UOV: Unbalanced Oil and Vinegar

Algorithm Specifications and Supporting Documentation
Version 1.0

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

May 30, 2023

Contents

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. First proposed in 1999 [35], UOV has withstood two decades of cryptanalysis, testifying to its enduring security and reliability.

UOV excels in time efficiency and signature size. Particularly at the NIST security level 1, UOV outshines other post-quantum digital signature candidates such as Dilithium [28], Falcon [54], and SPHINCS+ [25] in multiple ways:

- **Signature size**. UOV signatures are more compact, having significantly shorter lengths compared to those produced by Dilithium, Falcon, and SPHINCS+.
- **Signing Speed**. The `classic` variant of UOV is at least a factor 2 faster than 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 the new NIST standards by most measures, except for public key size. At NIST level 1, the `classic` UOV has a public key size of 272 KB, which is significantly larger than the public keys of Dilithium, Falcon, and SPHINCS+. We propose variants of UOV with smaller public keys (e.g., 43 KB at SL 1), at the cost of longer verification times.

In summary, UOV is competitive with the new NIST standards by most measures, except for public key size. For instance, at NIST security level 1, the `classic` variant of UOV has a public key size of 272 KB, which is significantly larger than those of Dilithium, Falcon, and SPHINCS+. We propose variants of UOV with smaller keys (*e.g.*, 43 KB at NIST security level 1), at the cost of longer verification time.

§2: Preliminaries. 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 a 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]. 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; 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.*,

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

where

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

and all coefficients are taken from the finite field $\mathbb{F}_q$. For cryptographic purposes, the central map $\mathcal{F}$ is carefully designed so that carrying out the inversion operation of $\mathcal{P}$ is hard, but is 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 message/signature pair is invalid.

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

§3: Specifications. Section 3 starts with the design rationale behind our UOV digital signature scheme. Then we specify three variants of UOV, *i.e.*, `classic`, `pkc` and `pkc+skc`, which offer various tradeoffs between space efficiency and time efficiency, so as to accommodate a variety of different use-cases. 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 variants in combination with the four recommended parameter sets, and the benchmarking results of their implementations over NIST PQC Reference Platform can be found in Table 2..

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	48	128
`uov-Is`	1	160	64	16	412 160	348 704	66 576	48	96
`uov-III`	3	184	72	256	1 225 440	1 044 320	189 232	48	200
`uov-V`	5	244	96	256	2 869 440	2 436 704	446 992	48	260

§4: Security analysis. On the one hand, researchers have been working on the security proof of UOV for the past decades; for instance, a security proof was proposed in 2011 [32] which connects the security of a UOV variant (*i.e.*, the salt-UOV to be introduced in Appendix A) to the hardness of a less natural problem in MPKC realm (namely, the UOV problem). On the other hand, it should be noted that we do not have a formal security proof which reduces certain commonly accepted "hard" mathematical problem(s), *e.g.*, the MQ problem [23], to the security of UOV. Instead, the security analysis for UOV is usually carried out by looking at all the known attacks against UOV that may influence its concrete hardness. This is in sharp contrast to that of lattice-based cryptosystems. Though lattice-based cryptosystems lay the claims of the provable security, none of the recommended parameters of the lattice-based candidates selected by NIST [1] satisfies the conditions of the provable security, and until now there has been no *solid* theoretical foundation on the concrete hardness estimate of lattice-based ones.

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 CPU 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`	3 311 188	116 624	82 668	2 903 434	105 324	90 336
`uov-Ip-pkc`	3 393 872		311 720	2 858 724		224 006
`uov-Ip-pkc+skc`	3 287 336	2 251 440		2 848 774	1 876 442	
`uov-Is-classic`	4 945 376	123 376	60 832	4 332 050	109 314	58 274
`uov-Is-pkc`	5 002 756		398 596	4 376 338		276 520
`uov-Is-pkc+skc`	5 448 272	3 042 756		4 450 838	2 473 254	
`uov-III-classic`	22 046 680	346 424	275 216	17 603 360	299 316	241 588
`uov-III-pkc`	22 389 144		1 280 160	17 534 058		917 402
`uov-III-pkc+skc`	21 779 704	11 381 092		17 157 802	9 965 110	
`uov-V-classic`	58 162 124	690 752	514 100	48 480 444	591 812	470 886
`uov-V-pkc`	57 315 504		2 842 416	46 656 796		2 032 992
`uov-V-pkc+skc`	57 306 980	26 021 784		45 492 216	22 992 816	
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].

Generally speaking, our confidence in the security of the UOV scheme lies in the following facts: UOV remains secure after more than twenty years 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. It should be stressed that the foregoing four sets of recommended parameters presented in Table 1 are chosen such that they satisfy the required levels of security, respectively.

§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 full details on our implementations.

First, we present our optimization for x86-64 platforms, which is designated as the reference platform in NIST PQC standardization. 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. Specifically, 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, the implementations of UOV for the Arm Cortex-M4, as well as the implementation of UOV on the popular FPGA platforms.

§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 documentation. 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 processing 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 documentation, q always denotes a positive integer, 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 $\mathbf{I}_k$ denote 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:

- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$. KeyGen is the key generation algorithm that, on input the security parameter 1^λ, outputs a public key pk and its associated secret key sk.

- $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, \mu)$. 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 μ 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

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

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 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 a digital signature scheme states that, given a public key pk, and given access to a signing oracle that on input a message μ outputs $\mathsf{Sign}(\mathsf{sk}, \mu)$ (where sk is the secret key corresponding to pk), every computationally bound adversary is unable to come up with a valid signature for a new message μ' that was not given to the signing oracle with not-negligible probability. Please refer to [26] for formal security definitions.

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]. 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 $\mathcal{P}$ a *nonlinear* map from $\mathbb{F}_q^n$ to $\mathbb{F}_q^m$ in essence, and consists of a sequence $p^{(1)}(\mathbf{x}), ..., p^{(m)}(\mathbf{x})$ of multivariate polynomials in n variables $\mathbf{x} = [x_i]_{i\in[n]}$, where n, m, q are public parameters, and $\mathbb{F}_q$ is a finite field with q elements. For cryptographic purposes, $\mathcal{P}$ should be carefully designed so that it works like a trapdoored one-way function: first, it should be easy to carry out the evaluation operation $\mathbf{x} \mapsto \mathcal{P}(\mathbf{x})$ of $\mathcal{P}$ on any input from $\mathbb{F}_q^n$; moreover, given the trapdoor information associated with the public key $\mathcal{P}$, the inversion operation of $\mathcal{P}$ can be carried out *efficiently* in the sense that for the given $\mathbf{t} \in \mathbb{F}_q^m$, we can efficiently find a preimage $\mathbf{s} \in \mathbb{F}_q^n$ such that $\mathcal{P}(\mathbf{s}) = \mathbf{t}$; finally, it should "hard" to do the inversion operation for any (even quantum) efficient adversary without the trapdoor associated with $\mathcal{P}$.

When $m \geq n$, we can construct a public-key encryption scheme based on the map $\mathcal{P}$, which is similar to the "Textbook RSA" encryption scheme: for a given plaintext $\mathbf{s} \in \mathbb{F}_q^n$, its corresponding ciphertext is $\mathbf{t} := \mathcal{P}(\mathbf{s})$. Conversely, what interest us most is that when $m < n$, the multivariate polynomials are well suited to building secure digital signature schemes using hash-and-sign paradigm: for a given message $\mu \in \{0,1\}^*$, its corresponding signature is $\mathbf{s} \in \mathbb{F}_q^n$ such that $\mathcal{P}(\mathbf{s}) = \mathsf{Hash}(\mu)$, where $\mathsf{Hash} : \{0,1\}^* \to \mathbb{F}_q^m$ is a hash function.

In practice, the polynomials in $\mathcal{P}$ are usually quadratic, which explains why MPKCs are often referred to as Multivariate Quadratic (MQ) cryptosystems. In this case, we have

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

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 are taken from $\mathbb{F}_q$. In such cases, the evaluation operation associated with $\mathcal{P}$ is obviously efficient. Moreover, the time complexity of the inversion operation associated with $\mathcal{P}$, as well as the security of the aforementioned public-key cryptosystems, is firmly connected to the hardness of the following NP-hard problem.

Definition 1 (MQ problem)**.** Given $(n, m, q, \mathcal{P})$ where n, m, q are positive integers, and $\mathcal{P}$ denotes a multivariate quadratic map

$$\mathcal{P} = (p^{(1)}, p^{(2)}, ..., p^{(m)}) : \mathbb{F}_q^n \to \mathbb{F}_q^m,$$

find an n-dimensional vector $\mathbf{s} = [s_i]_{i\in[n]} \in \mathbb{F}_q^n$ such that

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

The corresponding MQ assumption states that, for every (even quantum) probabilistic polynomial-time algorithm, its success probability in sampling an element in the set $\mathcal{P}^{-1}(\mathbf{0}_m) = \left\{\mathbf{u} \in \mathbb{F}_q^n \,\middle|\, \mathcal{P}(\mathbf{u}) = \mathbf{0}_m\right\}$ is negligibly small, when only $(n, m, q, \mathcal{P})$ is given.

In the security analysis of multivariate quadratic public-key cryptosystems, the central role played by the MQ problem could be gleaned from the fact that solving an MQ problem is at least as hard as finding a preimage $\mathbf{s}$ in $\mathbb{F}_q^n$ for an arbitrary target vector $\mathbf{t}$ in $\mathbb{F}_q^m$.

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

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

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

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

Furthermore, the general MQ problem is proven to be NP-hard on every finite field $\mathbb{F}_q$, and the proof is particularly simple and direct when $q = 2$ [23]. In particular, the most difficult instances of the MQ problem are generally obtained when m and n are of the same order of magnitude, and efficient algorithms are known [38]when m is either *much* larger than or *much* smaller than n. It is also interesting to note that very often the best known algorithms on the MQ problem have a similar complexity for worse cases and for random cases, which is also true for quantum algorithms. Until now, no efficient quantum algorithm against the MQ problem has been found; taking the NP-hardness of MQ into consideration, it is generally believed that this will still be the case in the future.

2.3 An Overview of UOV

History of UOV. Here we only consider the cases where $m < n$ and the *quadratic* map $\mathcal{P} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ is applied for the construction of digital signature schemes; specifically, what interest us most is the UOV digital signature scheme [35] and its variants. The history of UOV scheme, 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 signature scheme (OV) [31] in 1997. After the *balanced* version of this scheme was broken by an invariant subspace attack [36] in 1998, Kipnis, Patarin and Goubin proposed the Unbalanced Oil and Vinegar (UOV) digital signature scheme [35] in 1999. 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 give us the confidence in the security of the UOV 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 build in the key generation algorithm its 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:

- In the key generation algorithm, given the security parameter, it outputs a random key pair $(\mathsf{pk} = \mathcal{P}, \mathsf{sk} = \mathsf{td})$.
- In 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 compute $\mathbf{t} \leftarrow \mathsf{Hash}(\mu)$, 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.
- In 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 not a valid 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

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

  In the literature, the $n - m$ variables $x_1, ..., x_{n-m}$ in the UOV scheme are called the *vinegar variables*, the remaining m variables $x_{n-m+1}, ..., x_n$ 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 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. 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. It 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 vinegar vector $\mathbf{v} = [v_i]_{i \in [n-m]} \in \mathbb{F}_q^{n-m}$ induces a linear transformation $\eta_{\mathbf{v}} = \mathcal{F}\left(\begin{bmatrix}\mathbf{v}\\ \cdot\end{bmatrix}\right) : \mathbb{F}_q^m \to \mathbb{F}_q^m$; thus, with gaussian elimination we can recover, if possible, a preimage $\begin{bmatrix}\mathbf{v}\\ \mathbf{w}\end{bmatrix} \in \mathbb{F}_q^n$ of $\mathbf{t}$ under $\mathcal{F}$, and hence a preimage $\mathbf{s} \in \mathbb{F}_q^n$ of $\mathbf{t}$ under $\mathcal{P}$, where

$$\mathbf{s} = \mathcal{T}^{-1}\left(\begin{bmatrix}\mathbf{v}\\ \mathbf{0}_n\end{bmatrix} + \begin{bmatrix}\mathbf{0}_{n-m}\\ \mathbf{w}\end{bmatrix}\right) = \mathcal{T}^{-1}\left(\begin{bmatrix}\mathbf{v}\\ \mathbf{0}_n\end{bmatrix}\right) + \mathcal{T}^{-1}\left(\begin{bmatrix}\mathbf{0}_{n-m}\\ \mathbf{w}\end{bmatrix}\right)$$

moreover, the induced linear transformation $\eta_{\mathbf{v}}$ is full-rank with probability approximately $1 - 1/q$. and polynomial number of random attempts on the choice of $\mathbf{v}$ enables us to recover a preimage for a random $\mathbf{t} \in \mathbb{F}_q^m$, except with negligible probability. This shows that $\mathcal{F}$ is indeed efficiently invertible.

Reformulation of trapdoor. For every $i \in [n]$, define the constant vector $\mathbf{e}_i = [\delta_{i,j}]_{j\in[n]} \in \mathbb{F}_q^n$, where δ denotes the Kronecker delta function. Clearly $\mathcal{F}(\mathbf{e}_{n-m+1}) = \cdots = \mathcal{F}(\mathbf{e}_n) = 0$, and the foregoing analysis on the efficient inversion of $\mathcal{P}$ relies the efficient computation of $\mathcal{T}^{-1}\left(\begin{bmatrix}\mathbf{0}_{n-m}\\ \mathbf{w}\end{bmatrix}\right)$ in $O = \mathsf{span}\left(\mathcal{T}^{-1}(\mathbf{e}_{n-m+1}), ..., \mathcal{T}^{-1}(\mathbf{e}_n)\right)$. Here O is the m-dimensional vector space over $\mathbb{F}_q$ which is commonly referred to as the *oil space* in the literature, which could be seen as the column space of the matrix $\left[\mathcal{T}^{-1}(\mathbf{e}_{n-m+1}), ..., \mathcal{T}^{-1}(\mathbf{e}_n)\right] \in \mathbb{F}_q^{n\times m}$. Therefore, the trapdoor information associated with $\mathcal{P}$ could be reformulated 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 most m-dimensional subspaces of $\mathbb{F}_q^n$ could be seen as column spaces of the matrices in the form of $\begin{bmatrix}\mathbf{O}\\ \mathbf{I}_m\end{bmatrix}$, where $\mathbf{O} \in \mathbb{F}_q^{(n-m)\times m}$, we would adopt the convention in this submission that the trapdoor of $\mathcal{P}$ is re-defined as the matrix $\overline{\mathbf{O}} = \begin{bmatrix}\mathbf{O}\\ \mathbf{I}_m\end{bmatrix}$ where $\mathbf{O} \in \mathbb{F}_q^{(n-m)\times m}$, and this does not reduce the key space of UOV much.

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 in Equation 1 is *homogeneous*, and an inhomogeneous generalization on $f^{(k)}$ does not seem to improve the security of UOV significantly.

UOV is *unbalanced* in the sense that we have more vinegar variables than oil variables, which is in sharp contrast to the *original* (balanced) OV scheme [31], where we have the same number of vinegar variables and oil variables.

In UOV, the key lies in the specific design of the central map $\mathcal{F}$, where the oil variables never *mix* with oil variables in every OV-polynomial $f^{(k)}$, and UOV 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 the UOV digital signature scheme in full detail.

First and foremost, we specify the design rationale behind our UOV digital signature scheme in Section 3.1.

Section 3.2 is devoted to the full specification of our UOV digital signature scheme. First come the parameters and the choice of symmetric primitives used in UOV. Then we specify the UOV digital signature scheme itself as a tuple of five algorithms. In addition to the usual **key generation**, **signing**, and **verification** algorithms, we also specify a **secret key expansion** algorithm and a **public key expansion** algorithm. The idea is that the key generation algorithm outputs compact representations of a secret key and a public key, and the keys need to be expanded before they can be used in the signing or verification algorithm respectively. This API gives more flexibility to the end user than the usual 3-part API for signature schemes. For example, if the use case demands that keys are small, then we can store and transmit only the compact representations of the keys, and perform the key expansion as part of the signing or verification algorithm. Alternatively, if the use case demands that signing and/or verification is fast, then the keys can be stored in the expanded representation to avoid having to expand the key during the signing or verification operations. Finally, to comply with the NIST API for digital signatures, and to facilitate a comparison with other signature schemes that only have the traditional 3-part API, we specify three variants of the UOV signature scheme with a 3-part API, as depicted in Figure 3.

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

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

After discussing the data layout of UOV (variants) in Section 3.3, we conclude by proposing four sets of recommended parameters for UOV specified in Table 4 in Section 3.4. The $12 = 3 \times 4$ UOV instances to be implemented in Section 5 correspond precisely to the combinations of the three UOV variants in combination with the four recommended parameter sets.

3.1 Design Rationale Behind UOV

First, we present the design rationale behind the design of our UOV digital signature scheme to be presented in Section 3.2.

The reformulated trapdoor $\overline{\mathbf{O}}$. Recall that in UOV, the matrix $\overline{\mathbf{O}} = \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix}$ now can be considered as the trapdoor of the public key $\mathcal{P}$, and its column space O is usually referred to as the oil space. To sample a random trapdoor $\overline{\mathbf{O}}$ in the key generation algorithm, it suffices to pick an $\mathbf{O} \leftarrow \mathbb{F}_q^{(n-m)\times m}$ uniformly at random; moreover, we can set the secret key to be $\mathbf{O}$, which could decrease the size of the secret key significantly.

Generation of the public key $\mathcal{P}$. As indicated in Figure 1, in the key generation algorithm of the *original* UOV [35], we first sample a random secret key $\mathsf{sk} = (\mathcal{F}, \mathcal{T})$ and then derive its associated public key $\mathsf{pk} = \mathcal{P}$ *deterministically*, as the key generation algorithms in most cryptosystems normally do. Nevertheless, inspired by the design of the CyclicRainbow scheme [42], the process of generating a public key from the secret key could be *partially reversible*, as the following indicates.

In the key generation algorithm, after the new trapdoor $\overline{\mathbf{O}} = \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix}$ 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 every element in the oil space O. Recall that each 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 $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-m)\times(n-m)}$, $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{m\times m}$ are both upper-triangular, and $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-m)\times m}$. 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

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

is skew-symmetric.

Thus, given the trapdoor $\overline{\mathbf{O}}$, 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 $\mathbf{P}_i^{(2)} \leftarrow \mathbb{F}_q^{(n-m)\times m}$ and m *upper triangular* matrices $\mathbf{P}_i^{(1)} \leftarrow \mathbb{F}_q^{(n-m)\times(n-m)}$ uniformly at random (note that this operation is *independent* of $\overline{\mathbf{O}}$ at all); 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.

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* $\mathbf{v} \leftarrow \mathbb{F}_q^{n-m}$, 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 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 is routine to verify that

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

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 $\mathbf{S}_i = (\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O} + \mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-m)\times m}$. 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 $\mathbf{v} \leftarrow \mathbb{F}_q^{n-m}$. 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 $\mathbf{S}_i \in \mathbb{F}_q^{(n-m)\times m}$ depends 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 secret key and compute them only once in the key generation algorithm, which would improve the time efficiency of the signing algorithm.

Use of PRNG. 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).

In UOV, we can first sample a short seed $\mathsf{seed}_{\mathsf{sk}} \leftarrow \{0,1\}^{\mathsf{sk\_seed\_len}}$ uniformly at random, and then carry out the *deterministic* expansion

$$\mathsf{Expand}_{\mathsf{sk}} : \ \mathsf{seed}_{\mathsf{sk}} \longmapsto \mathbf{O}$$

determined by the PRNG $\mathsf{Expand}_{\mathsf{sk}}(\cdot)$, which could reduce the size of the secret key significantly; similarly, we can reduce the size of the public key by first sampling a short seed $\mathsf{seed}_{\mathsf{pk}} \leftarrow \{0,1\}^{\mathsf{pk\_seed\_len}}$ uniformly at random, and then carrying out the *deterministic* expansion

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

determined by the PRNG $\mathsf{Expand}_{\mathbf{P}}(\cdot)$.

3.2 The UOV Digital Signature Scheme

This section is devoted to the introduction to the UOV digital signature scheme.

3.2.1 Parameters in UOV

The UOV digital signature algorithm is parameterized by the following values

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

where

- q denotes the size of a finite field $\mathbb{F}_q$. In this submission, we always have $q \in \{16, 256\}$.
- m denotes the number of multivariate quadratic polynomials in the public key.
- n denotes the number of variables in the multivariate quadratic polynomials in the public key.
- $\mathsf{salt\_len}$ 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 public key.
- $\mathsf{sk\_seed\_len}$ 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.

As we shall see later, for all recommended parameter sets proposed in this submission, we have $\mathsf{salt\_len} = 128$, $\mathsf{pk\_seed\_len} = 128$, and $\mathsf{sk\_seed\_len} = 256$.

3.2.2 Choice of symmetric primitives

Here we define a variety of hash functions and PRNGs that are needed for the specification of our UOV digital signature scheme. There are three functions Hash, $\mathsf{Expand}_{\mathbf{v}}$, and $\mathsf{Expand}_{\mathsf{sk}}$, whose performance is not critical. We instantiate these functions with `shake256` [22]. The remaining function is $\mathsf{Expand}_{\mathsf{pk}}$, whose performance has a high impact on the performance of the overall signature scheme. The input and output of this function are public, so the implementation of this function does not need to be side-channel resistant. We instantiate $\mathsf{Expand}_{\mathsf{pk}}$ with `aes128` [21] because this results in much faster implementations. The precise instantiation of our symmetric primitives is as described below:

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

It maps a message μ and a 16-byte salt to the target vector $\mathbf{t}$. The size of target vector is $m \cdot \log_2 q$ bits. In our implementations $\mathsf{Hash}(\cdot)$ is instantiated with `shake256(·)`.

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

It samples a vinegar vector $\mathbf{v}$ based on the message μ, a 16-byte salt, the secret seed $\mathsf{seed}_{\mathsf{sk}}$, and a 1-byte counter. The output size is $(n-m) \cdot \log_2 q$ bits. In our implementations $\mathsf{Expand}_{\mathbf{v}}(\cdot)$ is instantiated with `shake256(·)` as well.

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

It expands the seed for the secret key to the matrix $\mathbf{O}$. The output size is $(n-m) \cdot m \cdot \log_2 q$ bits. In our implementations, $\mathsf{Expand}_{\mathsf{sk}}(\cdot)$ is instantiated with `shake256(·)`, and we sample the matrix in column-major order as it is required in key generation and signing algorithms.

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

It expands the 16-byte public seed to the matrices $\{\mathbf{P}_i^{(1)}\}_{i\in[m]}$ and $\{\mathbf{P}_i^{(2)}\}_{i\in[m]}$. We first sample the $\{\mathbf{P}_i^{(1)}\}_{i\in[m]}$ matrices, and then the $\{\mathbf{P}_i^{(2)}\}_{i\in[m]}$ matrices. The m matrices are expanded in an interleaved fashion, in column-major order. That is, we start by sampling the (0,0) entry of $\mathbf{P}_1^{(1)}$, followed by the (0,0) entry of $\mathbf{P}_2^{(1)}$, etc. After sampling the (0,0) entry of the last matrix $\mathbf{P}_m^{(1)}$ we continue with the (1,0) entries, followed by the (1,1) entries and proceeding column by column, *i.e.*, in lexicographic order. The size of $\{\mathbf{P}_i^{(1)}\}_{i\in[m]}$ is $m \cdot \frac{(n-m)(n-m+1)}{2} \cdot \log_2 q$ bits. The size of $\{\mathbf{P}_i^{(2)}\}_{i\in[m]}$ is $m \cdot m \cdot (n-m) \cdot \log_2 q$ bits. We implement $\mathsf{Expand}_{\mathbf{P}}$ using `aes128ctr` using the seed as the key and a zero nonce. If the `aes128ctr` API allows passing a custom counter value, this allows sampling at arbitrary output positions which allows for some optimizations.

Note that we do not require $\mathsf{Expand}_{\mathbf{P}}$ to be a cryptographically secure stream cipher. We (optionally) propose to use `aes128ctr` reduced to 4 (instead of 10) rounds. 4-round `aes128` has been proven to have a maximal differential probability of 2^{-114} [37] which is deemed sufficient for the purpose of public-key expansion in UOV.

3.2.3 Functionalities in UOV

We now specify the five functionalities/algorithms in our UOV signature scheme, and their pseudocode can be found in Figure 2.

- $\mathsf{UOV.CompactKeyGen}$: $1^\lambda \to (\mathsf{csk}, \mathsf{cpk})$. Given the security parameter 1^λ, 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.
- $\mathsf{UOV.ExpandSK}$:$\mathsf{csk} \mapsto \mathsf{esk}$. 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* in nature.

- $\mathsf{UOV.ExpandPK}{:}\mathsf{cpk} \mapsto \mathsf{epk}$. 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.
- $\mathsf{UOV.Sign}{:}(\mathsf{esk}, \mu) \to \sigma$. It takes an expanded secret key esk, a message $\mu \in \{0,1\}^*$ to be signed, and outputs a signature σ of μ.
- $\mathsf{UOV.Verify}{:}(\mathsf{epk}, (\mu, \sigma)) \mapsto \{\mathsf{True}, \mathsf{False}\}$. It takes as input an expanded public key epk, a message/signature pair (μ, σ), and outputs True or False if the given message/signature pair (μ, σ) is deemed valid or invalid, respectively.

Compact key generation. This functionality is to generate $(\mathsf{cpk}, \mathsf{csk})$, the compact representations of a public/secret key pair.

It first samples two seeds, $\mathsf{seed}_{\mathsf{pk}}$ and $\mathsf{seed}_{\mathsf{sk}}$, uniformly at random. And the compact secret key csk is defined to be $\mathsf{csk} = (\mathsf{seed}_{\mathsf{pk}}, \mathsf{seed}_{\mathsf{sk}})$.

Then we define the oil space corresponding to the matrix $\overline{\mathbf{O}} = \begin{bmatrix} \mathbf{O} \\ \mathbf{I}_m \end{bmatrix}$ by expanding the matrix $\mathbf{O}$, where $\mathbf{O} \in \mathbb{F}_q^{m \times (n-m)}$ is obtained by expanding the random seed $\mathsf{seed}_{\mathsf{sk}} \in \{0,1\}^{\mathsf{sk\_seed\_len}}$ using the PRNG $\mathsf{Expand}_{\mathsf{sk}}(\cdot)$.

Furthermore, to determine the m multivariate quadratic polynomials $p_1, \ldots, p_m$ in 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 $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-m)\times(n-m)}, \mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-m)\times m}$ are both obtained by expanding the random seed $\mathsf{seed}_{\mathsf{pk}} \in \{0,1\}^{\mathsf{pk\_seed\_len}}$ using the PRNG $\mathsf{Expand}_{\mathbf{P}}$, and

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

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

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

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

These matrices will be used in the $\mathsf{UOV.Sign}$ functionality. Finally, $\mathsf{UOV.ExpandSK}(\mathsf{csk})$ outputs the *expanded* representation of the secret key $\mathsf{esk} = \left(\mathsf{seed}_{\mathsf{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right)$.

Signature generation. Given the message $\mu \in \{0,1\}^*$ to be signed, the signature generation algorithm 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 $\mathbf{v} \leftarrow \mathsf{Expand}_{\mathbf{v}}(\mu \| \mathsf{salt} \| \mathsf{seed}_{\mathsf{sk}} \| \mathsf{ctr}) \in \mathbb{F}_q^{n-m}$ from the message μ, the salt salt, $\mathsf{seed}_{\mathsf{sk}}$ 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 $\mathcal{P}\left(\begin{bmatrix} \mathbf{v} \\ \mathbf{0}_m \end{bmatrix} + \overline{\mathbf{O}} \cdot \mathbf{x}\right) = \mathbf{t}$ by solving the system of linear equations $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$, where

```
UOV.CompactKeyGen():
    $\mathsf{seed_{sk}} \leftarrow \{0,1\}^{\mathsf{sk\_seed\_len}}$
    $\mathsf{seed_{pk}} \leftarrow \{0,1\}^{\mathsf{pk\_seed\_len}}$
    $\mathbf{O} := \mathsf{Expand_{sk}}(\mathsf{seed_{sk}})$    ▷ $\mathbf{O} \in \mathbb{F}_q^{(n-m)\times m}$
    $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed_{pk}})$    ▷ $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-m)\times(n-m)}$ upper triangular
    for $i = 1$ upto $m$ do    ▷ $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-m)\times m}$
        $\mathbf{P}_i^{(3)} := \mathsf{Upper}(-\mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(2)})$
    $\mathsf{cpk} := (\mathsf{seed_{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]})$
    $\mathsf{csk} := (\mathsf{seed_{pk}}, \mathsf{seed_{sk}})$
    return $(\mathsf{cpk}, \mathsf{csk})$.

UOV.ExpandSK(csk):    ▷ $\mathsf{csk} = (\mathsf{seed_{pk}}, \mathsf{seed_{sk}})$
    $\mathbf{O} := \mathsf{Expand_{sk}}(\mathsf{seed_{sk}})$
    $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed_{pk}})$
    for $i = 1$ upto $m$ do
        $\mathbf{S}_i := \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}\right)\mathbf{O} + \mathbf{P}_i^{(2)}$
    $\mathsf{esk} := \left(\mathsf{seed_{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right)$
    return esk.

UOV.Sign($\mathsf{esk} = \left(\mathsf{seed_{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right), \mu$):
    $\mathsf{salt} \leftarrow \{0,1\}^{\mathsf{salt\_len}}$
    $\mathbf{t} \leftarrow \mathsf{Hash}(\mu \| \mathsf{salt})$    ▷ $\mathbf{t} \in \mathbb{F}_q^m$.
    for ctr $= 0$ upto $255$ do
        $\mathbf{v} := \mathsf{Expand}_{\mathbf{v}}(\mu \| \mathsf{salt} \| \mathsf{seed_{sk}} \| \mathsf{ctr})$    ▷ $\mathbf{v} \in \mathbb{F}_q^{n-m}$.
        $\mathbf{L} := \mathbf{0}_{m\times m}$
        for $i = 1$ upto $m$ do
            Set $i$-th row of $\mathbf{L}$ to $\mathbf{v}^{\mathsf{T}}\mathbf{S}_i$.
        if $\mathbf{L}$ is invertible then
            $\mathbf{y} \leftarrow \left[\mathbf{v}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{v}\right]_{i\in[m]}$
            Solve $\mathbf{L}\mathbf{x} = \mathbf{t} - \mathbf{y}$ for $\mathbf{x}$
            $\mathbf{s} := \begin{bmatrix}\mathbf{v}\\ \mathbf{0}_m\end{bmatrix} + \overline{\mathbf{O}} \cdot \mathbf{x}$    ▷ $\mathbf{s} \in \mathbb{F}_q^n$.
            $\sigma := (\mathbf{s}, \mathsf{salt})$
            return $\sigma$
    return $\perp$.

UOV.ExpandPK(cpk):    ▷ $\mathsf{cpk} = \left(\mathsf{seed_{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]}\right)$
    $\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed_{pk}})$
    for $i = 1$ upto $m$ do
        $\mathbf{P}_i := \begin{bmatrix}\mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)}\\ \mathbf{0} & \mathbf{P}_i^{(3)}\end{bmatrix}$
    $\mathsf{epk} := \{\mathbf{P}_i\}_{i\in[m]}$
    return epk.

UOV.Verify($\mathsf{epk} = \{\mathbf{P}_i\}_{i\in[m]}, \mu, \sigma = (\mathbf{s}, \mathsf{salt})$):
    $\mathbf{t} \leftarrow \mathsf{Hash}(\mu \| \mathsf{salt})$
    return $\left(\mathbf{t} == \left[\mathbf{s}^{\mathsf{T}}\mathbf{P}_i\mathbf{s}\right]_{i\in[m]}\right)$.
```

Figure 2: The key generation, key expansion, signing and verification algorithms of the UOV signature scheme.

Table 3: Qualitative comparisons of three UOV variants.

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

$\mathbf{L} \in \mathbb{F}_q^{m\times m}$ 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.

Public key expansion. The public key expansion algorithm 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. Given the expanded public key epk and the message/signaure pair (μ, σ), the verification algorithm recomputes the salted hash digest $\mathsf{t} = \mathsf{Hash}(\mu\|\mathsf{salt})$, evaluates $\mathcal{P}$ on the input $\mathsf{s} \in \mathbb{F}_q^n$, and accepts the mesage/signature pair if and only if $\mathsf{t} = \mathcal{P}(\mathsf{s})$.

3.2.4 Specification of the UOV variants

Based on the 5-part API for UOV that we specified thus far, we now instantiate the usual 3-part API in a digital signature scheme (Key Generation, Signing, Verification) in three different ways. We refer to these three instantiations as three *variants* of UOV, but it should be understood that the three variants are essentially the same signature scheme, but with different representations of the secret and public keys. In particular, a signature generated with one variant can be verified by the other variants, and they achieve the same concrete hardness when instantiated with the same set of parameters.

In a nutshell, the three variants `classic`, `pck` and `pkc+skc` are summarized as follows:

- `classic`: in this variant, the public/secret key pair is $(\mathsf{epk}, \mathsf{esk})$, *i.e.*, the ExpandPK and ExpandSK operations are both considered to be part of the key generation algorithm. This means the key sizes are larger, but signing and verification are faster.

- `pkc`: in this public-key-compressed variant the public/secret key pair is $(\mathsf{cpk}, \mathsf{esk})$, *i.e.*, ExpandSK is considered part of the key generation algorithm, but ExpandPK is considered part of the verification algorithm. This makes the public key much smaller (by a factor between 6 and 7), but makes verification slower.

- `pkc+skc`: in this doubly-compressed variant, the public/secret key pair is $(\mathsf{cpk}, \mathsf{csk})$, *i.e.*, ExpandSK is part of the signing algorithm, and ExpandPK is part of the verification algorithm. Compared to the compressed `pkc` variant, the key generation algorithm is faster, and the secret key becomes tiny (only $\mathsf{pk\_seed\_len} + \mathsf{sk\_seed\_len}$ bits), but the signing algorithm becomes much slower.

The key generation, signing, and verification algorithms of the three variants are straightforward combinations of the foregoing five functionalities, *i.e.*, UOV.CompactKeyGen, UOV.ExpandSK, UOV.Sign, UOV.ExpandPK, and UOV.Verifyspecified in Figure 3. For the implementers convenience, we expand out the UOV.CompactKeyGen, UOV.ExpandSK, and

UOV.ExpandPK subroutines in UOV.classic.KeyGen and UOV.pkc.KeyGen, because this allows to reuse some work. Please refer to Table 3 for the qualitative comparisons of these three variants.

3.3 Data Layout in UOV

In this subsection we specify how objects in UOV are encoded as byte strings in our implementations.

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 element 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. A vector of $\mathbb{F}_{16}^{\ell}$ is represented as an $\ell/2$-byte string (we only ever need to encode vectors of even length). The first element of the vector corresponds to the first nibble of the string.

Encoding of epk. Recall that in $\mathsf{epk} = \{\mathbf{P}_i\}_{i\in[m]}$, each upper triangular matrix $\mathbf{P}_i \in \mathbb{F}_q^{n\times n}$ corresponds to a homogeneous quadratic polynomial. Based on n and m, each $\mathbf{P}_i$ matrix is further divided into three components $\mathbf{P}_i^{(1)} \in \mathbb{F}_q^{(n-m)\times(n-m)}$, $\mathbf{P}_i^{(2)} \in \mathbb{F}_q^{(n-m)\times m}$, and $\mathbf{P}_i^{(3)} \in \mathbb{F}_q^{m\times m}$, such that

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

The *expanded* public key epk is encoded as a list of $mn(n+1)/2$ field elements. The first $m(n-m)(n-m+1)/2$ elements encode the $\{\mathbf{P}_i^{(1)}\}_{i\in[m]}$ matrices, the next $m^2(n-m)$ elements correspond to the $\{\mathbf{P}_i^{(2)}\}_{i\in[m]}$ matrices, and the remaining $m^3/2$ elements encode the $\{\mathbf{P}_i^{(3)}\}_{i\in[m]}$ matrices.

Each sequence of m matrix is encoded in an m-fold interleaved fashion: we first encode the first element of each matrix, before moving on to the second element of each matrix and so on. The elements from each matrix appear in the encoding in row-major order. In particular, note that the $\mathbf{P}_i^{(1)}$ and $\mathbf{P}_i^{(3)}$ matrices are upper-triangular, and we do not encode the elements below the diagonals, since they are always zero.

The size of the expanded public key is

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

Encoding of esk. Recall that

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

The byte string for esk is the concatenation of the bit-packed representations of $\mathsf{seed}_{\mathsf{sk}}$, the column-major matrix $\mathbf{O}$, the column-major Macaulay matrix of $\{\mathbf{P}_i^{(1)}\}_{i\in[m]}$, and the column-major Macaulay matrix of $\{\mathbf{S}_i\}_{i\in[m]}$. To be precise,

```
$\mathsf{UOV.}\texttt{classic}\mathsf{.KeyGen}()$:
seed$_{\mathsf{sk}} \leftarrow \{0,1\}^{\mathsf{sk\_seed\_len}}$
seed$_{\mathsf{pk}} \leftarrow \{0,1\}^{\mathsf{pk\_seed\_len}}$
$\mathbf{O} := \mathsf{Expand}_{\mathsf{sk}}(\mathsf{seed}_{\mathsf{sk}})$
$\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$
for $i = 1$ upto $m$ do
    $\mathbf{P}_i^{(3)} := \mathsf{Upper}(-\mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(2)})$
    $\mathbf{P}_i := \begin{bmatrix} \mathbf{P}_i^{(1)} & \mathbf{P}_i^{(2)} \\ \mathbf{0} & \mathbf{P}_i^{(3)} \end{bmatrix}$
    $\mathbf{S}_i := \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}\right)\mathbf{O} + \mathbf{P}_i^{(2)}$
$\mathsf{esk} := \left(\mathsf{seed}_{\mathsf{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right)$
$\mathsf{epk} := \{\mathbf{P}_i\}_{i\in[m]}$
return $(\mathsf{epk}, \mathsf{esk})$.
```

```
$\mathsf{UOV.}\texttt{classic}\mathsf{.Sign}(\mathsf{esk}, \mu)$:
return $\mathsf{UOV.Sign}(\mathsf{esk}, \mu)$.

$\mathsf{UOV.}\texttt{classic}\mathsf{.Verify}(\mathsf{epk}, \mu, \mathsf{sig})$:
return $\mathsf{UOV.Verify}(\mathsf{epk}, \mu, \mathsf{sig})$.
```

```
$\mathsf{UOV.}\texttt{pkc}\mathsf{.KeyGen}()$:
seed$_{\mathsf{sk}} \leftarrow \{0,1\}^{\mathsf{sk\_seed\_len}}$
seed$_{\mathsf{pk}} \leftarrow \{0,1\}^{\mathsf{pk\_seed\_len}}$
$\mathbf{O} := \mathsf{Expand}_{\mathsf{sk}}(\mathsf{seed}_{\mathsf{sk}})$
$\{\mathbf{P}_i^{(1)}, \mathbf{P}_i^{(2)}\}_{i\in[m]} := \mathsf{Expand}_{\mathbf{P}}(\mathsf{seed}_{\mathsf{pk}})$
for $i$ from 1 to $m$ do
    $\mathbf{P}_i^{(3)} := \mathsf{Upper}(-\mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(1)}\mathbf{O} - \mathbf{O}^{\mathsf{T}}\mathbf{P}_i^{(2)})$
    $\mathbf{S}_i := \left(\mathbf{P}_i^{(1)} + \mathbf{P}_i^{(1)\mathsf{T}}\right)\mathbf{O} + \mathbf{P}_i^{(2)}$
$\mathsf{cpk} := (\mathsf{seed}_{\mathsf{pk}}, \{\mathbf{P}_i^{(3)}\}_{i\in[m]})$
$\mathsf{esk} := \left(\mathsf{seed}_{\mathsf{sk}}, \mathbf{O}, \{\mathbf{P}_i^{(1)}, \mathbf{S}_i\}_{i\in[m]}\right)$
return $(\mathsf{cpk}, \mathsf{esk})$.
```

```
$\mathsf{UOV.}\texttt{pkc}\mathsf{.Sign}(\mathsf{esk}, \mu)$:
return $\mathsf{UOV.Sign}(\mathsf{esk}, \mu)$.

$\mathsf{UOV.}\texttt{pkc}\mathsf{.Verify}(\mathsf{cpk}, \mu, \mathsf{sig})$:
$\mathsf{epk} := \mathsf{UOV.ExpandPK}(\mathsf{cpk})$
return $\mathsf{UOV.Verify}(\mathsf{epk}, \mu, \mathsf{sig})$.
```

```
$\mathsf{UOV.}\texttt{pkc+skc}\mathsf{.KeyGen}()$:
$(\mathsf{cpk}, \mathsf{csk}) \leftarrow \mathsf{UOV.CompactKeyGen}()$
return $(\mathsf{cpk}, \mathsf{csk})$.
```

```
$\mathsf{UOV.}\texttt{pkc+skc}\mathsf{.Sign}(\mathsf{csk}, \mu)$:
$\mathsf{esk} := \mathsf{UOV.ExpandSK}(\mathsf{csk})$
return $\mathsf{UOV.Sign}(\mathsf{esk}, \mu)$.

$\mathsf{UOV.}\texttt{pkc+skc}\mathsf{.Verify}(\mathsf{cpk}, \mu, \mathsf{sig})$:
$\mathsf{epk} := \mathsf{UOV.ExpandPK}(\mathsf{cpk})$
return $\mathsf{UOV.Verify}(\mathsf{epk}, \mu, \mathsf{sig})$.
```

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

- $\mathsf{seed}_{\mathsf{sk}}$ is a sequence of $\mathsf{sk\_seed\_len}/8$ bytes;
- We encode $\mathbf{O}$ as a column-major matrix; equivalently, we concatenate the encodings of the m column vectors of length $n - m$;
- The $\{\ \mathbf{P}_i^{(1)}\}_{i\in[m]}$ matrices are encoded in the same way as the $\{\mathbf{P}_i^{(1)}\}_{i\in[m]}$ component of epk;
- And the $\{\mathbf{S}_i\}_{i\in[m]}$ matrices are encoded in the same way as the $\{\mathbf{P}_i^{(2)}\}_{i\in[m]}$ component of epk.

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

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

Encodings of cpk and csk. Recall that

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

The byte string of cpk is the concatenation of the bit-packed representations of $\mathsf{seed}_{\mathsf{pk}}$ and the column-major Macaulay matrix of $\{\mathbf{P}_i^{(3)}\}_{i\in[m]}$. Here, the encoding of the matrices $\{\mathbf{P}_i^{(3)}\}_{i\in[m]}$ is the same as that of the $\{\mathbf{P}_i^{(3)}\}_{i\in[m]}$ component in epk. The byte string for csk is trivially the concatenation of the binary representations of $\mathsf{seed}_{\mathsf{pk}}$ and $\mathsf{seed}_{\mathsf{sk}}$.

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

$$|\mathsf{cpk}| = (\mathsf{pk\_seed\_len} + m^2(m+1)/2 \log q))/8,$$
$$|\mathsf{csk}| = (\mathsf{sk\_seed\_len} + \mathsf{pk\_seed\_len})/8.$$

Encoding of the signature. A signature consists of a vector $\mathbf{s} \in \mathbb{F}_q^n$ and a bitstring $\mathsf{salt} \in \{0,1\}^{\mathsf{salt\_len}}$. The signature is encoded as the concatenation of the encoding of $\mathbf{s}$, which is $\lceil n \log q/8 \rceil$ bytes long, and that of salt. Therefore the signature size is

$$|\mathsf{sig}| = \lceil n/8 \cdot \log q \rceil + \mathsf{salt\_len}/8$$

bytes.

3.4 Recommended Parameter Sets for UOV

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.

First, it should be stressed that each set of recommended parameters in Table 4 is applicable to *every* UOV variant presented in Section 3.2.4. Moreover, note that among all four sets of recommended parameters, we always have $\mathsf{pk\_seed\_len} = 128, \mathsf{sk\_seed\_len} = 256$, and $\mathsf{salt\_len} = 128$. Finally, as shown in Table 4, their key 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.

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

	NIST S.L.	n	m	q	$\lvert\mathsf{epk}\rvert$ (bytes)	$\lvert\mathsf{esk}\rvert$ (bytes)	$\lvert\mathsf{cpk}\rvert$ (bytes)	$\lvert\mathsf{csk}\rvert$ (bytes)	$\lvert\sigma\rvert$ (bytes)
`uov-Ip`	1	112	44	256	278 432	237 896	43 576	48	128
`uov-Is`	1	160	64	16	412 160	348 704	66 576	48	96
`uov-III`	3	184	72	256	1 225 440	1 044 320	189 232	48	200
`uov-V`	5	244	96	256	2 869 440	2 436 704	446 992	48	260

- 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 variants 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 variant with that of the recommended parameters set. For instance, `uov-Is-classic` refers to the UOV instance when we instantiate the `classic` UOV variant with the `uov-Is` parameter set, while `uov-V-pkc+skc` refers to the UOV instance when we instantiate the `pkc+skc` UOV variant with the `uov-V` parameter set.

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

4 Concrete Security Analysis

In this section we introduce the state-of-the-art attacks against UOV scheme, and analyze the hard estimation result of the four sets of recommended parameters proposed in Section 3.4. Table 5 contains lower bounds for the bit-complexity of the state-of-the-art attacks against UOV, and we clarify how the complexities 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. Here, in this documentation the security analysis for UOV is carried out by listing some of the 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* aims 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;
- The *forgery attacks* that aims 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).

salt-UOV. There is yet a UOV variant, *i.e.*, the salt-UOV, which was proposed in 2011 [32] and is very close to our recommended UOV depicted in Section 3. 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 scheme(s) and hence is not as *natural* as the other problems in MQ realm, say the MQ problem. Compared with salt-UOV,

we prefer the recommended UOV depicted in Section 3 for two 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.

For completeness, the salt-UOV scheme and its security argument are presented in A.

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 KS and Intersection attacks are key-recovery attacks, and the Birthday and Direct attacks are universal 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.

Compute evaluations of $\mathcal{P}$. Using 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 ignore the second step in our cost analysis. The cost of the polynomial evaluations is $\frac{3mr}{2}X$ bit operations.

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

$$\frac{1}{(1-e^{-\alpha})}\left(\frac{3mr}{2}X + 2^{17.5}Y\right),$$

which is approximately equal to $2^{10.7}\sqrt{q^m mr}$ for optimally chosen X, Y and α[1]. This is the formula we use in Table 5. This above formula should be interpreted as a conservative lower bound for the "true" cost of the attack. Note that there the `uov-Is` entry should compute to 142.7, which we are comfortable rounding up to 143 because the collision-finding approach we describe here would require a huge amount of memory. This incurs a cost[2]. We have not multiplied the numbers in Table 5 with this factor because realistically, an attacker would use a memoryless collision-finding algorithm such as e.g., [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 $\mathsf{salt}^* \in \{0,1\}^*$ on his will, computes $\mathbf{t} = \mathsf{Hash}(\mu^*\|\mathsf{salt}^*)$, 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 $m' = m - 1$

[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'."

equations in $n' = m - 1$ variables with the approach of Thomae and Wolf [50], and then using the hybrid WiedemannXL algorithm to solve the new system. The estimated cost of this state-of-the-art approach is

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

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] 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 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 $m = O(n)$ times so as to get a basis for O. Nevertheless, this does not contribute an m factor into the overall cost intuitively, because once a 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 [**?**]. It tries to simultaneously find k 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 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 $n < \frac{2k-1}{k-1}m$. For details, we refer to [8].

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].. 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, and the Minors Modeling method.

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, with cost

The complexity of MinRank attack on UOV is

$$\mathcal{O}\left(\left(\left\lceil \frac{n-m}{m} \right\rceil (n-m)\binom{n'}{n-m}\right)^2 \cdot \left\lceil \frac{n-m}{m} \right\rceil (n-m)(n-m+1)\right),$$

where the positive integer $n' \in \{n-m+1, ..., n\}$ satisfies

$$m\binom{n'}{n-m+1} \geq (n-m)\binom{n'}{n-m} - 1.$$

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 $128, 192$, or 256-bit keys respectively. Grover speeds up a key search by almost a square root factor, so, for a quantum attack to break the NIST security targets it needs to improve on classical attacks by more than a square root factor, which is not possible by relying on Grover's algorithm alone.

5 Implementations and Performance

Recall that in Section 3 we have presented *three* UOV variants as well as *four* sets of recommended parameters. This section specifies the implementations of these $12 = 3 \times 4$ UOV instances 3 over various platforms, as well as their performance, so as to fully demonstrate the strengths of UOV in practice. The contents of this section come from [7].

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

UOV signing requires solving the system of linear equations $\mathbf{Ax} = \mathbf{b}$ for the m variables $\mathbf{x} \in \mathbb{F}_q^m$, where $\mathbf{A} = \mathbf{L}$ and $\mathbf{b} = \mathbf{t} - \mathbf{y}$. For this we use a constant-time Gaussian elimination algorithm and back-substitution (Algorithm 1). As the first step (line 3) in the outer loop, we conditionally add all following rows to make sure the pivoting 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 7) as the matrix is not invertible or the system of linear equations has no unique solution. Leaking that the matrix is not invertible via a timing side-channel is not an issue as the matrix is discarded if it is not invertible. Then, we invert the pivoting element (line 8) and multiply the current row by the inverse (line 9). We then add multiples of that row to the remainder of the matrix (line 11). We then back-substitute the variables into the system of equations to obtain the solutions (line 14).

Note that, the previous works [47, 17], explicitly compute the inverse of the matrix $\mathbf{A}$ and then derive the solution with a matrix multiplication as $\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b}$. 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 pivoting element is already nonzero. We instead propose to limit the additions to a small number of rows. We propose to add at most 15 rows for $\mathbb{F}_{16}$ and at most 7 rows for $\mathbb{F}_{256}$. This results in a probability of at most $m \cdot 16^{-16} = 2^{-58}$ and $m \cdot 256^{-8} \leq 2^{-57.4}$ to wrongly abort for the $\mathbb{F}_{16}$ and $\mathbb{F}_{256}$ parameters, respectively, which we deem is sufficiently small.

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 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 end and, hence, vastly reducing the number of required multiplications. This results in a substantial speed-up. In the case of $\mathbb{F}_{16}$, we use 15 accumulators: one for each possible value of $s_i s_j$ except for zero as those columns can be discarded straight away. In the case of $\mathbb{F}_{256}$, we use 2×15 accumulators: one set for the four least significant bits, and one set for the four most

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
        $a'_{j,m} := a'_{j,m} + a'_{i,j} a'_{i,m}$
return last column of $\mathbf{A}'$
```

significant bits. Each column gets added to the corresponding accumulator of each set. By using different accumulators for the high and low bits, we keep the memory requirements for this approach reasonable while still vastly reducing the number of required costly field multiplications. Note that this approach results in signature-dependent memory access patterns which may be problematic in case signatures are secret and if the targeted device leaks memory addresses, *e.g.*, through cache timing side channels. For the majority of cases, however, the signature is public and this approach should be used for signing speed.

Skipping parts of the public key. As already pointed out by Chou, Kannwischer, and Yang [17], the verification can be further sped-up by exploiting that in case a monomial $s_i s_j$ is zero, the corresponding columns in the Macaulay do not affect the result as they are multiplied by zero. We, hence, skip ahead in case either of the variables is zero. This is particularly significant when working with $\mathbb{F}_{16}$ as 1/16 of variables are expected to be zero, which means 31/256 of the products $s_i s_j$ is expected to be zero.

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

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`	3 311 188	116 624	82 668	2 903 434	105 324	90 336
`uov-Ip-pkc`	3 393 872		311 720	2 858 724		224 006
`uov-Ip-pkc+skc`	3 287 336	2 251 440		2 848 774	1 876 442	
`uov-Is-classic`	4 945 376	123 376	60 832	4 332 050	109 314	58 274
`uov-Is-pkc`	5 002 756		398 596	4 376 338		276 520
`uov-Is-pkc+skc`	5 448 272	3 042 756		4 450 838	2 473 254	
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`	22 046 680	346 424	275 216	17 603 360	299 316	241 588
`uov-III-pkc`	22 389 144		1 280 160	17 534 058		917 402
`uov-III-pkc+skc`	21 779 704	11 381 092		17 157 802	9 965 110	
`uov-V-classic`	58 162 124	690 752	514 100	48 480 444	591 812	470 886
`uov-V-pkc`	57 315 504		2 842 416	46 656 796		2 032 992
`uov-V-pkc+skc`	57 306 980	26 021 784		45 492 216	22 992 816	

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

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

be possible when using a sponge-based extendable-output function (XOF) like `shake256` which may have appeared to be a natural choice for seed expansion. "Lazy sampling" results in a significant speed-up especially for $\mathbb{F}_{16}$.

5.2 x86 AVX2 Implementation

In this subsection we present our optimization 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. While NIST is requiring code primarily for the Intel Haswell microarchitecture, we additionally study the Intel Skylake microarchitecture as it is easily available more than Haswell and results in better performance.

Symmetric primitives. For implementing the four symmetric primitives (Hash, $\mathsf{Expand}_{\mathbf{v}}$, $\mathsf{Expand}_{\mathsf{sk}}$, and $\mathsf{Expand}_{\mathbf{P}}$), we call the OpenSSL library when relating to standard cryptographic primitives, *e.g.*, `shake256` and `aes128`. For $\mathsf{Expand}_{\mathbf{P}}$ using round-reduced AES, we adapt the `aes128ctr` implementation in [24], which utilizes x86 AES instructions, to implement only 4 AES rounds.

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 `clang` version `14.0.0-1ubuntu1` and the performance numbers are measured on Intel Xeon E3-1230L v3 1.80GHz (Haswell) and Intel Xeon CPU E3-1275 v5 3.60GHz (Skylake) with turbo boost and hyper-threading disabled.

Results. Table 6 reports the performance of our AVX2 implementations and comparisons to other standard PQC schemes, whereas Table 7 shows the results with round-reduced AES. In Table 6, we merge the numbers for **Sign()** from `classic` and `pkc` versions and **Verify()** from `pkc` and `pkc+skc` to indicate that they use the same implementations. Among all comparisons, Table 6 shows that

1) `uov-Ip` has the fastest signing while `uov-Is` signing is only 2% slower;

2) `uov-Is` has the fastest verification although its public key is larger than `uov-Ip`. This stems from the fact that `uov-Ip` uses more `XOR` operations for the 2 accumulators while evaluating $\mathbb{F}_{256}$ public polynomials (see Section 5.1.2);

3) For verification with compressed keys, the computation of $\mathsf{Expand}_{\mathbf{P}}$, *i.e.*, `aes128ctr`, dominates the execution time, which can be seen by comparing with the results of 4-round AES in Table 7. The round-reduced AES improves the verification time by around 40%;

4) For signing with compressed secret keys, the main computation is spent on expanding the compressed keys.

5.3 Arm Neon Implementation

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

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

We have two different Neon implementations for `aes128ctr` depending on the availability of Arm AES instructions. On platforms supporting AES instructions, *e.g.*, Apple M1, we implement the standard and round-reduced `aes128ctr` with AES instructions. On platforms without AES instructions, *e.g.*, Raspberry Pi4b, we port the bitsliced implementation for 32-bit platforms in [3], 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 Debian `clang` version `version 11.0.1-2`. The Macbook has an Apple M1 CPU running at 3.2 GHz with Arm AES instruction support. Its compiler is Apple `clang` version `14.0.0 (clang-1400.0.29.202)`.

Table 8: Benchmarking results of our Neon implementations. Numbers are median CPU cycles of 10 000 executions.

	Cortex-A72			Apple M1		
	KeyGen	Sign	Verify	KeyGen	Sign	Verify
`uov-Ip-classic`	11 172 204	245 095	142 868	1 793 119	55 289	49 719
`uov-Ip-pkc`	11 193 794		3 677 844	1 775 826		112 934
`uov-Ip-pkc+skc`	11 229 231	7 617 137		1 774 748	1 056 617	
`uov-Is-classic`	29 269 925	460 655	141 528	3 391 967	74 633	45 908
`uov-Is-pkc`	28 906 183		5 070 253	3 360 648		138 496
`uov-Is-pkc+skc`	29 467 684	16 413 501		3 393 812	2 089 131	
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`	66 871 027	1 542 143	574 080	9 836 359	147 564	189 837
`uov-III-pkc`	66 554 826		17 161 246	9 803 637		461 896
`uov-III-pkc+skc`	64 147 364	42 794 977		9 751 198	6 353 401	
`uov-V-classic`	313 814 250	3 316 413	1 319 092	28 286 979	293 826	376 000
`uov-V-pkc`	305 700 907		39 337 795	26 743 866		1 011 331
`uov-V-pkc+skc`	312 729 427	107 305 680		26 663 940	15 830 169	

† Security level II.

Table 9: Benchmarking results of Neon implementations using 4-round AES for public-key expansion. Numbers are median CPU cycles of 10 000 executions.

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

Results. Table 8 reports the results of Neon UOV implementation and comparison with other PQC signatures on the two Armv8-A platforms. Table 9 shows results with the 4-round AES option of $\mathsf{Expand}_{\mathbf{P}}$. The results show that:

1) `uov-Ip` has the best signing time which is consistent with the results of AVX2 implementation (Table 6). However, `uov-Ip` outperforms `uov-Is` by a margin on Neon while, on AVX2, `uov-Ip` leads `uov-Is` by $< 10\%$. This is caused by the mismatch between the sizes of registers and vectors. When processing line 9 and 10 of **Sign()** in Figure 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.

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

Table 10: 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 10 000 executions.

		speed (clock cycles)		
	variants	KeyGen	Sign	Verify
`uov-Ip` (This work)	`classic`	138 833k	2 482k	995k
	`pkc`	175 020k		11 551k (10 717k)
	`pkc+skc`	175 021k	88 757k	
`uov-Is` (This work)	`classic`	195 744k	2 374k	616k
	`pkc`	203 321k		16 045k (15 175k)
	`pkc+skc`	296 161k	113 446k	
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

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

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

5.4 Arm Cortex-M4 Implementation

This section covers our implementations of UOV for the Arm Cortex-M4. We base our implementation on the Rainbow implementation by Chou, Kannwischer, and Yang [17]. Due to the stack limitations of available Cortex-M4 cores, in this section we restrict our implementations to the two sets of recommended parameters for NIST security level 1, *i.e.*, `uov-Ip` and `uov-Is`.

Symmetric primitives. For implementing Hash, $\mathsf{Expand}_{\mathbf{v}}$, and $\mathsf{Expand}_{\mathsf{sk}}$, we use `shake256` as implemented in pqm4 [45] which integrates the Keccak permutation in `Armv7-M` assembly from the XKCP [53]. For implementing the sampling of the public key ($\mathsf{Expand}_{\mathbf{P}}$), 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.

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

		memory consumption (bytes)			code size (KiB)
	variants	KeyGen	Sign	Verify	
`uov-Ip` (This work)	`classic`	15 744	5 268	2 548	72.4
	`pkc`	142 312		6 592 (280 980)	75.3
	`pkc+skc`	380 248	243 204		75.5
`uov-Is` (This work)	`classic`	613 056	5 468	1 024	31.6
	`pkc`	350 072		5 248 (413 632)	33.2
	`pkc+skc`	416 636	354 216		33.6
Dilithium-2 [2]		38 000	49 000	36 000	26.0
Falcon-512 [44, 45]		18 384	42 528	4 484	79.9
sphincs-128f† [45]		2 104	2 168	2 656	13.3
sphincs-128s‡ [45]		2 432	2 392	1 960	13.6

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

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

	variants	key generation w/o flashing (cc)	flashing (cc)	key generation w/ flashing (cc)
`uov-Is`	`classic`	195 744k	202 296k	398 040k
	`pkc`	203 321k	110 744k	314 065k
	`pkc+skc`	296 161k	18 287k	314 447k

Keys exceeding RAM size. For the `uov-Is` parameter sets, the total size of the expanded secret key and the expanded public key is 743 KB which exceeds the RAM of our target platform. To still be able to benchmark all primitives, we split up key generation into secret key and public key computation. We then write the keys to flash memory as was previously proposed by Chen and Chou for Classic McEliece [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 10 contains the performance benchmarks for Arm Cortex-M4. We present cycle counts for all six variants of the level one parameter sets. Due to timing variations (depending only on public data) in signing and verification, we perform 10 000 measurements and report the average. Note that public key compression does not affect signing performance, while secret key compression does not affect verification performance. For the `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 11 contains the memory utilization of our implementation excluding the key material. The parameter sets using secret key compression are currently performing

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)		
	variants	KeyGen	Sign	Verify
`uov-Ip`	`pkc`	169 280k	2 502k	5 804k
	`pkc+skc`	169 281k	83 018k	
`uov-Is`	`pkc`	194 875k	2 390k	7 594k
	`pkc+skc`	287 715k	105 004k	

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

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

5.5 FPGA Implementation

In this section, we present our field-programmable gate array (FPGA) design for UOV signatures and report the performance of the design on popular platforms. Since our design supports multiple parameters and variants of UOV, 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 functions in Figure 2 with pre-loaded firmware using the proposed instructions.

Target platform. We test our design on two Xilinx Artix-7 platforms: Zynq-7000$^{\text{TM}}$ 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 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 variants 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

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

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 in `classic` mode, 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 `pkc+skc` mode is dominated by the **ExpandSK()** function, specifically, the calculation of the $\mathbf{S}_i=(\mathbf{P}_i^{(1)}+\mathbf{P}_i^{(1)\mathsf{T}})\mathbf{O}+\mathbf{P}_i^{(2)}$. 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`, **ExpandSK()** takes 248 336 cycles. The remaining computation includes 7 515 cycles for tasks such as Gaussian elimination and polynomial evaluation, and 189 618 cycles for expanding $\mathbf{P}^{(1)}$ and $\mathbf{P}^{(2)}$ from $\mathsf{seed}_{\mathsf{pk}}$. In the end, with savings from overlapping these computations, it results in 78 262 cycles in `uov-Ip-pkc+skc`.

The cycle count of verification in `classic` mode is approximately $n\times(n+1)/2$ cycles, which is consistent with 6 328 for `uov-Ip`. On the other hand, the cycle count of verification in `pkc` mode, is limited by the throughput of the $\mathsf{Expand}_{\mathbf{P}}()$ function. The AES module

Table 16: The performance results using pipelined AES.

AES		Utilization					Cycle Count			Freq.
rounds		Slices	LUTs	FFs	BRAM	DSP	KeyGen	Sign	Verify	(MHz)
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

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 $\mathsf{Expand}_{\mathbf{P}}()$ in Table 15. It shows a reduction in cycles for verification in `pkc` mode and signing in `skc`. The saving for verification matches our expectation, which can be estimated by the difference in rounds multiplied by the number of calls to the AES module. It is $(8 \cdot 44 \cdot ((112+44)(112-44)/2)/128) \cdot 6 = 87\,516$ cycles in the case of `uov-Ip`. For signing in `skc` variants, the saving is less significant because computing the $\mathbf{S}_i$'s 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 in `pkc+skc`, the cycle count slightly improves since the bottleneck is the computation of the $\mathbf{S}_i$. The cycles for 4-round and 10-round AES are similar since both are pipelined, generating 128-bits per cycle.

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 of size only a few hundred bits 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. Though there does not exist a formal security proof which connects the security of UOV to a hard mathematical problem such as MQ problem, there are good reasons to have confidence in the security of UOV. 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. We furthermore note here 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 utilized to attack the scheme.

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 variants of UOV with smaller keys to accommodate use cases that would benefit from them.

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. NISTIR 8413, 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`.

[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, accessed October 12, 2022.

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

[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. 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. In `https://www.linaro.org/blog/accelerated-aes-for-the-arm64-linux-kernel/`.

[12] Ward Beullens, Bart Preneel, Alan Szepieniec and Frederik Vercauteren. LUOV signature scheme proposal for NIST PQC project (Round 2 version), 2019.

[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.

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

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

[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. In https://bench.cr.yp.to, accessed February 13, 2022.

[21] FIPS PUB 197 – Advanced Encryption Standard (AES), 2001. Available at https://doi.org/10.6028/NIST.FIPS.197.

[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.

[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.

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

[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.

[29] H. Ong, C.P., Scnorr and A. Shamir. 1984. An Efficient Signature Scheme Based on Quadratic Equations In Proceedings of the sixteenth annual ACM symposium on Theory of computing (STOC '84). Association for Computing Machinery, New York, NY, USA, 208–216. https://doi.org/10.1145/800057.808683

[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 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`.

[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`

[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`.

[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 June 2020, available at `http://precision.moscito.org/by-publ/recent/rainbow-pars.pdf`

[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://homes.esat.kuleuven.be/~nsmart/MPC/`.

[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`. Accessed January 19, 2023.

[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`.

[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 from
`csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions`

A The salt-UOV and Its EUF-CMA Security

There is yet a UOV variant, *i.e.*, the salt-UOV, which 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 scheme(s) and hence is not as *natural* as the other problems in MQ realm, say the MQ problem. Compared with salt-UOV, our confidence in the recommended UOV lies in the fact that all the state-of-the-art attacks against our recommended UOV are applicable to salt-UOV, and vice versa.

For completeness, we present the salt-UOV scheme and its security argument below.

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 $\mathbf{v} \in \mathbb{F}_q^{n-m}$, 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 sampled uniformly in $\mathbb{F}_q^m$, $\mathcal{P} = \mathcal{F} \circ \mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^m$ 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 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) probabilistic polynomial-time algorithm A, its success probability in solving the UOV problem is negligibly small.

Security proof of salt-UOV. And the relation between the UOV problem and the salt-UOV scheme is summarized by the following theorem. Please refer to [32] for the full proof of Theorem 1.

Theorem 1. *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 $q_h = \mathsf{poly}(\lambda)$ hash queries and $q_s = \mathsf{poly}(\lambda)$ 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*

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

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

Last but not the least, it should be stressed that the UOV problem presented in Definition 2 is slightly *different* from the classic definition of one-way function [55] in terms of the distribution of the image $\mathbf{t}$. The requirement that $\mathbf{t}$ should follow the uniform distribution over $\mathbb{F}_q^n$ is essential for the *correctness* of Theorem 1, but makes the security argument in Theorem 1 less convincing than expected.

```
**KeyGen**(params = $(n, m, q, \mathsf{salt\_len})$):
    Choose OV-polynomials $f^{(1)}(x_1, ..., x_n), ..., f^{(m)}(x_1, ..., x_n)$ uniformly at random
    $\mathcal{F} := (f^{(1)}, ..., f^{(m)})$
    Choose an invertible linear transformation $\mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ uniformly at random
    $\mathcal{P} := \mathcal{F} \circ \mathcal{T}$
    $\mathsf{pk} := \mathcal{P}$
    $\mathsf{sk} := (\mathcal{F}, \mathcal{T})$
    **return** $(\mathsf{pk}, \mathsf{sk})$

**Sign**$\big(\mathsf{params}, \mathsf{sk} = (\mathcal{F}, \mathcal{T}), \mu \in \{0,1\}^*\big)$:
    $\mathbf{v} \leftarrow \mathbb{F}_q^{n-m}$
    **repeat**                                                          ▷ Beginning of rejection-sampling phase
        $\mathsf{salt} \leftarrow \{0,1\}^{\mathsf{salt\_len}}$
        $\mathbf{t} \leftarrow \mathsf{Hash}(\mu \| \mathsf{salt})$                                     ▷ $\mathsf{Hash} : \{0,1\}^* \to \mathbb{F}_q^m$
        $\Delta_{\mathbf{t}} := \left\{ \begin{bmatrix} \mathbf{v} \\ \mathbf{u} \end{bmatrix} \in \mathbb{F}_q^n \;\middle|\; \mathcal{F}\left(\begin{bmatrix} \mathbf{v} \\ \mathbf{u} \end{bmatrix}\right) = \mathbf{t} \right\}$           ▷ $\mathbf{u} \in \mathbb{F}_q^m$
    **until** $\Delta_{\mathbf{t}} \neq \emptyset$                                               ▷ End of rejection-sampling phase
    $\mathbf{w} \leftarrow \Delta_{\mathbf{t}}$
    $\mathbf{s} := \mathcal{T}^{-1}(\mathbf{w})$
    $\sigma := (\mathbf{s}, \mathsf{salt})$                                         ▷ $\sigma \in \mathbb{F}_q^n \times \{0,1\}^{\mathsf{salt\_len}}$
    **return** $\sigma$

**Verify**$(\mathsf{params}, \mathsf{pk} = \mathcal{P}, (\mu, \sigma = (\mathbf{s}, \mathsf{salt})))$:
    $\mathbf{t} \leftarrow \mathsf{Hash}(\mu \| \mathsf{salt})$
    $\mathbf{t}' := \mathcal{P}(\mathbf{s})$
    **return** $(\mathbf{t} == \mathbf{t}')$
```

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