LNCS vol. 1462, pp. 257–266. Springer, 1998.

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

[38] 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.

[39] 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 Jan 5, 2025.

[40] National Institute of Standards and Technology. Submission requirements and evaluation criteria for the post-quantum cryptography standardization process. Available at `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf`, last accessed on Jan 5, 2025.

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

[42] 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.

[43] Albrecht Petzoldt. Efficient key generation for Rainbow. In *PQCrypto 2020*, LNCS vol. 12100, pp. 92–107. Springer, 2020.

[44] Thomas Pornin. New efficient, constant-time implementations of Falcon. IACR Cryptology ePrint Archive, Report 2019/893.

[45] 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 Jan 5, 2025.

[46] 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 Jan 5, 2025.

[47] Kyung-Ah Shim, Sangyub Lee and Namhun Koo. Efficient implementations of Rainbow and UOV using AVX2. In *IACR TCHES 2022(1)*, pp. 245–269, 2021.

[48] Nigel Smart. *'Bristol Fashion' MPC Circuits*. Available at `https://old.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/`, last accessed on Jan 5, 2025.

[49] 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.

[50] Enrico Thomae and Christopher Wolf. Solving underdetermined systems of multivariate quadratic equations revisited. In *PKC 2012*, LNCS vol. 7293, pp. 156–171. Springer, 2012.

[51] 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.

[52] Xilinx, Inc. XMP100: cost-optimized portfolio product selection guide, 2.1 edition, November 2022. Available at `https://docs.xilinx.com/v/u/en-US/cost-optimized-product-selection-guide`.

[53] 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 at January 5, 2025.

[54] 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 Jan 5, 2025.

[55] Oded Goldreich. The Foundations of Cryptography - Volume 1: Basic Techniques. Cambridge University Press 2001, ISBN 0-521-79172-3.

[56] Olivier Billet and Henri Gilbert. Cryptanalysis of Rainbow. In *SCN 2006*, LNCS vol. 4116, pp. 336–347. Springer, 2006.

[57] Louis Goubin and Nicolas T. Courtois. Cryptanalysis of the TTM cryptosystem. In *Asiacrypt 2000*, LNCS vol. 1976, pp. 44–57. Springer, 2000.

[58] Chengdong Tao, Albrecht Petzoldt, and Jintai Ding. Efficient key recovery for all HFE signature variants. In *CRYPTO 2021(I)*, LNCS vol. 12825, pp. 70–93. Springer, 2021.

[59] 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.

[60] 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 Jan 5, 2025.

[61] 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.

[62] Jacques Patarin and Louis Goubin. Trapdoor One-Way Permutations and Multivariate Polynomials. In *ICICS 1997*, LNCS vol. 1334, pp. 356–368. Springer, 1997.

[63] 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.

[64] Johannes Buchmann, Jintai Ding, Mohamed Saied Emam Mohamed and Wael Said Abd Elmageed Mohamed. MutantXL: solving multivariate polynomial equations for cryptanalysis. In *Symmetric Cryptography 2009*.

[65] 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.

[66] 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.

[67] Jean-Charles Faugère. A new efficient algorithm for computing Gröbner bases (F_4). In *Journal of Pure and Applied Algebra*, 139(1–3), pp. 61–88, 1999.

[68] 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.

[69] 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.

[70] 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.

[71] 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 cryptology. In *ISSAC 2010*, pp. 257–264. ACM, 2010.

[72] 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.

[73] Jonathan Katz and Yehuda Lindell. Introduction to modern cryptography, Third Edition CRC Press, 2020.

[74] 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 Jan 5, 2025.

[75] Sophie Schmieg. Comments for round second on ramp candidate UOV. Available at `https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/lCrLRB4RzFU/m/t2WibzZrAAAJ`, last accessed on Jan 5, 2025.

[76] 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/SlBu7_oITN4/m/1MW-OVyfBgAJ`, last accessed on Jan 5, 2025.

[77] 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 Jan 5, 2025.

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 [32] 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 $\mathcal{F}\left(\begin{bmatrix}\mathbf{v}\\ \cdot\end{bmatrix}\right) = \mathsf{Hash}(\mu \| \mathsf{salt})$ 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 $\mathsf{params} = (n, m, q)$. Its input is $(\mathsf{params}, \mathcal{P}, \mathbf{t})$, where the target vector $\mathbf{t} \leftarrow \mathbb{F}_q^m$ is uniformly sampled, $\mathcal{P} = \mathcal{F} \circ \mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ is a multivariate quadratic map, $\mathcal{F} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ is a set of m OV-polynomials chosen uniformly at random, and $\mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ 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 ([32])**.** *Let the hash function* $\mathsf{Hash} : \{0,1\}^* \to \mathbb{F}_q^m$ *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* $q_h = \mathsf{poly}_1(\lambda)$ *hash queries and* $q_s = \mathsf{poly}_2(\lambda)$ *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 solve the UOV problem with probability* $\varepsilon' = \varepsilon'(\lambda)$*, where*

$$t' \leq t + (q_s + q_h + 1) \cdot (T + \mathcal{O}(1)), \qquad \varepsilon' \geq \varepsilon \cdot \frac{1 - (q_h + q_s) \cdot q_s \cdot 2^{-\mathsf{salt\_len}}}{q_s + q_h + 1},$$

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

KeyGen($\mathsf{params} = (n, m, q, \mathsf{salt\_len})$):

1: Choose OV-polynomials $f^{(1)}(x_1, ..., x_n), ..., f^{(m)}(x_1, ..., x_n)$ uniformly at random
2: $\mathcal{F} := (f^{(1)}, ..., f^{(m)})$
3: Choose an invertible linear transformation $\mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ uniformly at random
4: $\mathcal{P} := \mathcal{F} \circ \mathcal{T}$
5: $\mathsf{pk} := \mathcal{P}$
6: $\mathsf{sk} := (\mathcal{F}, \mathcal{T})$
7: **return** $(\mathsf{pk}, \mathsf{sk})$

Sign$\big(\mathsf{params}, \mathsf{sk} = (\mathcal{F}, \mathcal{T}), \mu \in \{0,1\}^*\big)$:

1: $\mathbf{v} \leftarrow \mathbb{F}_q^{n-m}$
2: **repeat** ▷ Beginning of rejection-sampling phase
3: $\quad \mathsf{salt} \leftarrow \{0,1\}^{\mathsf{salt\_len}}$
4: $\quad \mathbf{t} \leftarrow \mathsf{Hash}(\mu \| \mathsf{salt})$ ▷ $\mathsf{Hash} : \{0,1\}^* \to \mathbb{F}_q^m$
5: $\quad \Delta_{\mathbf{t}} := \left\{ \begin{bmatrix} \mathbf{v} \\ \mathbf{u} \end{bmatrix} \in \mathbb{F}_q^n \;\middle|\; \mathcal{F}\left(\begin{bmatrix} \mathbf{v} \\ \mathbf{u} \end{bmatrix} \right) = \mathbf{t} \right\}$ ▷ $\mathbf{u} \in \mathbb{F}_q^m$
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: $\sigma := (\mathbf{s}, \mathsf{salt})$ ▷ $\sigma \in \mathbb{F}_q^n \times \{0,1\}^{\mathsf{salt\_len}}$
10: **return** σ

Verify$(\mathsf{params}, \mathsf{pk} = \mathcal{P}, (\mu, \sigma = (\mathbf{s}, \mathsf{salt})))$:

1: $\mathbf{t} \leftarrow \mathsf{Hash}(\mu \| \mathsf{salt})$
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.