Embedded Multilayer Equations: a New Hard Problem for Constructing Post-Quantum Signatures Smaller than RSA (without Hardness Assumption)*

Dongxi Liu

CSIRO Data61, Australia
dongxi.liu@csiro.au

Abstract. We propose a new hard problem, called the Embedded Multilayer Equations (eMLE) problem in this paper. An example of eMLE, with one secret variable x and three layers, is given below.

$$6268 = 57240 * x + (1248 * x + (9 * x \texttt{ mod } 16) \texttt{ mod } 2053) \texttt{ mod } 65699$$

In this example, the eMLE problem is to find x from the above equation. eMLE in this paper has the same number of variables and equations. The hardness of eMLE problem lies in its layered structure. Without knowing the eMLE value of lower layer (i.e., the layer with modulus 2053), the top layer (i.e., the layer with modulus 65699) has many candidate solutions; the adversary has to search the solution space for a few valid ones. A lower-bound for the number of searches has been proven in the paper, together with the expected number of valid solutions. The hardness of eMLE can be increased by adding more layers, without changing the number of variables and equations; no existing NP-complete problems have this feature.

Over the hardness of eMLE, a post-quantum signature scheme, eMLE-Sig, is constructed. Compared with all existing signature schemes (conventional and post-quantum), eMLE-Sig might be the simplest to understand, analyze, instantiate, and implement. At the security level above 128 bits, five configurations are provided; all of them have keys and signatures smaller than RSA keys and signatures (above 380 bytes) at the 128-bit security level. The smallest configuration is with two variables and three layers, having 84.1/52.2 bytes for private/public key and 168.4 bytes for signatures.

1 Introduction

Current post-quantum signature schemes usually have bigger keys and/or bigger signatures than conventional signature algorithms[1] at the same security level. They are also more complex in terms of their constructions and underlying hardness assumptions. These issues increase the difficulty of deploying post-quantum signature schemes

* The updated signature scheme addressing the attacks found by Lorez Panny and Dan Brown in NIST's PQC forum is included in Appendix B as SageMath code. The paper has not been revised to reflect the revision; however, it is still helpful for understanding most parts of the new scheme.

[1] https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

in various environments and cause security uncertainty from, for instance, improved solutions to hard problems and insecure implementation for new environments.

In this paper, we propose a new hard problem, the Embedded Multiplayer Equations (eMLE) problem, and use it to construct a signature scheme eMLE-Sig to address the issues of existing post-quantum signature schemes. An example of eMLE with one variable and three layers is given in the Abstract. In eMLE-Sig, we use eMLE at least two variables (for the moment). Hence, an example of eMLE, with two integer variables x_0 and x_1, and three layers, is provided below to illustrate, where $p_0 < p_1 < p_2$ are three co-prime positive integers.

$$\begin{bmatrix} h_0 \\ h_1 \end{bmatrix} = \begin{bmatrix} g_0 & g_1 \\ g_2 & g_3 \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + (\begin{bmatrix} g'_0 & g'_1 \\ g'_2 & g'_3 \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} + (\begin{bmatrix} g''_0 & g''_1 \\ g''_2 & g''_3 \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \ \texttt{mod}\ p_0) \ \texttt{mod}\ p_1) \ \texttt{mod}\ p_2$$

For the above example, the eMLE problem is to find the two variables x_0 and x_1 from the two equations, where all other values are public. eMLE in this paper has the same number of variables and equations. In each equation, the calculations on a variable are performed over different moduli, each of which specifies a layer. h_0 and h_1 are called the eMLE values of the top layer, similarly for lower layers.

A lower layer has a smaller modulus, with its eMLE value embedded into the upper layer; hence, the eMLE values of lower layers are not known. The number of variables and the number of layers are independent. For example, still with two variables and two equations, the above example can be increased to four layers, five layers, or more.

On the contrary, for hard problems used by current post-quantum signature schemes, such as LWE [5], SIS [1] and Multivariate Quadratic Equations (MQ) [7], the numbers of equations and variables are usually different and all variables are operated with the same modulus. The LWE and SIS problems simply become meaningless when the numbers of equations and variables are equal.

The structure of problems in current post-quantum signature schemes is flat; their hardness is increased by adding more variables/dimensions and equations accordingly. On the contrary, the hardness of eMLE can be increased by independently adding either variables/equations or layers; that is, its hardness can be increased by adding more layers, without changing the number of variables and equations. More generally, the current NP-complete problems do not have this feature.

The layered structure of the eMLE problem is able to enforce the ways an adversary can take to solve the problem. The adversary can only start by solving the top layer equations (i.e, the layer with modulus p_2 in the example), because the eMLE values of lower layers are not known and thus the lower layer equations cannot even be established. Without knowing the eMLE values of the second top layer (the layer with modulus p_1 in the example), the top layer has many solutions and the adversary has to do some brute-force search to find a valid one. In this paper, we have proven the probability that a solution is valid and the lower-bound of the number of searches the adversary has to carry out. For the example above, the probability of obtaining a valid solution is $\frac{2^{(3-1)}}{p_1}$, and the lower-bound of the number of searches is p_1.

Two problems with layered structure (called factually hard problems) have been proposed in [4] to demonstrate the feasibility of public-key encryption without hardness assumptions. eMLE is also a factually hard problem because of its layered structure.

The problems in [4] are used for constructing public-key encryption schemes, whilst eMLE aims to construct an efficient post-quantum signature scheme and demonstrate the feasibility of constructing signature schemes without hardness assumptions.

Based on eMLE, the signature scheme eMLE-Sig is constructed. Its private key is the secret variables in eMLE (e.g., x_0, x_1), and its public key is the eMLE values (h_0, h_1). Compared with all current conventional and post-quantum signature schemes, eMLE-Sig might be the simplest one to understand, analyze, instantiate, and implement, since it does not need more complex mathematical concepts beyond basic modular arithmetic and hash functions.

We have implemented eMLE-SIG with SageMath and the implementation is in Appendix. Above the security level of 128 bits, we have provided five parameter configurations. Among them, the smallest instance with two variables of three layers has 84.1 bytes for private key, 52.2 bytes for public key, and 168.4 bytes for signatures. The biggest instance is with four layers of three variables, having 174.7 byes for private key, 126.7 bytes for public key, and 349.2 bytes for signatures. The biggest instance is still more compact than RSA signatures, which have more than 380 bytes for keys and signatures, at the 128-bit security level.

1.1 Notations

Given a positive integer q, let $\mathbb{Z}_q$ refer to the set $\{0, ..., q-1\}$. For a finite set, e.g. S, $x \leftarrow S$ means that x is uniformly sampled from S at random. Given two positive integers $a < b$, then (a, b) indicates the set of integers from a to b. The n-ary Cartesian product of a set S is denoted by S^n.

A lower-case boldface letter denotes a vector or a list (e.g., $\mathbf{x}$), while a list of matrices is represented by a upper-case calligraphy letter (e.g., $\mathcal{G}$). The ith element of list $\mathbf{x}$ is written as $\mathbf{x}[i]$, with the first element indexed by 0. The prime number just after an integer n is denoted by $\mathtt{np}(n)$.

2 Embedded Multilayer Equations (eMLE)

Let $n \geq 1$ and $d \geq 3$ be two positive integers, indicating the number of variables and the number of layers (or depth), respectively, in eMLE. Let $\mathbf{p}$ be a list containing d positive integers, which are co-prime and indicate the modulus of each layer. An upper layer is required to have a bigger modulus, hence $\mathbf{p}[j] < \mathbf{p}[i]$ for $0 \leq j < i \leq d-1$.

Let $\mathcal{G}$ be a list of length d, with the ith element being a matrix $\mathcal{G}[i] \leftarrow \mathbb{Z}_{\mathbf{p}[i]}^{n*n}$ and $\mathcal{G}[d-1]$ is invertible with respect to $\mathbf{p}[d-1]$. Let $\mathbf{x}$ be a vector of n integers, each of which is in the range $(\mathbf{p}[d-2], \mathbf{p}[d-1])$. Then, a system of embedded multilayer equations (eMLE) is represented as

$$\mathtt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}) = \mathbf{h}$$

where $\mathbf{h} = \mathbf{h}_{d-1}$ and $\mathbf{h}_{d-1}$ is recursively defined as:

$$\mathbf{h}_i = \begin{cases} \mathcal{G}[i] * \mathbf{x} \bmod \mathbf{p}[i], & \text{if } i = 0 \\ \mathcal{G}[i] * \mathbf{x} + \mathbf{h}_{i-1} \bmod \mathbf{p}[i], & \text{otherwise} \end{cases}$$

Two eMLE examples have been presented in the previous section and the abstract. In current definition, an eMLE has n variables and n equations; however, underdetermined/overdetermined eMLE can be supported.

Theorem 1. *Given* $\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}) = \mathbf{h}$*, and* $\mathbf{x} \neq \mathbf{x}'$*, the probability of* $\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}') = \mathbf{h}$ *is at most* $(\frac{2^{d-1}}{\mathbf{p}[d-2]})^n$.

Proof. If $\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}') = \mathbf{h}'$, then $\mathbf{x}'$ can be uniquely determined with ${\mathbf{h}'}_{d-2}$. There are $\mathbf{p}[d-2]^n$ possible values for ${\mathbf{h}'}_{d-2}$, each of which can lead to a potential solution to $\mathbf{x}'$, since $\mathcal{G}[d-1]$ is invertible.

If $\mathbf{h} = \mathbf{h}'$, then we have

$$\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}') - \texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}) = 0 \bmod \mathbf{p}[d-1].$$

With such $\mathbf{x}'$, let $\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}' - \mathbf{x}) = \mathbf{h}''$. Then, each element in $\mathbf{h}''$ has a value $\Sigma_{i=0}^{d-2} b_i * \mathbf{p}[i]$, where b_i can be either 0 or 1. That is, at most $2^{(d-1)n}$ possible values for $\mathbf{x}'$ to generate the same $\mathbf{h}$. Thus, the probability of $\mathbf{x}'$ with $\mathbf{h}' = \mathbf{h}$ is $(\frac{2^{d-1}}{\mathbf{p}[d-2]})^n$. □

Theorem 2. *Given* $\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}) = \mathbf{h}$*, the expected number of solutions for* $\mathbf{x}$ *is* $2^{(d-1)n}$.

Proof. Given $\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}) = \mathbf{h}$, there are $\mathbf{p}[d-2]^n$ possible values for $\mathbf{x}$, because

$$\mathbf{x} = \mathcal{G}[d-1]^{-1} * (\mathbf{h} - \mathbf{h}_{d-2}) \bmod \mathbf{p}[d-1],$$

where $\mathbf{h}_{d-2} \in \mathbb{Z}^n_{\mathbf{p}[d-2]}$. Based on Theorem 1, each value in $\mathbb{Z}^n_{\mathbf{p}[d-2]}$ has the probability $(\frac{2^{d-1}}{\mathbf{p}[d-2]})^n$ to ensure its eMLE value is $\mathbf{h}$. Hence, the expected number of solutions is $2^{(d-1)n}$. □

We then evaluate the above theorem with experiments. In these experiments, a brute-force search of $\mathbf{h}_{d-2}$ in $\mathbb{Z}^n_{\mathbf{p}[d-2]}$ is carried out to recover valid $\mathbf{x}$. Let $n = 1$ for all experiments. Table 1 lists the configurations of parameters d, $\mathcal{G}$, $\mathbf{p}$, and the corresponding number of valid solutions to $\mathbf{x}$, where each parameter configuration is repeated 100 times. For example, in the first case of Table 1, $\mathbf{x}$ has one solution 39 times, and two solutions 61 times.

Furthermore, for the first case in Table 1, we also search $\mathbf{x}$ directly in $\mathbb{Z}_{\mathbf{p}[d-1]}$; the same result is obtained. The experiments show that the number of valid solutions to $\mathbf{x}$ is less than the expected value $2^{(d-1)*1}$. A smaller number of solutions is better for security, since the adversary has less chance to recover a valid $\mathbf{x}$, as shown by the following theorem.

Theorem 3. *Let* $\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}) = \mathbf{h}$*, with* $n \geq 2$*,* $d \geq 3$*, and* $\mathbf{p}[d-1] - \mathbf{p}[d-2] > \mathbf{p}[d-2]$*. Then, it is necessary for the adversary to exhaustively search at least* $\mathbf{p}[d-2]$ *times with a probability* $\frac{2^{(d-1)n}}{\mathbf{p}[d-2]}$ *to find a solution to* $\mathbf{x}$*.*

Proof. Given $\texttt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x}) = \mathbf{h}$, we have

$$\mathbf{x} = \mathcal{G}[d-1]^{-1} * (\mathbf{h} - \mathbf{h}_{d-2}) \bmod \mathbf{p}[d-1],$$

	1	2	3	4	5	6	7
$\mathbf{p} = (8, 37, 149)$ $\mathcal{G} = [[0], [13], [135]]$ $d = 3$	39	61	0	0	0	0	0
$\mathbf{p} = (16, 1031, 32993)$ $\mathcal{G} = [[7], [205], [16193]]$ $d = 3$	37	63	0	0	0	0	0
$\mathbf{p} = (8, 131, 2099, 33587)$ $\mathcal{G} = [[1], [103], [1103], [18464]]$ $d = 4$	35	57	8	0	0	0	0
$\mathbf{p} = (16, 521, 16673, 533543)$ $\mathcal{G} = [[7], [140], [13441], [200411]]$ $d = 4$	29	40	28	3	0	0	0

Table 1: Number of Solutions to $\mathbf{x}$ (100 tests for each configuration)

where $\mathbf{h}_{d-2} \in \mathbb{Z}^n_{\mathbf{p}[d-2]}$. It is an under-determined system of equations. To find a solution to $\mathbf{x}$, the adversary has to start by guessing at least one element for $\mathbf{h}_{d-2}$ in $\mathbb{Z}_{\mathbf{p}[d-2]}$, or at least one element in $(\mathbf{p}_{d-2}, \mathbf{p}_{d-1})$ for $\mathbf{x}$. Since the set $\mathbb{Z}_{\mathbf{p}[d-2]}$ is smaller, the better strategy for the adversary is to search $\mathbb{Z}_{\mathbf{p}[d-2]}$ and then recover $\mathbf{x}$ with the above equation. Theorem 3 shows the expected number of solutions to $\mathbf{x}$ is $2^{(d-1)n}$. Hence, the adversary has a probability $\frac{2^{(d-1)n}}{\mathbf{p}[d-2]}$ to find the solution to $\mathbf{x}$ after searching in $\mathbb{Z}_{\mathbf{p}[d-2]}$, which contains $\mathbf{p}[d-2]$ elements.

Note that $\mathbf{h}_{d-2}$ and deeper layers in the above linear equations can be expanded. The expanded system includes n equations and n variables, with $\mathcal{G}[d-1]$ invertible. However, the generic way of solving exactly-determined system by calculating $\mathcal{G}[d-1]^{-1} * \mathbf{h}$ is no longer applicable to eMLE. The adversary can only proceed by guessing at least one element of $\mathbf{x}$ (i.e., a case discussed above). Since $n \geq 2$ and this element could be a valid one with probability $\frac{2^{(d-1)n}}{\mathbf{p}[d-1]}$, the adversary has a probability $1 - \frac{2^{(d-1)n}}{\mathbf{p}[d-1]}$ to fail to find a solution to other elements of $\mathbf{x}$. □

3 Construction of eMLE-Sig

In this section, we use eMLE to construct the signature scheme eMLE-Sig. This signature scheme is parameterized with five integers: $d > 2$, $n > 2$, c_max, q, and z, and a pseudo-random function $\mathbf{H}$. Optionally, some meta information associated with signatures can also be included as parameters, such as IP addresses/domain names of a server and expiry dates of public keys or signatures. The outputs from $\mathbf{H}$ is required to be less than c_max^2 for the correctness of eMLE-Sig. For example, if $\mathbf{H}$ returns a 256-bit value, then c_max should be at least 2^{128}. These parameters are represented by pp.

With these parameters, we define two auxiliary algorithms, which are used to generate $\mathbf{p}$ and $\mathcal{G}$, respectively. The algorithm $\mathbf{mkP}(pp)$ returns $\mathbf{p}$ by following the definition

Algorithm 1: Generation of $\mathcal{G}$ ($\mathbf{mkG}$)

input : pp
output: $\mathcal{G}$

Let $\mathcal{G}$ be a list of length d
for $i = 0$ **to** $d - 1$ **do**
  for $j = 0$ **to** $n - 1$ **do**
    for $k = 0$ **to** $n - 1$ **do**
      $\mathcal{G}[i][j,k] = \mathbf{H}(pp, i, j, k) + \lfloor \frac{\mathbf{p}[i]}{i*d+k*n+j+1.23} \rfloor \bmod \mathbf{p}[i]$
    end
  end
end
Ensure $\mathcal{G}[d-1]$ is invertible with respect to $\mathbf{p}[d-1]$
return $\mathcal{G}$

below.

$$\mathbf{p}[i] = \begin{cases} 2^z, & \text{if } i = 0 \\ \mathtt{np}(2 * c\_max * \mathbf{p}[i-1]), & \text{if } i = d-1 \\ \mathtt{np}(2 * q * c\_max * \mathbf{p}[i-1]), & \text{otherwise} \end{cases}$$

The algorithm $\mathbf{mkG}(pp)$ is described in Algorithm 1. In eMLE, $\mathcal{G}$ is uniformly sampled at random. However, in eMLE-Sig, each element of a matrix in $\mathcal{G}$ is generated deterministically by adding the pseudo-random or hash value $\mathbf{H}(pp, i, j, k)$ and $\lfloor \frac{\mathbf{p}[i]}{i*d+k*n+j+1.23}) \rfloor$. This way of generating $\mathcal{G}$ reduces the size of public parameters of eMLE-Sig and the burden of generating random numbers. If an optional parameter is not used, the hash value $\mathbf{H}(pp, i, j, k)$ can be removed.

Note that $\mathcal{G}[d-1]$ should be invertible; if not, the first element of the matrix $\mathcal{G}[d-1]$ can be increased in a deterministic way until it is invertible.

Algorithm 2: Key Generation ($\mathbf{keyGen}$)

input : pp
output: $\mathbf{x}$, $\mathbf{h}$

$\mathcal{G} = \mathbf{mkG}(pp), \mathbf{p} = \mathbf{mkP}(pp)$
$\mathbf{t} \leftarrow (1, \mathbf{p}[d-2])^n$
$\mathbf{x} = \mathcal{G}[d-1]^{-1} * \mathbf{t} \bmod \mathbf{p}[d-1]$
Ensure $\mathbf{x} \in (\mathbf{p}[d-2], \mathbf{p}[d-1])^n$
$\mathbf{h} = \mathtt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{x})$
return $\mathbf{x}$, $\mathbf{h}$

The signature scheme eMLE-Sig consists of three algorithms as defined and constructed below.

Key Generation: Given the public parameter pp, this algorithm $\mathbf{keyGen}$ generates the private key $\mathbf{x}$ and the public key $\mathbf{h}$, as defined in Algorithm 2. In the algorithm,

Algorithm 3: Signing ($\mathbf{sign}$)

input : pp, m, $\mathbf{x}$
output: $\mathbf{s}$, $\mathbf{u}$

$\mathcal{G} = \mathbf{mkG}(pp), \mathbf{p} = \mathbf{mkP}(pp)$
$\mathbf{t} \leftarrow (1, \mathbf{p}[d-2])^n$
$\mathbf{y} = \mathcal{G}[d-1]^{-1} * \mathbf{t} \bmod \mathbf{p}[d-1]$
Ensure $\mathbf{y} \in (\mathbf{p}[d-2], \mathbf{p}[d-1])^n$
$\mathbf{u} = \mathtt{eMLE}_{n,d,\mathbf{p},\mathcal{G}}(\mathbf{y})$
$(c, c') = \mathbf{H}(m, \mathbf{u})$
Ensure $c > 0$, $c < c\_max$, $c' > 0$, $c' < c\_max$
$\mathbf{s} = c * \mathbf{x} + c' * \mathbf{y}$
return $\mathbf{s}$, $\mathbf{u}$

we do not directly sample $\mathbf{x}$ from $(\mathbf{p}[d-2], \mathbf{p}[d-1])^n$; instead, we choose a random $\mathbf{t}$ from $(1, \mathbf{p}[d-2])^n$, and then calculate $\mathbf{x}$, such that $\mathcal{G}[d-1] * \mathbf{x} \bmod \mathbf{p}[d-1]$ returns $\mathbf{t}$. This is to optimize the size of the public key $\mathbf{h}$.

Signing: On input pp, $\mathbf{x}$, and message m, the signing algorithm, as defined in Algorithm 3, returns the signature including $\mathbf{s}$ and $\mathbf{u}$. Note that the random vector $\mathbf{y} \in (\mathbf{p}[d-1], \mathbf{p}[d-2])^n$ is generated in the same way as $\mathbf{x}$. From $\mathbf{y}$, the corresponding eMLE value $\mathbf{u}$ is calculated. The pair c and c' is obtained by equally splitting the hash value $\mathbf{H}(m, \mathbf{u})$. If the hash value has an odd number of bits, c can have one more bit. Note that c and c' are less than c_max and both of them are not zero.

Verification: Defined in Algorithm 4, on input pp, public key $\mathbf{h}$, message m, the signature pair $\mathbf{s}$ and $\mathbf{u}$, this algorithm returns $\mathbf{true}$ if the signature can be verified. In this algorithm, c and c' is generated in the same as in the signing algorithm. The algorithm recursively peels off each layer with the *same* $\mathbf{s}$, and checks the remaining value in a certain range. If all checks are passed and the result l is zero at the lowest layer (i.e., $i = d-1$), then $\mathbf{true}$ is returned; otherwise, $\mathbf{false}$ is returned. This algorithm also checks the optional parameter if applicable; for example, the expiry date of signatures or IP address of signers can be checked.

The signature scheme eMLE-Sig is correct in terms that for any key pair $(\mathbf{x}, \mathbf{h}) \leftarrow \mathbf{keyGen}(pp)$, and for any signature $(\mathbf{s}, \mathbf{u}) \leftarrow \mathbf{sign}(pp, m, \mathbf{x})$ of any message m, we have $\mathbf{verify}(pp, \mathbf{h}, m, \mathbf{s}, \mathbf{u}) = \mathbf{true}$. Its correctness is ensured by the conditions on parameters, definition of $\mathbf{p}$, and the structure of eMLE. The formal proof of correctness is omitted and the implementation of eMLE-Sig in Appendix can be used to evaluate the correctness.

3.1 Security Discussion

We will not give the formal security notion of signature schemes [3]. Briefly, eMLE-Sig is secure if an adversary cannot forge a signature for a new message, even if it can access a signing oracle. We do not consider the case on whether a different signature

Algorithm 4: Verification ($\mathbf{verify}$)

input : pp, $\mathbf{h}$, m, $\mathbf{s}$, $\mathbf{u}$
output: true or false

#optionally check meta information in pp (e.g.,expiry date)
$\mathcal{G} = \mathbf{mkG}(pp), \mathbf{p} = \mathbf{mkP}(pp)$
$(c, c') = \mathbf{H}(m, \mathbf{u})$
v = true
for $j = 0$ **to** $n - 1$ **do**
  $v = (v$ and $\mathbf{s}[j] < (c + c') * \mathbf{p}[d - 1]$ and $\mathbf{s}[j] > (c + c') * \mathbf{p}[d - 2])$
  $l = c * \mathbf{h}[j] + c' * \mathbf{u}[j]$
  for $i = 0$ **to** $d - 1$ **do**
    $l = (l - \mathbf{G}[d - i - 1][j] * \mathbf{s})$ mod $\mathbf{p}[d - i - 1]$
    if $d - i - 1 > 0$ **then**
      $v = (v$ and $l < (c + c') * \mathbf{p}[d - i - 2])$
    end
  end
  $v = (v$ and $l = 0)$
end
return v

can be forged for an old message. The security of eMLE-Sig is based on the hardness of eMLE, as stated below.

Theorem 4. *If eMLE is hard, then eMLE-Sig is secure.*

Proof. A signature for a message m includes $\mathbf{u}$ and $\mathbf{s}$, where $\mathbf{u}$ can be generated without knowing the private key. If an arbitrary pair of $\mathbf{u}$ and $\mathbf{s}$ can be verified correctly through all layers, then $\mathbf{s}$ must have the form $\mathbf{s} = c * \mathbf{x} + c' * \mathbf{y}$, where $\mathbf{y}$ can be selected by the adversary.

If a PPT adversary can forge a signature for a new message, then it is able to efficiently generate $\mathbf{s}$ by choosing some $\mathbf{y}$. From $\mathbf{s}$, the secret value $\mathbf{x}$ can be recovered, because the adversary knows $\mathbf{y}$, c, and c'. This is contrary to the condition that eMLE is hard, which means that the adversary needs to exhaustively search at least in $\mathbb{Z}_{\mathbf{p}[d-2]}$, as stated in Theorem 3. □

To forge a signature for a message, the adversary needs to generate $\mathbf{s}$ and $\mathbf{u}$. Without depending on secret values, $\mathbf{u}$ can be generated easily from any $\mathbf{y}$. The above theorem shows that it is not straightforward for the adversary to generate $\mathbf{s}$ when $\mathbf{x}$ is secret. In the following, we use experiments to evaluate the probability of finding a valid $\mathbf{s}$ by exhaustively searching $\mathbf{x}$.

In the experiments, we let $n = 1$ and $d = 3$, such that $\mathbf{x}$ can be searched exhaustively from $\mathbf{p}[d - 2]$ to $\mathbf{p}[d - 1]$. Three parameters z, q, and c_max are configured differently; with each configuration, the test is repeated 100 times.

Based on the condition $l = 0$ in Algorithm 4, the probability should be not bigger than $\frac{1}{\mathbf{p}[0]}$ or $\frac{1}{2^z}$. Furthermore, this probability is reduced due to the range condition on l, which is dependent on the parameter q. However, a bigger c_max makes the range

condition less effective, though it does not affect the condition $l = 0$. Table 2 confirms the probability of finding a valid $\mathbf{s}$ is less than 0.25 when $z = 2$ or less than 0.5 when $z = 1$. We hence use z to indicate the basic security level, with q providing extra security for the configurations later.

z	q	c_max	solutions	searches	probability
1	1	4	1589	12000	0.133
2	1	4	1065	27000	0.040
1	2	4	897	27000	0.033
2	2	4	786	47400	0.017
1	1	16	28415	208600	0.136
2	1	16	25335	407000	0.062
1	2	16	25475	407000	0.063
2	2	16	23859	797400	0.030

Table 2: Probability of Solutions to $\mathbf{s}$ (searches = $100 * (\mathbf{p}[2] - \mathbf{p}[1])$)

3.2 Variants of eMLE-Sig

In current eMLE-Sig, $\mathbf{s}$ is defined as $c*\mathbf{x}+c'*\mathbf{y}$, where $(c, c') = \mathbf{H}(m, \mathbf{u})$. In a variant, multiple secret and blind vectors can be supported, such as $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{y}_1$, $\mathbf{y}_2$. Then, we have

$$\mathbf{s} = c_1 * \mathbf{x}_1 + c_2 * \mathbf{x}_2 + c'_1 * \mathbf{y}_1 + c'_2 * \mathbf{y}_2,$$

where $(c_1, c_2, c'_1, c'_2) = \mathbf{H}(m, \mathbf{u}_1, \mathbf{u}_2)$. The details of such variants will be left for the future work.

4 Performance Evaluation

We have evaluated the hardness of eMLE and security of eMLE-Sig in previous sections. In this section, we evaluate the performance of eMLE-Sig in terms of the sizes of keys and signatures, and provide a set of parameter configurations for security levels above 128 bits.

Compared with all conventional and post-quantum signature schemes, eMLE-Sig might be the easiest to instantiate or configure parameters. The hash algorithm $\mathbf{H}$ basically determines the parameter c_max. Then, only four integer parameters n, d, z, q need to be decided. Among them, z and n rely on the security level, making sure $n \geq 2$ and $n*z$ bigger than the desired security level in bits. d and q can be determined by balancing the sizes of keys and signatures and extra security expectation. There is no need to decide, for instance, suitable prime numbers and cyclic groups as in conventional schemes and the particular lattices in lattice-based post-quantum signatures schemes.

4.1 Sizes of Keys and Signatures

The private key $\mathbf{x}$ contains n elements, each of which has $\log_2(\mathbf{p}[d-1])$ bits. Hence, its size is $n*\log_2(\mathbf{p}[d-1])$ bits. Similarly, the public key has $n*(\log_2(\mathbf{p}[d-2])+1)$ bits. A signature includes $\mathbf{s}$ and $\mathbf{u}$. The size of $\mathbf{s}$ in bits is determined by $n*(\log_2(c\_max)+\log_2(\mathbf{p}[d-1]+1))$, while $\mathbf{u}$ has the size $n*(\log_2(\mathbf{p}[d-2])+1)$ bits.

The calculation above depends on the sizes of $\mathbf{p}[d-2]$ and $\mathbf{p}[d-1]$. These sizes rely on not only parameters z, q,d, c_max, but also the positions of prime numbers, which are not straightforward to estimate. Hence, $\mathbf{p}[d-2]$ has about $z+(d-2)*(\log_2(c\_max)+\log_2(q)+1)$ bits, with $\mathbf{p}[d-1]$ about $z+(d-2)*(\log_2(c\_max)+\log_2(q)+1)+\log_2(c\_max)+1$ bits.

4.2 Configurations

The sizes of keys and signatures of major conventional and PQC schemes have been summarized in [6], where RSA signature scheme has above 380 bytes for keys and signatures. No existing PQC scheme can be more efficient than RSA in terms of both key and signature sizes.

In Table 3, we give several configurations and the corresponding sizes of private key, public key and signature. The sizes are obtained by averaging the sizes from ten experiments. All configurations lead to keys and signatures smaller than RSA, with security level above 128 bits. In all experiments, SHA256 is be used as $\mathbf{H}$, $c\_max = 2^{128}$, and $q = 2^{16}$. The value of c_max ensures that the condition $\mathbf{p}[2]-\mathbf{p}[1] > \mathbf{p}[1]$ required by Theorem 3 is satisfied.

n	d	z	Private Key	Public Key	Signature
2	3	64	84.1	52.2	168.4
2	4	64	120.4	88.5	240.8
2	5	64	156.6	124.7	313.4
3	3	48	120.2	72.3	240.4
3	4	48	174.7	126.7	349.2
4	3	32	152.2	88.5	304.9

Table 3: Parameter Configurations and Sizes from Experiments (Bytes)

For the first configuration in the table, we also calculate the sizes of keys and signatures by analysis; this is to confirm the sizes from experiments roughly match the sizes from analysis with the method in last section.

In this configuration, $\mathbf{p}[d-2]$ and $\mathbf{p}[d-1]$ have about $64+1*(128+16+1)=209$ bits and $209+128+1=338$ bits, respectively. Thus, a private key in this configuration has $2*338=676$ bits (or 84.5 bytes), and the public key has $2*(209+1)=420$ bits (or 52.5 bytes). The size of $\mathbf{s}$ in a signature is about $2*(128+338+1)=934$ bits, and $\mathbf{u}$ has about $2*(209+1)=420$ bits. Hence, a signature has the about size $934+420=1354$ bits (or 169.3 bytes).

With the configurations in Table 3, the correctness of eMLE-Sig is checked by generating and verifying 1000 signatures for each configuration.

5 Related Works

Compared with the conventional or pre-quantum signatures, such as Schnorr signature, ECDSA, and RSA [3], eMLE-Sig makes implementation and configuration much simpler, because it does not need to select particular mathematical structures for its security (e.g., a cyclic group with prime order, a particular curve), or particular prime numbers. eMLE-Sig is instantiated by just specifying a few integer parameters.

The Short Integer Solution (SIS) problem [1] underlies multiple lattice-based signature schemes in NIST PQC standardization. For these schemes, the parameter selection is tricky, because the selection is determined by worst attacks *known so far* and the relation between attacks and parameters might not be straightforward. The layered structure of eMLE can enforce the ways an adversary could take to break eMLE-Sig, hence the security of eMLE-Sig is more certain by relying on the worst attacks that are *possible* to eMLE. On the other hand, the hardness of eMLE-Sig might be enhanced by taking short secret $\mathbf{x}$.

The exiting NP-complete problems has a flat structure in terms that all occurrences of variables are defined with the same modulus, such as the variables in the Boolean satisfiability (SAT) and the variables in LWE problem [5]. Thus, all occurrences of a variable can be manipulated together, such as by regarding a secret value as a coefficient of a lattice basis. However, for eMLE, the same variables occur in multiple layers, each of which has a different modulus; only the occurrences at the top layer can be processed by the adversary. The hardness of eMLE can be increased by adding layers without changing the number of variables and equations; existing NP-complete problems do not have this feature. The relation between NP-complete problems and eMLE could be of independent interest.

Hash-based signatures schemes, such as SPHINCS+ [2], take less hard assumption, but it generates much bigger ciphertexts. eMLE-Sig aims to demonstrate the feasibility of constructing an efficient signature scheme without assuming hard computational problems, since eMLE is a factually hard problem as described in [4]. Two factually hard problems with layered structure are proposed in [4]; however, those layered problems are for public-key encryption, not suitable for constructing signature schemes.

6 Conclusion

In this paper we have proposed the new hard problem Embedded Multilayer Equations (eMLE), with its hardness analyzed and discussed. The hardness of eMLE can be increased by adding more layers without changing the number of equations and variables; no existing NP-complete problem has this feature. The relation between eMLE and existing NP-problems need further investigation as future work.

Based on the new hard problem eMLE, we constructed the signature scheme eMLE-Sig, which addresses the issues of the current post-quantum signature schemes (i.e., issues of big key/ciphertext sizes and security uncertainty). eMLE-Sig has keys and

signatures smaller than pre-quantum RSA scheme. In addition, compared with all existing signature schemes, it might the simplest signature scheme to understand, analyze, instantiate, and implement.

Acknowledgements I wish to expresses my sincere gratitude to Lorez Panny and Dan Brown for their attacks to help improve eMLE-Sig.

References

1. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996. pp. 99–108. ACM (1996)
2. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The sphincs$^+$ signature framework. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 2129–2146. ACM (2019)
3. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman & Hall/CRC (2014)
4. Liu, D., Nepal, S.: Compact-lwe-mqˆ{H}: Public key encryption without hardness assumptions. IACR Cryptol. ePrint Arch. p. 974 (2020), https://eprint.iacr.org/2020/974
5. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005. pp. 84–93. ACM (2005)
6. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: A performance study. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society (2020)
7. Thomae, E., Wolf, C.: Solving underdetermined systems of multivariate quadratic equations revisited. In: Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7293, pp. 156–171. Springer (2012)

Appendix A eMLE-Sig Implementation (Old)

```
# ================================================================================
# eMLE-Sig (Version with attacks found by Lorez Panny and Dan Brown)
# ================================================================================
from hashlib import sha256
set_random_seed(1)

n = 2
d = 3
z = 64
c_max = 2^128
q = 2^16
o = 'expiry date: sep 2021'

def mkP():
  p = vector(ZZ, [0 for _ in range(d)])
  for i in range(d):
    if i==0:
      p[i] = (2^z)
    else:
      if i==d-1:
        p[i] = next_prime(2*c_max*p[i-1])
      else:
        p[i] = next_prime(2*q*c_max*p[i-1])
  return p

def mkG():
  p = mkP()
  G = []
    for i in range(d):
    g = random_matrix(ZZ,n,n,x=1,y=p[i])
    for j in range(n):
      v = str(n+d+z+c_max+q) + str(i)+str(j)
      g[j,:] = vector(ZZ,[(int(p[i]/(i*d+k*n+j+1.23))+
                int(sha256((str(k)+v).encode()).hexdigest()
           , 16) )%p[i] for k in range(n)])
    G = G + [g]
  return G

def eMLE(x):
  p = mkP()
  G = mkG()
  h = vector(ZZ, [0 for _ in range(n)])
  for i in range(d):
    h = (h+G[i]*x)%p[i]
  return h
```

```
def keygen():
  p = mkP()
  G = mkG()
  Rp = Integers(p[d-1])
  gp = G[d-1].change_ring(Rp)
  while true:
    x = gp.inverse()*vector(Rp,[randint(1, p[d-2]) for _ in range(n)])
    x_check = true
    for i in range(n):
      if x[i] <= p[d-2]:
        x_check = false
    if x_check:
      break
  x = x.change_ring(ZZ)
  h = eMLE(x)
  return x, h

def sign(x, m):
  p = mkP()
  G = mkG()
  Rp = Integers(p[d-1])
  gp = G[d-1].change_ring(Rp)

  while true:
    y = gp.inverse()*vector(Rp,[randint(1, p[d-2]) for _ in range(n)])
    y_check = true
    for i in range(n):
      if y[i] <= p[d-2]:
        y_check = false
    if y_check:
      break
  y = y.change_ring(ZZ)
  u = eMLE(y)
  hv = sha256((str(m)+str(u)).encode()).hexdigest()
  l = int(len(hv)/2)
  c = int(hv[0:l], 16)
  cp = int(hv[l:len(hv)], 16)
  c = c%c_max
  cp = cp%c_max
  if (c==0 or cp==0):
    return sign(x, m)
  s = c*x + cp*y
  return u, s

#verification
def verify(m, h, u, s):
  hv = sha256((str(m)+str(u)).encode()).hexdigest()
  l = int(len(hv)/2)
  c = int(hv[0:l], 16)
  cp = int(hv[l:len(hv)], 16)
```

```
  c = c%c_max
  cp = cp%c_max
  p = mkP()
  G = mkG()
  v = true
  for j in range(n):
    v = v and (s[j] < (c+cp)*p[d-1]) and (s[j] > (c+cp)*p[d-2])
    l = c*h[j] + cp*u[j]
    for i in range(d):
      l = (l - G[d-i-1][j]*s)%p[d-i-1]
      if d-i-1>0:
        v = (v and (l < (c+cp)*p[d-i-2]))
    v = v and (l==0)
  return v


x, h = keygen()
count = 10
correct = 0
for i in range(count):
  u, s = sign(x, i)
  r = verify(i, h, u, s)
  if r:
    correct = correct +1
print ("correctness: ", correct, count)
```

Appendix B eMLE-Sig Implementation (New)

```
# ============================================================================
# eMLE-Sig, Date 26 Oct 2021
# Version addressing attacks found by Lorenz Pannyand and Dan Brown
# ============================================================================
from hashlib import sha256

#n >= 2
n,d,z = 2,3,64
#n,d,z = 2,4,64
#n,d,z = 2,5,64
#n,d,z = 3,3,48
#n,d,z = 3,4,48
#n,d,z = 4,3,32

c_max = 2^64
q = 2^16
o = 'expirydate:oct 2021'

def mkP():
    p = vector(ZZ, [0 for _ in range(d)])
    for i in range(d):
        if i == 0:
            p[i] = next_prime(2^z)
        else:
            if i == d-1:
                p[i] = next_prime(5*c_max*p[i-1])
            else:
                p[i] = next_prime(5*q*c_max*p[i-1])
    return p

def mkG():
    p = mkP()
    G = []
    for i in range(d):
        g = random_matrix(ZZ, n, n, x=1, y=p[i])
        for j in range(n):
            v = str(n+d+z+c_max+q) + str(i) + str(j) + o
            g[j,:] = vector(ZZ,[(int(p[i]/(i*d+k*n+j+1.23))+
                     int(sha256((str(k)+v).encode()).hexdigest()
                 ,16))%p[i] for k in range(n)])
        G = G + [g]
    return G

def eMLE(x, F1, F2):
    p = mkP()
```

```
    G = mkG()
    h = vector(ZZ,[0 for _ in range(n)])
    for i in range(d):
        h = (h+F2*G[i]*F1*G[i]*x)%p[i]
    return h

def keygen():
    p = mkP()
    G = mkG()
    pp=prod(p)

    f1 = vector(ZZ,[randint(1, p[0]^(n-1)+1)for _ in range(n)])
    F1 = matrix.circulant(f1)

    fp1 = (randint(1, pp)*f1)%pp

    f2 = vector(ZZ,[randint(1, p[0]^(n-1)+1)for _ in range(n)])
    F2 = matrix.circulant(f2)
    fp2 = (randint(1, pp)*f2)%pp

    x = vector(ZZ,[randint(1, p[0]+1) for _ in range(n)])
    xp = vector(ZZ,[randint(1, p[0]+1) for _ in range(n)])

    h = eMLE(x, F1, F2)
    hp = eMLE(xp, F1, F2)
    return (x,xp, f1, f2), (h, hp, fp1, fp2)

def sign(x, xp, f1, f2, m):
    p = mkP()
    G = mkG()
    pp=prod(p)
    F1 = matrix.circulant(f1)
    F2 = matrix.circulant(f2)

    y = vector(ZZ,[randint(1, p[0]+1) for _ in range(n)])
    u = eMLE(y, F1, F2)

    yp = vector(ZZ,[randint(1, p[0]+1) for _ in range(n)])
    up = eMLE(yp, F1, F2)

    ye = vector(ZZ,[randint(1, c_max*p[0]+1) for _ in range(n)])
    ue = eMLE(ye, F1, F2)

    hv = sha256((str(m)+str(u)+str(up)+str(ue)).encode()).hexdigest()
    l = int(len(hv)/4)
    c0 = int(hv[0:l], 16)
    c1 = int(hv[l:2*l], 16)
    c2 = int(hv[2*l:3*l], 16)
    c3 = int(hv[3*l:len(hv)], 16)
    c0 = c0%c_max
```

```
    c1 = c1%c_max
    c2 = c2%c_max
    c3 = c3%c_max

    if (c0==0 or c1==0 or c2==0 or c3==0):
        return sign(x, xp, f1, f2, m)

    s = F1*F2*(c0*x+ c1*xp + c2*y + c3*yp + ye)
    return u, up, ue, s

# verification
def verify(m, h, u, up, ue, s, fp1, fp2):
    hv = sha256((str(m)+str(u)+str(up)+str(ue)).encode()).hexdigest()
    l = int(len(hv)/4)
    c0 = int(hv[0:l], 16)
    c1 = int(hv[l:2*l], 16)
    c2 = int(hv[2*l:3*l], 16)
    c3 = int(hv[3*l:len(hv)], 16)
    c0 = c0%c_max
    c1 = c1%c_max
    c2 = c2%c_max
    c3 = c3%c_max

    p = mkP()
    G = mkG()
    pp=prod(p)

    Fp1 = matrix.circulant(fp1)
    Fp2 = matrix.circulant(fp2)
    Rp = Integers(pp)
    FF1 = Fp1.change_ring(Rp)
    FF2 = Fp2.change_ring(Rp)

    sp = (FF1.inverse()*s)%pp
    sp = (FF2.inverse()*sp)%pp
    sp = sp.change_ring(ZZ)

    v = True
    for j in range(n):
        v = v and (s[j] < (c0+c1+c2+c3)*n*(p[0]^n+1)*(p[0]+1)*(p[0]^n+1)
                          +n*(p[0]^n+1)*(c_max*p[0]+1)*(p[0]^n+1)
                   ) and (s[j] > (c0+c1+c2+c3+1)*n*2)
        l = c0*h[j] + c1*hp[j] + c2*u[j] + c3*up[j] + ue[j]
        for i in range(d):
            l = (l-(Fp2*G[d-i-1]*Fp1*G[d-i-1])[j]*sp)%p[d-i-1]
            if d-i-1>0:
                v = (v and (l < (c0+c1+c2+c3)*p[d-i-2]))
        v = v and (l==0)
    return v
```

```
(x,xp, f1, f2), (h, hp, fp1, fp2) = keygen()

#correctness tests
count=10
correct=0
for i in range(count):
    u,up,ue, s =sign(x, xp, f1, f2,i)
    r=verify(i,h,u,up, ue, s, fp1, fp2)
    if r:
        correct=correct+1
print("correctness:",correct,count)
```