HTM-PQC: Hardening Cryptography Keys Under the Trend of Post-Quantum Cryptography Migration on Industrial Internet

Lingjia Meng, Yu Fu, Fangyu Zheng, Mingyu Wang, Ziqiang Ma, Jiankuo Dong, and Jingqiang Lin, *Senior Member, IEEE*

***Abstract*—With the rapid expansion of Industry 4.0 technology, the proliferation of large-scale devices faces increasingly severe cyber threats, underscoring the critical importance of cryptographic technology for secure communication and authentication. However, cryptographic systems, as the bedrock of security, have faced a barrage of attacks in recent years, including potential threats from quantum computing and memory disclosure vulnerabilities. In this article, we focus on enhancing the security of two standard quantum-safe cryptographic algorithms, Dilithium and eXtended Merkle signature scheme (XMSS), by leveraging hardware transactional memory (HTM) to create a secure operational environment. Unlike traditional cryptography such as Rivest–Shamir–Adleman (RSA) and elliptic curve cryptography (ECC), Dilithium, and XMSS involve more and larger sensitive variables, rendering conventional solutions inadequate. By conducting a comprehensive sensitivity analysis of variables within the above-mentioned algorithms, we confine sensitive operations to transactional execution regions and employ transaction-splitting technology for efficiency. Our prototype, utilizing Intel transactional synchronization extension (TSX), demonstrates robust protection against memory disclosure attacks with acceptable performance overheads. Notably, our security-enhanced Dilithium and XMSS software implementations, recommended by NIST, achieve an average throughput factor of 0.75 compared to the (unprotected) reference implementations.**

***Index Terms*—Dilithium, hardware transactional memory (HTM), industrial internet, post-quantum cryptography (PQC), eXtended Merkle signature scheme (XMSS).**

Received 29 May 2024; revised 3 August 2024, 9 September 2024, 3 November 2024, and 13 December 2024; accepted 4 January 2025. Date of publication 27 January 2025; date of current version 4 April 2025. This work was supported by the National Natural Science Foundation of China under Grant 61902392 and Grant CCF-AFSG Research Fund under Award CCF-AFSG RF20230206. An earlier version of this article was presented in part at the 2023 26th International Conference Information Security [DOI: 10.1007/978-3-031-49187-0_15]. Paper no. TII-24-2610. *(Corresponding author: Fangyu Zheng.)*

Lingjia Meng is with the School of Cryptology, University of Chinese Academy of Sciences, Beijing 100049, China, and also with Zhongguancun Laboratory, Beijing 100094, China (e-mail: menglj@mail.zgclab.edu.cn).

Yu Fu and Jingqiang Lin are with the School of Cyber Science and Technology, University of Science and Technology of China, Hefei 230026, China (e-mail: fuyu22@mail.ustc.edu.cn; linjq@ustc.edu.cn).

Fangyu Zheng is with the School of Cryptology, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: zhengfangyu@ucas.ac.cn).

Mingyu Wang is with the School of Information Science and Technology, Dalian Maritime University, Dalian 116026, China (e-mail: wangmingyu@dlmu.edu.cn).

Ziqiang Ma is with the School of Information Engineering, Ningxia University, Yinchuan 750021, China (e-mail: maziqiang@nxu.edu.cn).

Jiankuo Dong is with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210023, China (e-mail: djiankuo@njupt.edu.cn).

Digital Object Identifier 10.1109/TII.2025.3528582

I. Introduction

As the Industrial Internet continues to expand and deepen, its applications in intelligent manufacturing, supply chain management, remote monitoring, and automated control are becoming increasingly widespread. Industrial Internet comprehensively connects the production and operation elements such as people, devices, and materials in the industrial field, forming a fundamental information system that influences industrial and economic development. Transitioning from a closed industrial environment to an open Internet network environment, the industrial Internet is additionally facing dual risks brought about by network security. With the rapid development of the industrial Internet, the global Industrial Internet security situation is becoming increasingly severe [1], [2].

Public key cryptography serves as the security foundation of the industrial Internet. The two communicating parties use Diffie–Hellman (DH), Rivest–Shamir–Adleman (RSA), elliptic curve digital signature algorithm (ECDSA), etc., to complete key negotiation, encryption, and digital signature functions. With the rapid development of quantum computers, the real threat posed to the traditional public key cryptographic algorithms represented by DH/RSA/ECDSA has gradually attracted great attention. For this reason, both academia and industry have turned their attention to the research of advanced public-key cryptographic primitives that can resist the threat of quantum computers, namely post-quantum cryptography (PQC).

Recently, the U.S. federal government and the European Commission have urged organizations to transition to PQC [3], [4]. Dilithium and eXtended Merkle signature scheme (XMSS), as their respective outstanding representatives, have been identified as standardized algorithms by national institute of standards and technology (NIST) and internet engineering task force (IETF), respectively. Currently, many researchers tried to use Dilithium

and XMSS for various critical scenarios of the industrial Internet [5], such as the secure boot [6], [7], [8], network access [9], and blockchain service [10].

However, since the industrial Internet is the extension of the traditional Internet, existing cryptographic primitives in the industrial Internet are usually implemented in software form, where cryptographic keys are barely protected, making them vulnerable to different forms of memory disclosure threats such as cold boot attacks [11]. On this basis, attackers can even control the target device, threatening the entire Industrial Internet ecosystem [12].

Existing post-quantum protection studies target mitigating side-channel attacks [13], [14], [15], yet fail to address memory disclosure effectively. Compared with indirect side-channel attacks, memory disclosure attacks enable adversaries to access keys in memory directly, making them more destructive. In this article, we introduce, `HTM-PQC`, which leverages hardware transactional memory (HTM) to protect private keys and all other sensitive values that exist in Dilithium and XMSS signing procedures against memory disclosure attacks. To the best of authors' knowledge, `HTM-PQC` stands as the first work dedicated to safeguarding post-quantum private keys against memory disclosure attacks. The main contributions are as follows.

1) We have seamlessly integrated HTM to safeguard sensitive variables in PQC for the first time, and conducted an in-depth analysis of the variable sensitivity for Dilithium and XMSS, ensuring that the HTM mechanism comprehensively covers all sensitive variables.
2) We have innovatively embraced the concept of "breaking the whole into parts," employing transaction splitting technology to tackle the challenge of an overwhelming number of sensitive variables in Dilithium and XMSS, which achieves refined data protection.
3) We have pioneered the support of both standard and fast modes in our XMSS implementation, striking a balance between performance and memory consumption.
4) We have constructed the prototype based on Intel transactional synchronization extension (TSX) and demonstrated its security against some typical memory disclosure attacks and moderate performance overheads through confirmatory experiments.

II. Preliminaries and Related Work

A. Post-Quantum Cryptography

The development of quantum computers has seriously threatened the existing public key cryptography system. PQC refers to cryptographic algorithms that resist traditional and quantum computer threats. Currently, PQC includes algorithms for two functions: public key encryption/key encapsulation mechanism (KEM) and digital signature.

According to the underlying difficult problems on which the algorithms are based, the mainstream post-quantum cryptographic algorithms can be roughly divided into five categories: lattice-based, hash-based, code-based, multivariate-based, and isogeny-based solutions. Among them, the two most widely used post-quantum cryptosystems at present are lattice-based cryptographic algorithms and hash-based signature schemes, represented by Dilithium and XMSS, which are standardized by NIST and IETF, respectively.

B. HTM-Based Cryptographic Key Protection

Transactional memory is originally proposed to eliminate expensive software synchronization mechanisms, thereby improving the parallel computing's performance. Compared with software-based implementation, HTM usually utilizes hardware resources such as cache or store buffer to implement transactional memory, which is significantly more efficient.

Intel TSX is the most typical HTM instance, which supports instruction set extension for transactional memory based on the caches. Only when a transaction succeeds, the operation results are submitted to the main memory; otherwise, all the performed operations are rolled back. In detail, transactional memory is typically divided into a write-set and a read-set [16], which are deployed in the L1 cache and L3 cache respectively. For a read-set of the transactional execution, the read operation from other threads or processes is allowed while the write operation will make the transaction abort. For the write-set, both the read and write operations from others make the transaction abort.

Cryptographic key protection schemes can be constructed based on HTM. Previous works [17], [18] utilized Intel TSX to achieve a secure cryptosystem, which ensures that sensitive values such as private key only exist as plaintext in the exclusive cache at the core of the process (i.e., the L1D Cache). When private keys participate in the cryptographic computation, a transaction is created, in which the private key is decrypted and the signing operation is performed.

Furthermore, a transaction splitting mechanism is proposed to reduce the time-consuming cryptographic operation within a single transaction, thereby reducing the overhead caused by transaction rollback. Some other schemes [16], [19] leveraged HTM to defend against cache side-channel attacks, which also enhances the robustness of the private key.

C. Secure Implementation Against PQC

With the advancement of PQC standardization, researchers have carried out various security implementations for PQC. Existing security solutions focused on hardware implementations of PQC, mitigating side-channel attacks or fault attacks against several standardized PQC algorithms [13], [14], [15], [20]. However, software-based PQC implementations will remain the primary form of PQC migration for a long time. Compared with side-channel attacks, memory disclosure attacks allow adversaries to obtain keys in memory more directly. Unfortunately, the current landscape lacks research dedicated to countering memory disclosure attacks against PQC software implementations, while our work aims to bridge a critical gap in the field.

III. Challenge and Analysis

A. Technical Challenges

Compared with traditional asymmetric primitives such as RSA and ECDSA, Dilithium and XMSS have a more complicated cryptographic key structure and signature logic. Therefore, when considering the protection of sensitive values such as

Algorithm 1: Dilithium Key Generation.

Output: A public/secret key pair (pk, sk).

1: $\zeta \leftarrow \{0,1\}^{256}$
2: $(\rho, \varsigma, K) = \mathrm{H}(\zeta)$
3: $\mathbf{A} = \mathrm{ExpandA}(\rho)$
4: $(\mathbf{s_1}, \mathbf{s_2}) = \mathrm{ExpandS}(\varsigma)$
5: $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$
6: $(\mathbf{t_1}, \mathbf{t_0}) = \mathrm{Power2Round}(\mathbf{t}, \mathrm{d})$
7: $\mathrm{tr} = \mathrm{H}(\rho \parallel \mathbf{t_1})$
8: **return** $pk = (\rho, \mathbf{t_1})$, $sk = (\rho, K, tr, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$

private keys in the Dilithium and XMSS to achieve secure and efficient signature operations, the following challenges need to be addressed.

1) *More sensitive variables need to be considered and protected:* In RSA and ECDSA, there are limited variables related to the private key. Therefore, they do not need too much space to store these variables. However, more and larger sensitive variables are involved in the Dilithium. Once the values are leaked, the adversary could deduce the long-term secret key. On the other hand, for XMSS, in addition to the risk of private keys being leaked directly or through computation, there may also be a situation of forged signatures due to the leakage of intermediate variables. Therefore, all sensitive variables must be protected against memory disclosure attacks.
2) *Memory control for XMSS is more flexible:* The specific implementation of XMSS can impose more flexible memory control. In detail, the private key can be stored using only the random seed, and the one-time private key for signing will be derived when performing the signature operation, thereby saving private key storage space. Of course, the implementer can also pregenerate all of the one-time private keys to improve the efficiency of signing. Different modes raise distinct implementation issues.
3) *Hardening more complex sensitive operations will result in greater performance overhead:* Placing the Dilithium or XMSS signature operations in an atomic transaction for protection will waste lots of clock cycles and lead to a performance bottleneck. Because the long execution time and huge memory consumption will lead to lots of transaction aborts, the transactions cannot be successfully committed. Therefore, how to improve efficiency is also an important challenge.

B. Variable Sensitivity Analysis for Dilithium

Dilithium [21] has been selected by the NIST PQC standardization organization as the primary signature scheme for standardization. Dilithium offers three different parameter sets, namely Dilithium2, Dilithium3, and Dilithium5, which correspond to three NIST security levels 2, 3, and 5, respectively. In this section, we conduct a detailed analysis of sensitive variables involved in the Dilithium key generation and signing process.

1) Key Generation: The key generation procedure is listed in Algorithm 1. First, a bit string ζ is randomly generated, and placed into the hash function H to derive three seeds, namely ρ, ς, and K. Next, a polynomial matrix $\mathbf{A}$ is generated from seed ρ. Two secret vectors $\mathbf{s_1}$ and $\mathbf{s_2}$ are sampled uniformly from ς. Then, the vector $\mathbf{t} = \mathbf{A}\mathbf{s_1} + \mathbf{s_2}$ is calculated, and split into two parts: high order bits as $\mathbf{t_1}$, which is made public, and low order bits as $\mathbf{t_0}$, which is kept secret. Similarly, matrix $\mathbf{A}$ is also replaced by the small seed ρ, which forms the public key with $\mathbf{t_1}$. Finally, tr is computed by hashing $\rho \parallel \mathbf{t_1}$. Therefore, the output of key generation is the public key $\mathrm{pk} = (\rho, \mathbf{t_1})$ and the secret key $\mathrm{sk} = (\rho, K, \mathrm{tr}, \mathbf{s_1}, \mathbf{s_2}, \mathbf{t_0})$.

We first discuss the public and secret key fields in Dilithium. The public key pk is nonsensitive in the whole scheme, and any participant or even the adversary can obtain it publicly. Similarly, the matrix $\mathbf{A}$ is also public, which is derived from the seed ρ and needs to be regenerated during both signing and verification.

The secret key sk contains six entries. Among them, the seed ρ that appears at the intersection of the public and secret key need not be protected. tr is also nonsensitive since it is a hash of the public key, which does not contain any secret information. Moreover, the vector $\mathbf{t_0}$ in sk is also judged as the public variable, which can be leaked. In addition, the entire learning with errors (LWE) vector $\mathbf{t}$ is publicly available in Dilithium's basic scheme; the purpose of decomposing $\mathbf{t}$ into $\mathbf{t_1}$ and $\mathbf{t_0}$ is only to reduce the size of the public key, not involving security. The rest of the variables in the secret key (i.e., K, $\mathbf{s_1}$, $\mathbf{s_2}$) must be considered as sensitive, which must be protected against the memory disclosure that if leaked, an adversary can use them to forge signatures.

In the key generation process, the seed ς must be regarded as a sensitive variable, since it will derive the long-term secrets $\mathbf{s_1}$ and $\mathbf{s_2}$ directly. Similarly, the variable ζ also needs to be protected, because it can act as an initial seed to generate subsequent secret values, including ς and K. Other values (e.g., $\mathbf{A}$ and $\mathbf{t}$) that appear during the key generation are considered as nonsensitive variables and, thus, can be made public and do not need to be protected against memory disclosure.

2) Signature Generation: Algorithm 2 shows Dilithium's signing process. It takes secret key sk and message M as input. The matrix $\mathbf{A}$ is reconstructed from seed ρ, and M and tr are hashed to create a fixed-length bit string μ. There are two signing options: deterministic (default) and randomized. The difference is in Line 4, where seed ρ' is either produced using μ and K (deterministic) or randomly generated (randomized). The masking vector $\mathbf{y}$ is sampled using ρ' and a rejection counter κ (initially 0). The vector $\mathbf{w} = \mathbf{A}\mathbf{y}$ is decomposed into $\mathbf{w_1}$ and $\mathbf{w_0}$, with $\mathbf{w_1}$ used to compute the challenge $\tilde{c}$. $\tilde{c}$ is converted into a polynomial c, used in calculating $\mathbf{z}$ and $\mathbf{r_0}$ with secret vectors $\mathbf{s_1}$ and $\mathbf{s_2}$. Boundary checks on $\mathbf{z}$ and $\mathbf{r_0}$ ensure signature security and correctness. After passing, the hint $\mathbf{h}$ is calculated as in Algorithm 2, with additional checks on $c\mathbf{t_0}$ and $\mathbf{h}$. If all conditions are met, the signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$ is generated. If not, the process repeats with a new nonce $\mathbf{y}$.

The signing's sensitivity analysis is more complex. At first, μ is a nonsensitive variable, since it is the hash of the public value tr and the message M. The masking vector $\mathbf{y}$ is sensitive. The reason is that the secret vector $\mathbf{s_1}$ can be computed by the formula $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$ (Line 11 in Algorithm 2) when $\mathbf{y}$ is leaked [22], given

Algorithm 2: Dilithium Signature Generation.

Input: Secret key sk and message M.
Output: Signature $\sigma = \text{Sign}(sk, M)$.

$\mathbf{A} = \text{ExpandA}(\rho)$
$\mu = \text{H}(tr \parallel M)$
$\kappa = 0$, $(\mathbf{z}, \mathbf{h}) = \perp$
$\rho' = \text{H}(K \parallel \mu)$ (or $\rho' \leftarrow \{0,1\}^{384}$ for randomized signing)
while $(\mathbf{z}, \mathbf{h}) = \perp$ **do**
  $\mathbf{y} = \text{ExpandMask}(\rho', \kappa)$
  $\mathbf{w} = \mathbf{Ay}$
  $(\mathbf{w_1}, \mathbf{w_0}) = \text{Decompose}(\mathbf{w}, 2\gamma_2)$
  $\tilde{c} = \text{H}(\mu \parallel \mathbf{w_1})$
  $c = \text{SampleInBall}(\tilde{c})$
  $\mathbf{z} = \mathbf{y} + c\mathbf{s_1}$
  $\mathbf{r_0} = \mathbf{w_0} - c\mathbf{s_2}$
  if $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$ *or* $\|\mathbf{r_0}\|_\infty \geq \gamma_2 - \beta$ **then**
    $(\mathbf{z}, \mathbf{h}) = \perp$
  else
    $\mathbf{h} = \text{MakeHint}(\mathbf{r_0}, c, \mathbf{t_0}, \mathbf{w_1}, \gamma_2)$
    if $\|c\mathbf{t_0}\|_\infty \geq \gamma_2$ *or the # of 1's in* $\mathbf{h} > \omega$ **then**
      $(\mathbf{z}, \mathbf{h}) = \perp$
    end
  end
  $\kappa = \kappa + l$
end
return $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$

a valid signature $\sigma = (\tilde{c}, \mathbf{z}, \mathbf{h})$. Therefore, the vector $\mathbf{w}$ must also be protected to prevent the adversary from recovering $\mathbf{y}$ through solving the system of equations: $\mathbf{w} = \mathbf{Ay}$. Considering backward the seed ρ' to obtain the sensitive vector $\mathbf{y}$, it must be identified as the protected variable, whether it is generated deterministically or randomly.

Next, we analysis two composing parts of $\mathbf{w}$ (i.e., $\mathbf{w_1}$ and $\mathbf{w_0}$). The vector $\mathbf{w_1}$ does not contain more information than the signature itself when the signature is generated successfully. In fact, the verifier needs to recalculate $\mathbf{w_1}$ from the signature σ. Therefore, $\mathbf{w_1}$ does not require additional protection. The sensitivity determination about $\mathbf{w_0}$ is similar to $\mathbf{y}$. We directly draw on the theoretical achievements from [23] here, which demonstrates a practical attack leveraging the leakage of $\mathbf{w_0}$ to recover the secret vector $\mathbf{s_2}$. At last, there is a discussion of the signature output and bound checks. The challenge $\tilde{c}$, c and the hint $\mathbf{h}$ are all nonsensitive. The condition judgments about them are for correctness only. However, $\mathbf{z}$ and $\mathbf{r_0}$ must remain protected until the bound checks about them have passed. After that, they do not leak any valid information.

C. Variable Sensitivity Analysis for XMSS

The XMSS scheme [24] is the earliest standardized hash-based signature scheme, which generates signatures by exposing the preimage of one-way functions. The main idea of this scheme is to combine the Merkle tree with the one-time signature (OTS) scheme to construct the many-time signature scheme. RFC 8391 [25] clearly defined the implementation details of the XMSS scheme and provided the reference implementation.

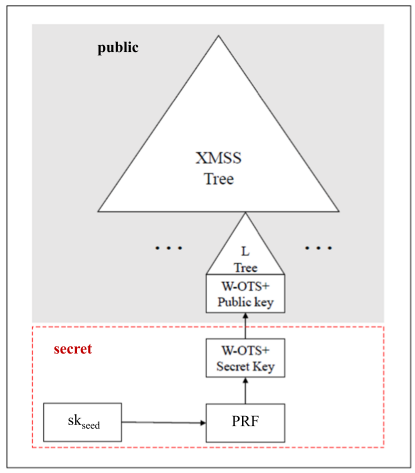

Fig. 1. Sensitivity analysis for XMSS implementation.

Fig. 1 shows the detailed framework of the XMSS. W-OTS+ is the OTS component that constitutes the XMSS, in which the message is signed through the hash chain's construction. The key management of W-OTS+ is realized through the construction of hash trees such as L-tree and XMSS-XOR tree, which are variants of the Merkle tree structure, using the down-top method to form the XMSS authentication tree [26]. It should be noted that XMSS is a stateful hash-based signature scheme, which requires state management of the generated keys to ensure that the keys will not be reused [27]. The specific implementation only needs to ensure that the leaf index of the private keys is incremented, which requires almost no time and space overhead.

The XMSS scheme can be divided into two layers from a structural perspective, as shown in Fig. 1. The core operation of the bottom layer is the W-OTS+ signature generation; the upper layer is the construction of the XMSS tree, including the L-tree and the XMSS-XOR tree. Since the leaf node of the L-tree is the W-OTS+ public key, which is a nonsensitive variable, the XMSS tree part above the leaf node does not contain any sensitive variables. Therefore, the entire tree structure is considered publicly accessible and does not need to be protected. In addition to the private keys, some sensitive variables and sensitive operations will also appear in the XMSS signature generation process.

1) During the signing process, the private variable sk_{seed} needs to be used to generate the actual W-OTS+ private keys through the PRF function. The W-OTS+ private keys are also sensitive. The operations that derive the W-OTS+ private keys from the seed sk_{seed} need to be determined as sensitive operations.
2) XMSS uses the W-OTS+ as the underlying OTS building block. All intermediate variables generated by the W-OTS+ private keys through multiple hash calculations during the W-OTS+ signature process need to be

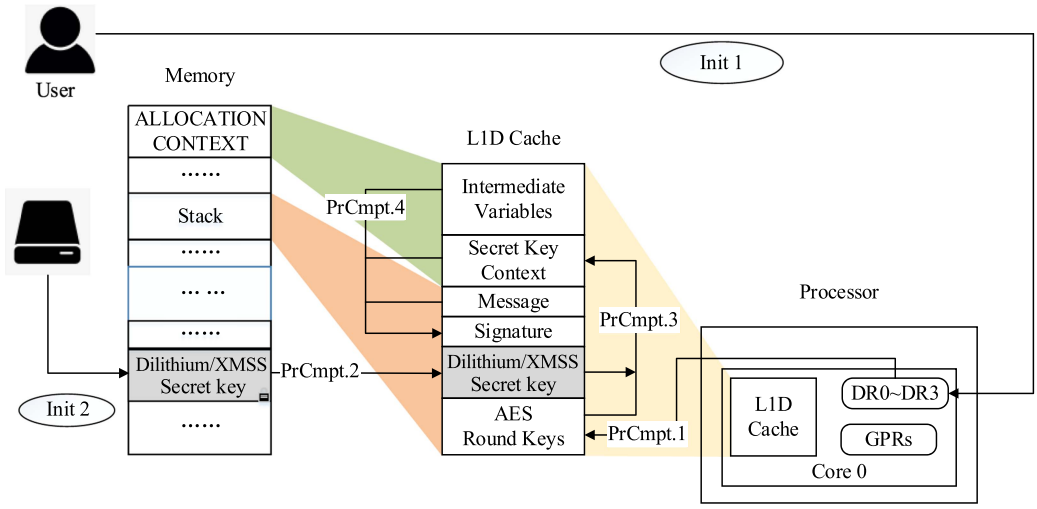

Fig. 2. System architecture of key protection for dilithium and XMSS.

considered sensitive variables. Once these intermediate variables within the hash chain are leaked, the adversary can forge the signature. Therefore, the operations of generating the hash chain also need to be defined as sensitive operations.

3) The XMSS signature generation process needs to generate the corresponding authentication path for the current leaf node. This process involves the generation of other leaf nodes, that is, iteratively calling the key generation function of W-OTS+ to use the private key seed sk_{seed} to generate the W-OTS+ public key as the leaf node of the L-tree. Therefore, these operations also need to be regarded as sensitive.

IV. Design and Implementation

A. Threat Model and Assumptions

First, the adversaries can launch different forms of memory disclosure attacks. Specifically, they can exploit software vulnerabilities to read memory data or launch cold boot attacks through physical access. Second, we assume that HTM can be implemented correctly to provide its claimed security capabilities. We also assume that the OS kernel is trustworthy, thus, users can derive the advanced encryption standard (AES) key securely during the initialization. Finally, since our solution borrows from TRESOR [28] to protect the AES key, it also needs to follow the assumptions made by TRESOR, such as prohibiting system calls from accessing debug registers.

B. System Architecture

`HTM-PQC` adopts the KEK structure, which contains the two-level key structure of "AES key—Dilithium/XMSS secret key." The AES key is generated when the system boots and then stored in the debug registers securely. Sensitive Dilithium or XMSS private key members are symmetrically encrypted offline by using the AES key. When participating in the signature operation, the private key is decrypted. Then, the decrypted private key is utilized to build the signing context and complete the final signature generation during transaction execution. After that, the system cleans up all sensitive intermediate variables and returns the signature result.

The system's architecture is shown in Fig. 2, along with the workflow. `HTM-PQC`'s operation can be divided into two phases, namely the *initialization phase* and the *protection computing phase*. The initialization phase is performed only once during the system initialization, responsible for initializing the AES key and preparing computing resources. The protection computing phase is invoked on each Dilithium or XMSS signing request and used to execute Dilithium or XMSS's signature generation task. All sensitive operations during the protection computing phase are protected by transaction memory and do not reveal any secret values.

Initialization phase: This phase consists of two steps.

1) *Init.1*: During the system boot, the user enters a password to derive the AES key, which is then stored in privileged debug registers of each CPU core. All intermediate variables need to be erased.
2) *Init.2*: The file containing the encrypted Dilithium or XMSS private key is loaded from the hard disks into the main memory. The private key is generated in a secure offline environment and then symmetrically encrypted with the AES key.

Protection computing phase: In the protection computing phase, our solution creates the transaction execution environment for Dilithium or XMSS signing operations, in which the secret key is decrypted. Then, the decrypted key participates in the signing process. Our solution will perform the following steps.

1) *Prepare*: HTM starts tracking memory accesses in the cache (maintaining the read/write sets).
2) *PrCmpt.1*: Loading the AES key to the cache from the debug registers, the round keys are then derived in the L1D cache.
3) *PrCmpt.2*: Loading the ciphertext Dilithium or XMSS secret key to the L1D cache from the memory.

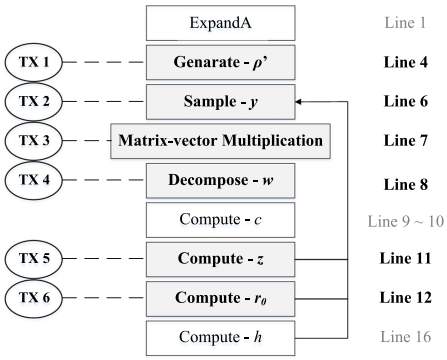

Fig. 3. Transaction splitting for dilithium signing. The rectangles in the figure represent the basic operations in the Dilithium signature scheme. The shaded ones represent sensitive operations that need to be placed in the transaction. The mark (e.g., Line 4) on the right side of the figure indicates the specific line number in Algorithm 2 corresponding to the operation.

4) *PrCmpt.3*: Using the AES key to decrypt the secret key and generate the private key context.
5) *PrCmpt.4*: Using the private key context for requesting signing operations. The details will be covered in the following part.
6) *PrCmpt.5*: Clearing all sensitive variables that exist in registers and cache.
7) *Commit*: Completing the signing process and returning the signature results.

C. Transaction Execution and Splitting

Empirically, putting the entire signature generation procedure into a single transaction is almost impossible to realize. Since it involves plenty of time-consuming and memory-intensive operations such as polynomial multiplication, hash chain generation, and so on. We have also tried to implement this situation, and the result has unsurprisingly failed.

In order to provide an effective signing service with HTM, we must consider breaking the Dilithium or XMSS signing procedure into several independent sensitive operations and placing them in different small transactions. Nonsensitive operations simply run in a normal manner.

1) Dilithium: Based on the variable sensitivity analysis for Dilithium mentioned above, we determined the six sensitive operations in Dilithium signing logic (that is, the operations that take sensitive variables as operation objects or operation results). For each sensitive operation, we need to build the atomic transaction. Fig. 3 shows the transaction splitting for Dilithium signing. Therefore, Dilithium signing is split into six transactions as follows.

TX1: Generation of the seed ρ'.
TX2: Sampling of temporary secret vector $\mathbf{y}$.
TX3: The matrix-vector multiplication $\mathbf{w} = \mathbf{Ay}$.
TX4: Decomposing operation against the vector $\mathbf{w}$.
TX5: Computation of $\mathbf{z}$, which involves the private $\mathbf{s_1}$ and secret vector $\mathbf{y}$.
TX6: Computation of $\mathbf{r_0}$ using the private $\mathbf{s_2}$ and secret vector $\mathbf{w_0}$.

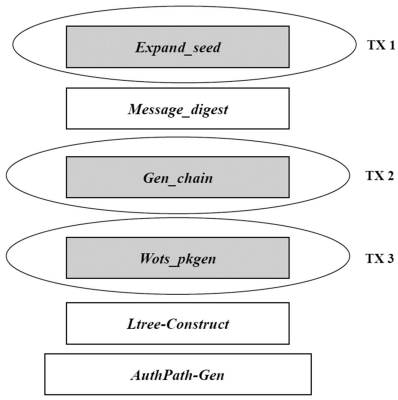

Fig. 4. Transaction splitting for the standard version of XMSS.

2) Standard Version of XMSS: Similarly, for XMSS standard mode, three sensitive operations involved in the XMSS signature generation are placed in different atomic transactions, which is shown in Fig. 4. In addition, the remaining operations are nonsensitive and do not need to be placed in transactions, such as calculating message digests and using leaf nodes to generate authentication paths. They can run in nontransactional memory mode. The three specific transactions involved in the standard version of XMSS include the following.

TX1: The operation of deriving an XMSS one-time private key from a private key seed, denoted by *Expand_seed*.
TX2: W-OTS+ signature operation, denoted by *Gen_chain*.
TX3: The key generation operations required to generate leaf nodes of the authentication path, denoted by *Wots_pkgen*.

3) Fast Version of XMSS: In addition to the standard version of using the seed sk_{seed} to derive W-OTS+ private keys at runtime, we also provide the fast version of XMSS implementation, that is, instead of introducing the random seed, all W-OTS+ private keys are generated in advance. At this time, it is no longer necessary to iteratively call W-OTS+ key generation when generating an authentication path for the current leaf node, which can improve the performance of signing operations. In this mode, sensitive variables include all one-time W-OTS+ private keys and all intermediate variables that appear in the W-OTS+ signing process.

In the XMSS fast mode, since the system is provided all W-OTS+ private keys, the only sensitive operations that need to be placed in the transactional memory for protection are the hash chain generation operations involved in the W-OTS+ signature generation procedure.

Using Dilithium[1] and XMSS's reference implementation[2] as a basis, we provide security enhancements for Dilithium and XMSS's signing, respectively. All AES encryption and decryption operations used in the system are implemented using AES-NI hardware instructions. We cannot place the entire signature logic in a single transaction. Therefore, we will divide the signing operation into several subprocesses based on the previous analysis of sensitive operations and place them in atomic

[1][Online]. Available: https://github.com/pq-crystals/dilithium
[2][Online]. Available:https://github.com/XMSS/xmss-reference

transactions, respectively. Since the abort may still occur, we will repeatedly call `_xbegin` in a loop to start the transaction, and only when the transaction is successfully committed or the number of failures reaches the threshold, the loop will exit. In addition, intermediate sensitive variables need to be encrypted using the AES key before the small transaction commits.

D. System Implementation

We implement `HTM-PQC` as a char module and integrate it into the Linux kernel. The module provides secure Dilithium or XMSS signing services to user space. It depends on the `ioctl` system call to receive the messages to be signed and the private key in the form of ciphertext from the user-space invoker, complete the signing operation, and then return the signature to the user application.

`HTM-PQC` chooses Intel TSX, the most representative and widely used instance of HTM, to build the transactional execution environment for Dilithium or XMSS's signing against memory disclosure attacks. Specifically, we choose the restricted transaction memory primitive as the programming interface of HTM, where the `_xbegin` function is called to start a transaction, and the `_xend` function is used to commit the transaction. The area between the two functions is the atomic region of transactional execution. Of course, our solution applies to HTM features on other platforms. However, there is basically no difference in their design principle, and therefore, in terms of overheads.

Dilithium or XMSS's signing procedure is time-consuming, and it will be interrupted by frequent context switches, which causes the transaction abort. Therefore, it is necessary to disable interrupts and kernel preemption in the transactional region to improve the success rate of transaction commit. In addition, we refer to the general method in TRESOR [28] to derive and protect the AES key.

V. Evaluation and Discussion

A. Experimental Setup

The experimental platform is Intel Core i7-6700 CPU with 3.4 GHz, 16 GB memory, and the operating system is Ubuntu 16.04 64-bit with a 4.15.0 kernel version.

B. Performance

1) Dilithium: We first take the most recommended parameter set Dilithium-3 as the instance and measure the time consumed by each operation with or without Intel TSX. In detail, we put each sensitive operation into the transaction or run it directly, and record the average clock cycles it consumes. The results are shown in Fig. 5.

In order to evaluate the overall efficiency of `HTM-PQC` in executing Dilithium, we repeated the Dilithium signing 1000 times and then selected the average value (in milliseconds) as the evaluation indicator. The reason why we chose 1000 times as the parameter in the performance evaluation is that we followed the parameters selected in the Dilithium official code library when measuring the signature speed, which is also the sample

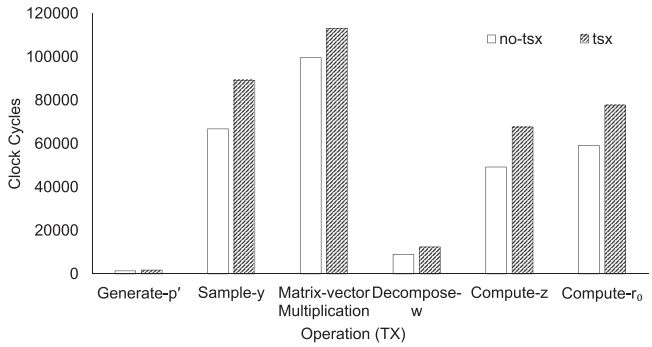

Fig. 5. Performance of sensitive operations in Dilithium-3.

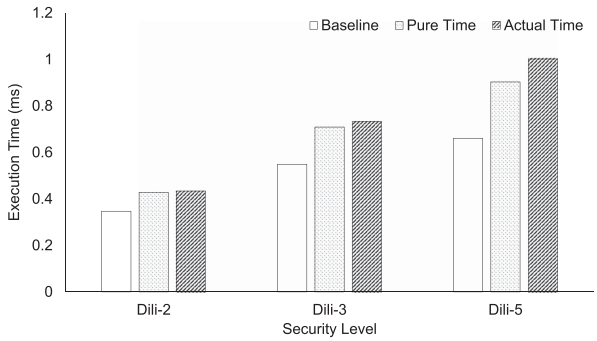

Fig. 6. Performance comparison between reference implementation (baseline) and our solution (`HTM-PQC`).

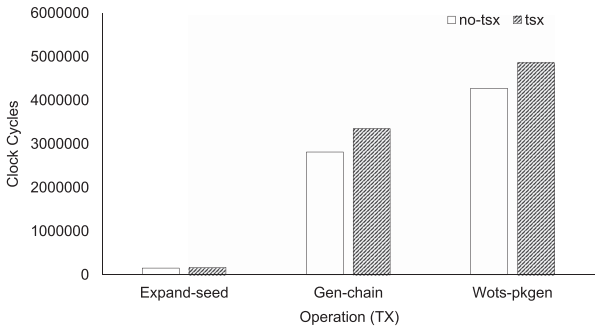

Fig. 7. Performance of TXs in XMSS instantiated with SHAKE-256.

code submitted to NIST. Fig. 6 shows the times of the reference implementation of Dilithium (baseline) and `HTM-PQC` under three different security levels. Since transactions have a certain probability of successful commits (i.e., success rate), we use the actual time to characterize the performance of `HTM-PQC`, that is, the pure time divided by the success rate. For Dilithium-2, Dilithium-3, and Dilithium-5, `HTM-PQC` achieves a factor of 0.80, 0.75, and 0.66 compared to the throughout of baseline, respectively.

2) XMSS: Similarly, we chose SHAKE-256 to instantiate XMSS and compare the execution time of sensitive operations with and without Intel TSX, as shown in Fig. 7.

Furthermore, we selected the XMSS instance with a tree height of 10, a parameter set recommended by the reference

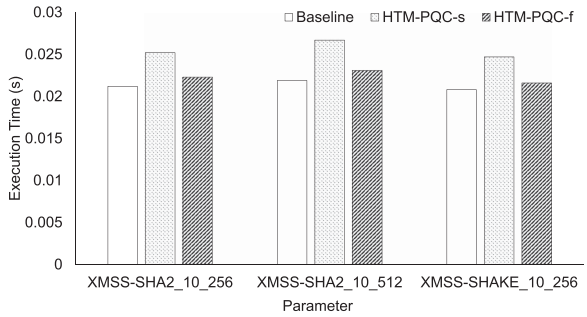

Fig. 8. Performance comparison between reference implementation (baseline) and security enhanced XMSS solution.

implementation, and instantiated it using different hash primitives. We executed the XMSS signing 1000 times iteratively and used the average value of the time consumed by the signing (in seconds) as the final performance indicator. The specific performance results of the XMSS instances are shown in Fig. 8. As we can see, for XMSS-SHA2_10_256, the signature throughput of the standard version of security-enhanced XMSS implementation (denoted as HTM-PQC-s) reached 0.84 times that of the baseline, while the signature throughput of the fast version (denoted as HTM-PQC-f) reaches 0.95 times that of the baseline. Other parameter sets show similar results.

We briefly analyze the experimental results. The operations that can introduce performance overhead in our solution include the following:

1) the transmission of messages, encrypted private keys, and signature results between the user space and the kernel space;
2) the encryption and decryption operations performed on sensitive variables in the single transaction; and
3) the transaction's rollback operation.

Among these, the second factor introduces the highest overhead, accounting for about 80%. Despite our ultrafine-grained control over the boundaries of sensitive variables, Dithium and XMSS still contain plenty of sensitive variables, and the encryption and decryption operations between transactions still bring about considerable overhead. Meanwhile, disabling interrupts and kernel preemption in kernel space and the transaction's atomic execution will also reduce the frequency of context switching during the signing process and improve performance to a certain extent.

For Dilithium, When the security level is 2 or 3, every small transaction will be submitted with a higher success rate without too many retries. Therefore, the performance overhead introduced by our solution is lower. When the security level reaches 5, the sensitive values will occupy more space, thereby increasing the probability of transaction abort. Plenty of clock cycles are consumed in the transaction's rollback operation, therefore, greater performance overhead will be introduced.

For XMSS, no matter which parameter for instantiation, either standard or fast version, all transactions in HTM-PQC can be committed successfully due to transactional splitting. The XMSS signing process involves thousands of hash operations, most of which need to be protected by HTM, therefore, the standard version will have a 16% performance loss. The fast version, however, has almost the same overhead as the reference implementation because all private keys are generated in advance.

Performance improvement: Admittedly, HTM-PQC does introduce a relatively high performance overhead due to the comprehensive protection of all sensitive values that appear during the signing procedure. This article has explored an optimal transactional splitting method to increase the probability of transaction commit. However, the memory consumption of the internal signature algorithm will also affect the probability of a successful transaction commit. In future work, we will focus on optimizing the implementation of the signature algorithm. Specifically, we will try to minimize the probability of transaction rollbacks by reducing the algorithm's memory consumption, for example, by preloading the precalculated table of number theoretic transform (NTT) in Dilithium into the YMM registers.

C. Security

1) Resistance to Memory Disclosure Attacks: Our solution relies on HTM to provide security enhancements. First, all sensitive operations in the PQC signature process are running in atomic transactions, and the whole sensitive variables are encrypted outside the transactions. During cryptographic operations (i.e., transactional execution), other threads cannot obtain the private key or intermediate sensitive variables in plaintext except the PQC signature computing thread. Therefore, it can defend against software memory attacks. Second, due to the implementation characteristics of HTM, the entire signing operation is limited to the write set of the transaction. Sensitive values only exist in the L1D Cache and will not appear in the RAM chips, thus cold-boot attacks and DMA attacks can also be resisted.

More practically, we launch memory dumplication and DMA attacks, respectively, to demonstrate the effectiveness of HTM-PQC against typical memory disclosure attacks. Dilithium signing is performed in a loop. For memory dump attacks, we use the lime tool to dump the memory image. For DMA attacks, we insert the LeetDMA device into the host through the PCIE interface and depend on the open source project pcileech to control LeetDMA to obtain the memory image. Then, we use hexdump to look for known private key fragments in the memory image and cannot obtain a binary string that overlaps for more than 4 B with the private key. In terms of quantitative attack results, memory disclosure attacks are not evaluated by the key recovery ratio like side-channel attacks. They are more direct and destructive attacks. Therefore, for unprotected implementations, the attack can obtain all (i.e., 100%) of the private keys; while for HTM-PQC, the private keys cannot be recovered.

2) Resistance to Side-Channel Attacks: Hardware cryptographic implementations are vulnerable to fault attacks and side-channel attacks [29], [30]. However, as a software PQC cryptographic implementation, it is relatively difficult to launch

conventional physical attacks against HTM-PQC, such as template attacks and fault injection attacks. In recent years, a variety of advanced side-channel attacks have threatened software implementations, namely timing attacks and cache side-channel attacks [31], [32]. We analyze from two perspectives: the PQC algorithm itself and its implementation.

1) Considering the characteristics of the algorithm itself, the PQC algorithm we chose for security enhancement is inherently resistant to side-channel analysis. Specifically, Dilithium adopts the design of uniform sampling, and all other basic operations such as polynomial multiplication and rounding are implemented in constant time in the reference implementation. For XMSS, due to the random nature of the hash function, the computations in XMSS have very little dependency on the key that can be attacked using side channels.
2) The hardware characteristics that HTM-PQC implementations rely on can also mitigate side-channel attacks. First, the encryption and decryption operations related to the private keys and other sensitive values are all implemented using AES-NI, which is resistant to timing attacks. Moreover, HTM-PQC performs the Dilithium or XMSS signing operations in HTM-backed transactions and ensures that all sensitive data resides in the L1D Cache during the execution. This prevents the attackers from distinguishing the timing differences between cache hits and misses accurately, thereby mitigating cache side-channel attacks.

3) Resistance to Fault Attacks and SCA-Combined Attacks: According to our investigation, existing fault attacks and SCA-combined attacks against PQC predominantly target hardware implementations [7], [29], [30], [33]. These attacks extract sensitive values such as cryptographic keys by actively tampering with the device, such as hitting the circuitry with a laser or undervolting the device voltage during (part of) the computation process. However, launching fault attacks against software implementations is an enormous challenge. Even if a malicious attacker manages to launch fault injection attacks, HTM's inherent integrity protection and rollback capabilities can naturally counteract them. Any attempt by an attacker to manipulate or inject values triggers a write conflict within the transaction, leading to its rollback, thus preventing the disclosure of any sensitive values.

D. Discussion

1) Scalability to Other PQC KEM and Signature Schemes: This article currently only instantiates two standard PQC schemes, Dilithium and XMSS. However, HTM-PQC is fundamentally extensible to any PQC KEM and signature scheme. Developers should perform a thorough sensitivity analysis on specific PQC cryptographic schemes to determine sensitive variables and operations, and then provide security-enhanced signature services through HTM. In the lattice-based PQC schemes, NTT, as an important performance factor, also needs to be considered. If NTT is related to sensitive values, such as NTT transformation of private keys, then these NTTs also need to be protected by HTM. According to our evaluation, our solution has a limited impact on NTT performance.

2) Compatibility With Different Platforms: Although we have implemented the prototype with Intel TSX, HTM-PQC applies to any processor platform that supports HTM. For example, both ARM TME [34] and AMD ASF [35] support HTM capabilities and provide similar instructions to specify transactional memory regions. Specifically, ARM TME adopts TSTART and TCOMMIT instructions to start and commit transactions, respectively. Similarly, the corresponding instructions supported by AMD ASF are SPECULATE and COMMIT. Therefore, replacing the relevant instructions, our solution demonstrates adaptive compatibility with ARM and AMD platforms.

3) Comparison With Trusted execution environment (TEE)-Based Solutions: TEE [36] extensions build the secure zone by isolating software and hardware resources, thereby ensuring the security of applications and data. TEE-based solutions are also applicable to PQC private key protection in the industrial Internet scenarios. Moreover, since its encryption logic is generally implemented by hardware directly, it has certain performance advantages over HTM-PQC. However, TEE-based solutions have limitations in terms of security and portability. From the security perspective, TEE-based solutions are vulnerable to widespread software side-channel attacks [37], especially transient execution attacks. From the portability perspective, since the programming interfaces and specifications of TEE extensions under different platforms vary greatly, TEE-based solutions require extremely complex code reconstruction. However, HTM instructions under different processor platforms are relatively similar and the development is relatively simple, making it easier to migrate and deploy across platforms.

4) Power Assumption: HTM-PQC is a software module, rather than a hardware implementation. The power consumption of running it on the machine is also a crucial metric to measure its real-world applicability. We exploited the tool powerstat to measure the machine's power consumption, which depends on the battery stats or the Intel RAPL interface. The average power consumption of performing the unprotected signing operations is 36.90 W, and when running the HTM-PQC, the growth rate of power consumption ranges from −1.2% (36.45 W) to 3.4% (38.16 W). Therefore, our approach can harden PQC with negligible additional power consumption.

E. Limitation

Our prototype has demonstrated the practicality of HTM-PQC. When deploying HTM-PQC in the industrial Internet scenarios, developers only need to compile and install the kernel driver and then invoke it for signature generation. However, a potential limitation is that the reliance on Intel TSX as the HTM instance may require additional efforts for porting to other platforms, although it is fundamentally feasible.

Another limitation is that HTM-PQC experiences performance degradation, approximately 25% compared to native implementations. However, since our solution targets security-sensitive Industrial Internet applications, this performance reduction is considered acceptable.

F. Future Work

In the future, we will consider extending HTM-PQC's security-enhanced capabilities to more signature schemes. We will select two representative PQC schemes as the priority targets, namely Raccoon [38], [39], a candidate of NIST PQC Round 1 Additional Signatures Standardization, and SPHINCS+[40], which NIST has also standardized.

1) Raccoon is also a lattice-based signature scheme with a similar signature structure to Dilithium. Compared to Dilithium, Raccoon improves the ability to resist side-channel attacks by adopting no rejection sampling and sums of uniform distributions [38]. Therefore, Raccoon has the same sensitive variables as Dilithium, such as the masking vector $\mathbf{y}$ and the secret vector $\mathbf{w}$. However, it is necessary to redefine sensitive operations according to the specific Raccoon signature process.
2) Unlike XMSS, SPHINCS+ consists of multiple smaller Merkle trees, presenting a hierarchical structure. The upper Merkle tree is used to sign the subtree. Any malicious message can be signed if the attacker chooses the middle-level W-OTS+ as the attack target. Once the signature is forged, the attacker can arbitrarily construct a complete tree that can be used to sign a limited number of arbitrary messages. Therefore, the W-OTS+ signature operations on the entire hierarchical SPHINCS+ tree need to be protected using HTM.

VI. Conclusion

PQC schemes, represented by Dilithium and XMSS, are expected to see widespread adoption in the Industrial Internet. However, their private keys remain vulnerable to various forms of memory disclosure attacks, which existing works have not adequately addressed. In this article, we conduct a comprehensive analysis of all sensitive variables involved in the signing processes of Dilithium and XMSS. Based on this analysis, we propose HTM-PQC, a novel protection mechanism. We implemented a prototype of HTM-PQC using Intel TSX, and evaluation results demonstrate that its efficiency is comparable to reference implementations. Our work aims to inspire more robust implementations of PQC in the industrial Internet, thereby advancing the security ecosystem.

References

[1] J. Beerman, D. Berent, Z. Falter, and S. Bhunia, "A review of colonial pipeline ransomware attack," in *Proc. IEEE/ACM 23rd Int. Symp. Cluster Cloud Internet Comput. Workshops*, 2023, pp. 8–15.

[2] N. Tuptuk and S. Hailes, "Identifying vulnerabilities of industrial control systems using evolutionary multiobjective optimisation," *Comput. Secur.*, vol. 137, 2024, Art. no. 103593.

[3] D. Joseph et al., "Transitioning organizations to post-quantum cryptography," *Nature*, vol. 605, no. 7909, pp. 237–243, 2022.

[4] C. Näther, D. Herzinger, S.-L. Gazdag, J.-P. Steghöfer, S. Daum, and D. Loebenberger, "Migrating software systems towards post-quantum-cryptography—A systematic literature review," 2024, *arXiv:2404.12854*.

[5] K.-A. Shim, "On the suitability of post-quantum signature schemes for Internet of Things," *IEEE Internet Things J.*, vol. 11, no. 6, pp. 10648–10665, Mar. 2024.

[6] A. Wagner, F. Oberhansl, and M. Schink, "To be, or not to be stateful: Post-quantum secure boot using hash-based signatures," in *Proc. Workshop Attacks Solutions Hardware Secur.*, 2022, pp. 85–94.

[7] A. Wagner, V. Wesselkamp, F. Oberhansl, M. Schink, and E. Strieder, "Faulting Winternitz one-time signatures to forge LMS, XMSS, or signatures," in *Proc. Int. Conf. Post-Quantum Cryptogr.*, 2023, pp. 658–687.

[8] P. Kampanakis, P. Panburana, M. Curcio, and C. Shroff, "Post-quantum hash-based signatures for secure boot," in *Proc. Silicon Valley Cybersecurity Conf.*, 2021, pp. 71–86.

[9] D. Ghinea et al., "Hybrid post-quantum signatures in hardware security keys," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, 2023, pp. 480–499.

[10] H. Gharavi, J. Granjal, and E. Monteiro, "Post-quantum blockchain security for the Internet of Things: Survey and research directions," *IEEE Commun. Surv. Tut.*, vol. 26, no. 3, pp. 1748–1774, Third Quarter 2024.

[11] M. R. Albrecht, A. Deo, and K. G. Paterson, "Cold boot attacks on ring and module LWE keys under the NTT," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2018, no. 3, pp. 173–213, 2018.

[12] N. Mishra, S. H. Islam, and S. Zeadally, "A survey on security and cryptographic perspective of industrial-internet-of-things," *Internet Things*, vol. 25, 2023, Art. no. 101037.

[13] V. Migliore, B. Gérard, M. Tibouchi, and P.-A. Fouque, "Masking dilithium: Efficient implementation and side-channel evaluation," in *Proc. Appl. Cryptogr. Netw. Secur.: 17th Int. Conf.*, 2019, pp. 344–362.

[14] A. Jati, N. Gupta, A. Chattopadhyay, and S. K. Sanadhya, "A configurable CRYSTALS-Kyber hardware implementation with side-channel protection," *ACM Trans. Embedded Comput. Syst.*, vol. 23, no. 2, pp. 1–25, 2024.

[15] A. Genêt, "On protecting SPHINCS+ against fault attacks," *Cryptology ePrint Arch.*, 2023.

[16] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Proc. 26th USENIX Secur. Symp. Secur. 17*, 2017, pp. 217–233.

[17] C. Li et al., "Mimosa: Protecting private keys against memory disclosure attacks using hardware transactional memory," *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 3, pp. 1196–1213, May/Jun. 2021.

[18] L. Meng et al., "Protecting private keys of Dilithium using hardware transactional memory," in *Proc. Int. Conf. Inf. Secur.*, 2023, pp. 288–306.

[19] R. Zhang, M. D. Bond, and Y. Zhang, "Cape: Compiler-aided program transformation for HTM-based cache side-channel defense," in *Proc. 31st ACM SIGPLAN Int. Conf. Compiler Construction*, 2022, pp. 181–193.

[20] A. Shaller, L. Zamir, and M. Nojoumian, "Roadmap of post-quantum cryptography standardization: Side-channel attacks and countermeasures," *Inf. Comput.*, vol. 295, 2023, Art. no. 105112.

[21] L. Ducas et al., "CRYSTALS-Dilithium: A lattice-based digital signature scheme," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2018, no. 1, pp. 238–268, 2018.

[22] S. Marzougui, V. Ulitzsch, M. Tibouchi, and J.-P. Seifert, "Profiling side-channel attacks on Dilithium: A small bit-fiddling leak breaks it all," *Cryptology ePrint Arch.*, 2022.

[23] A. Berzati, A. C. Viera, M. Chartouni, S. Madec, D. Vergnaud, and D. Vigilant, "A practical template attack on CRYSTALS-Dilithium," *Cryptology ePrint Arch.*, 2023.

[24] J. Buchmann, E. Dahmen, and A. Hülsing, "XMSS-a practical forward secure signature scheme based on minimal security assumptions," in *Proc. Post-Quantum Cryptogr.: 4th Int. Workshop*, 2011, pp. 117–129.

[25] A. Huelsing, D. Butin, S.-L. Gazdag, J. Rijneveld, and A. Mohaisen, "XMSS: EXtended Merkle signature scheme," May 2018, doi: 10.17487/RFC8391.

[26] L. Li, X. Lu, and K. Wang, "Hash-based signature revisited," *Cybersecurity*, vol. 5, no. 1, pp. 1–26, 2022.

[27] L. Groot Bruinderink and A. Hülsing, "'Oops, i did it again'–security of one-time signatures under two-message attacks," in *Proc. Int. Conf. Sel. Areas Cryptogr.*, 2017, pp. 299–322.

[28] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *Proc. 20th USENIX Secur. Symp. Secur.*, 2011, Art. no. 17.

[29] P. Ravi, A. Chattopadhyay, J. P. D'Anvers, and A. Baksi, "Side-channel and fault-injection attacks over lattice-based post-quantum schemes (Kyber, Dilithium): Survey and new results," *ACM Trans. Embedded Comput. Syst.*, vol. 23, no. 2, pp. 1–54, 2024.

[30] J. P. Thoma, D. Hartlief, and T. Güneysu, "Agile acceleration of stateful hash-based signatures in hardware," *ACM Trans. Embedded Comput. Syst.*, vol. 23, no. 2, pp. 1–29, 2024.

[31] J. Jancar et al., "'They're not that hard to mitigate': What cryptographic library developers think about timing attacks," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 632–649.
[32] S. Huang, R. Q. Sim, C. Chuengsatiansup, Q. Guo, and T. Johansson, "Cache-timing attack against HQC," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2023, no. 3, pp. 136–163, 2023.
[33] Z. Ni, A. Khalid, W. Liu, and M. O'Neill, "Bitstream fault injection attacks on CRYSTALS kyber implementations on FPGAs," in *Proc. Des., Autom. Test Eur. Conf. Exhib.*, 2024, pp. 1–6.
[34] C. Piatka, R. Amslinger, F. Haas, S. Weis, S. Altmeyer, and T. Ungerer, "Investigating transactional memory for high performance embedded systems," in *Proc. Architecture Comput. Syst.: 33rd Int. Conf.*, 2020, pp. 97–108.
[35] J. Chung et al., "ASF: AMD64 extension for lock-free data structures and transactional memory," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2010, pp. 39–50.
[36] P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Trusted execution environments: Properties, applications, and challenges," *IEEE Secur. Privacy*, vol. 18, no. 2, pp. 56–60, Mar./Apr. 2020.
[37] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang, "A systematic look at ciphertext side channels on AMD SEV-SNP," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 337–351.
[38] R. Del Pino et al., "A side-channel secure signature scheme," 2023. [Online]. Available: https://raccoonfamily.org
[39] R. del Pino, S. Katsumata, M. Maller, F. Mouhartem, T. Prest, and M.-J. Saarinen, "Threshold raccoon: Practical threshold signatures from standard lattice assumptions," in *Proc. Adv. Cryptology–EUROCRYPT 2024*, 2024, pp. 219–248.
[40] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, and P. Schwabe, "The SPHINCS signature framework," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 2129–2146.

Lingjia Meng received the B.E. degree in information security from the Huazhong University of Science and Technology, Wuhan, China, in 2017 and the Ph.D. degree in cyberspace security from the University of Chinese Academy of Sciences, Beijing, China, in 2024.

He is currently an Assistant Research Fellow with Zhongguancun Laboratory, Beijing, China. His research interests include system security and applied cryptography.

Yu Fu received the M.S. degree in cyberspace security from the University of Chinese Academy of Sciences, Beijing, China, in 2022. He is currently working toward the Ph.D. degree in cyberspace security with the School of Cyber Science and Technology, University of Science and Technology of China, Hefei, China.

His research interests include applied cryptography and privacy-preserving machine learning.

Fangyu Zheng received the B.E. degree in information security from the University of Science and Technology of China, Hefei, China, in 2011 and the Ph.D. degree in information security from the University of Chinese Academy of Sciences, Beijing, China, in 2016.

He is currently an Associate Professor with the School of Cryptology, University of Chinese Academy of Sciences. His research interests include applied cryptography and high-performance computing.

Mingyu Wang received the B.E. degree in computer science and technology from Dalian Maritime University, Dalian, China, in 2018 and the Ph.D. degree in computer application from the University of Chinese Academy of Sciences, Beijing, China, in 2024.

She is currently an Assistant Professor with the School of Information Science and Technology, Dalian Maritime University, Dalian, China. Her research interests include applied cryptography and operating system security.

Ziqiang Ma received the Ph.D. degree in cyberspace security from the University of Chinese Academy of Sciences, Beijing, China, in 2020.

He is currently an Associate Professor of School of Information Engineering, Ningxia University, Ningxia, China. His research interests include system security, network traffic identification and analysis, and blockchain applications.

Jiankuo Dong received the B.E. degree in computer science and technology from the Xi'an Jiaotong University, Xi'an, China, in 2014 and the Ph.D. degree in cyberspace security from the University of Chinese Academy of Sciences, Beijing, China, in 2019.

He is currently an Associate Professor with the School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, China. His research interests include public key cryptography and applied cryptography.

Jingqiang Lin (Senior Member, IEEE) received the M.S. degree in communications and information systems and the Ph.D. degree in information security from the University of Chinese Academy of Sciences, Beijing, China, in 2004 and 2009, respectively.

He is currently a Full Professor with the School of Cyber Science and Technology, University of Science and Technology of China, Hefei, China. His research interests include applied cryptography and system security.