Number "Not Used" Once - Practical fault attack on pqm4 implementations of NIST candidates

Prasanna Ravi, Debapriya Basu Roy, Shivam Bhasin, Anupam Chattopadhyay, Debdeep Mukhopadhyay

COSADE-2019
5th April 2019
Table of Contents

1 Context
2 Lattice based Crypto: Background
3 Fault Vulnerability
4 Key Recovery Attacks
5 Message Recovery Attacks
6 Experimental Validation
7 Countermeasures
8 Conclusion
Table of Contents

1. Context
2. Lattice based Crypto: Background
3. Fault Vulnerability
4. Key Recovery Attacks
5. Message Recovery Attacks
6. Experimental Validation
7. Countermeasures
8. Conclusion
Context

- Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.
Context

- Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.
- The most powerful universal gate quantum computer: 160 physical qbits from IonQ.

How many qubits do we need to break RSA-2048?

NIST process for standardization of Post-Quantum Cryptography (PQC) is underway.

Started in December 2017, 3-5 years analysis period, followed by 2 years for draft standards.
Context

• Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.
• The most powerful universal gate quantum computer: 160 physical qubits from IonQ.
• Bristlecone, Google’s quantum processor currently works with 72 physical qubits.
Context

• Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.

• The most powerful universal gate quantum computer: 160 physical qubits from IonQ.

• Bristlecone, Google’s quantum processor currently works with 72 physical qubits.

• How many qubits do we need to break RSA-2048??
Context

- Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.
- The most powerful universal gate quantum computer: 160 physical qbits from IonQ.
- Bristlecone, Google’s quantum processor currently works with 72 physical qubits.
- How many qubits do we need to break RSA-2048?? \(4096\) logical qubits
Context

- Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.
- The most powerful universal gate quantum computer: 160 physical qubits from IonQ.
- Bristlecone, Google’s quantum processor currently works with 72 physical qubits.
- How many qubits do we need to break RSA-2048?? 4096 logical qubits ← Millions of physical qubits
Context

• Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.

• The most powerful universal gate quantum computer: 160 physical qubits from IonQ.

• Bristlecone, Google’s quantum processor currently works with 72 physical qubits.

• How many qubits do we need to break RSA-2048?? 4096 logical qubits ← Millions of physical qubits
Context

• Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.
• The most powerful universal gate quantum computer: 160 physical qubits from IonQ.
• Bristlecone, Google’s quantum processor currently works with 72 physical qubits.
• How many qubits do we need to break RSA-2048?? 4096 logical qubits \(\leftarrow\) Millions of physical qubits
• NIST process for standardization of Post-Quantum Cryptography (PQC) is underway.
Context

- Huge money in quantum computing is being invested by computer industry giants like Google, IBM, Intel and other companies like D-Wave, IonQ.
- The most powerful universal gate quantum computer: 160 physical qubits from IonQ.
- Bristlecone, Google’s quantum processor currently works with 72 physical qubits.
- How many qubits do we need to break RSA-2048?? 4096 logical qubits ← Millions of physical qubits
- NIST process for standardization of Post-Quantum Cryptography (PQC) is underway.
- Started in December 2017, 3-5 years analysis period, followed by 2 years for draft standards.
NIST PQC Call

- Signatures
- Encryption
- Key-establishments (KEMs)

Selection Criteria:
- Security
- Performance
- Backward compatibility
- Perfect forward secrecy
- ...
NIST PQC Call

- Signatures
- Encryption
- Key-establishments (KEMs)
- Selection Criteria:
  - Security
  - Performance
  - Backward compatibility
  - Perfect forward secrecy
  - Resistance to implementation attacks
  - ...
## NIST PQC Call

<table>
<thead>
<tr>
<th>Type</th>
<th>Signatures</th>
<th>KEM/Encryption</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lattice-based</td>
<td>5</td>
<td>23</td>
<td>28</td>
</tr>
<tr>
<td>Code-based</td>
<td>3</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>Multivariate</td>
<td>8</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Hash-based</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Isogeny-based</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Others</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>21</strong></td>
<td><strong>48</strong></td>
<td><strong>69</strong></td>
</tr>
</tbody>
</table>
### NIST PQC Call

<table>
<thead>
<tr>
<th>Type</th>
<th>Signatures</th>
<th>KEM/Encryption</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lattice-based</td>
<td>3</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>Code-based</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Multivariate</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Hash-based</td>
<td>2</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>Isogeny-based</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Others</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9</strong></td>
<td><strong>17</strong></td>
<td><strong>26</strong></td>
</tr>
</tbody>
</table>
This Work

- Fault Attack on 4 Lattice-based schemes: NewHope, Frodo, Kyber, Dilithium
- Fault Vulnerability: Usage of nonces in the sampling operation.
- Fault Model: Instruction Skips on the ARM Cortex-M4.
- Number of faults: 1-5.
- Nonce-reuse attacks are not new... Well known in the context of ECC.
This Work

- Fault Attack on 4 Lattice-based schemes: NewHope, Frodo, Kyber, Dilithium
- Fault Vulnerability: Usage of nonces in the sampling operation.
- Fault Model: Instruction Skips on the ARM Cortex-M4.
- Number of faults: 1-5.
- Nonce-reuse attacks are not new... Well known in the context of ECC.
- Impact:
This Work

- Fault Attack on 4 Lattice-based schemes: *NewHope, Frodo, Kyber, Dilithium*
- Fault Vulnerability: Usage of nonces in the sampling operation.
- Number of faults: 1-5.
- Nonce-reuse attacks are not new... Well known in the context of ECC.
- Impact:
  - *Key Recovery Attack*
This Work

- Fault Attack on 4 Lattice-based schemes: NewHope, Frodo, Kyber, Dilithium
- Fault Vulnerability: Usage of nonces in the sampling operation.
- Fault Model: **Instruction Skips** on the ARM Cortex-M4.
- Number of faults: 1-5.
- Nonce-reuse attacks are not new... Well known in the context of ECC.
- Impact:
  - Key Recovery Attack
  - Message Recovery Attack in CCA-secure KEM schemes in Man In The Middle (MITM) setting
Table of Contents

1 Context

2 Lattice based Crypto: Background

3 Fault Vulnerability

4 Key Recovery Attacks

5 Message Recovery Attacks

6 Experimental Validation

7 Countermeasures

8 Conclusion
Learning With Errors (LWE) problem

Let $A \in \mathbb{Z}^{n \times n}_q$ and $S, E \in \mathbb{Z}^n_q \leftarrow D_{\sigma \cdot T}$. Then $T = (A \times S + E) \in \mathbb{Z}^{n \times n}_q$.

- **Search LWE**: Given several pairs $(A, T)$, find $S$.
- **Decisional LWE**: Distinguish between valid LWE pairs $(A, T)$ from uniformly random samples in $(\mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^n)$.

Computations over matrices and vectors were mapped to polynomials in the more efficient variants of LWE such as **Ring-LWE (RLWE)** and **Module-LWE (MLWE)**.

- **Ring LWE**: $\mathbb{R}_q^{\mathbb{Z}[X]/(X^n + 1)}$ with $A, S, E \in \mathbb{R}_q^{\mathbb{Z}[X]/(X^n + 1)}$.
- **Module LWE**: $\mathbb{R}_k^{\mathbb{Z}[X]/(X^n + 1)}^{\mathbb{Z}[X]/(X^n + 1)}$ with $A \in \mathbb{R}_k^{\mathbb{Z}[X]/(X^n + 1)}, S \in \mathbb{R}_l^{\mathbb{Z}[X]/(X^n + 1)}, E \in \mathbb{R}_k^{\mathbb{Z}[X]/(X^n + 1)}$.

- **Learning With Rounding (LWR)**: Error deterministically generated by rounding to a lower modulus.
Learning With Errors (LWE) problem

- Let $A \in \mathbb{Z}_q^{n \times n}$ and $S, E \in \mathbb{Z}_q^n \leftarrow D_\sigma$
Learning With Errors (LWE) problem

- Let \( A \in \mathbb{Z}^{n \times n}_q \) and \( S, E \in \mathbb{Z}^n_q \leftarrow D_\sigma \)
- \( T = (A \times S + E) \in \mathbb{Z}^n_q \)
Learning With Errors (LWE) problem

- Let $A \in \mathbb{Z}_q^{n \times n}$ and $S, E \in \mathbb{Z}_q^n \leftarrow D_\sigma$
- $T = (A \times S + E) \in \mathbb{Z}_q^n$
- Search LWE: Given several pairs $(A, T)$, find $S$. 

Computations over matrices and vectors were mapped to polynomials in the more efficient variants of LWE such as Ring-LWE (RLWE) and Module-LWE (MLWE).

- Ring LWE: $R_q = \mathbb{Z}_q[X]/(X^n+1)$ with $A, S, E \in R_q$.
- Module LWE: $R_{k \times \ell_q} = (\mathbb{Z}_q[X]/(X^n+1))^{k \times \ell}$ with $A \in R_{k \times \ell_q}$, $S \in R_{\ell_q}$, $E \in R_{k_q}$.

Learning With Rounding (LWR): Error deterministically generated by rounding to a lower modulus.
Learning With Errors (LWE) problem

- Let $A \in \mathbb{Z}_q^{n \times n}$ and $S, E \in \mathbb{Z}_q^n \leftarrow D_{\sigma}$
- $T = (A \times S + E) \in \mathbb{Z}_q^n$
- Search LWE: Given several pairs $(A, T)$, find $S$.
- Decisional LWE: Distinguish between valid LWE pairs $(A, T)$ from uniformly random samples in $(\mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^n)$.
Learning With Errors (LWE) problem

- Let $A \in \mathbb{Z}^{n \times n}_q$ and $S, E \in \mathbb{Z}^n_q \leftarrow D_\sigma$
- $T = (A \times S + E) \in \mathbb{Z}^n_q$
- Search LWE: Given several pairs $(A, T)$, find $S$.
- Decisional LWE: Distinguish between valid LWE pairs $(A, T)$ from uniformly random samples in $(\mathbb{Z}^{n \times n}_q \times \mathbb{Z}^n_q)$.
- Computations over matrices and Vectors were mapped to polynomials in the more efficient variants of LWE such as Ring-LWE (RLWE) and Module-LWE (MLWE).
Learning With Errors (LWE) problem

- Let \( A \in \mathbb{Z}_q^{n \times n} \) and \( S, E \in \mathbb{Z}_q^n \leftarrow D_\sigma \)
- \( T = (A \times S + E) \in \mathbb{Z}_q^n \)
- Search LWE: Given several pairs \((A, T)\), find \( S \).
- Decisional LWE: Distinguish between valid LWE pairs \((A, T)\) from uniformly random samples in \((\mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^n)\).
- Computations over matrices and Vectors were mapped to polynomials in the more efficient variants of LWE such as Ring-LWE (RLWE) and Module-LWE (MLWE).
- Ring LWE: \( R_q = \mathbb{Z}_q[X]/(X^n + 1) \) with \( A, S, E \in R_q \).
Learning With Errors (LWE) problem

- Let \( A \in \mathbb{Z}_q^{n \times n} \) and \( S, E \in \mathbb{Z}_q^n \leftarrow D_\sigma \)
- \( T = (A \times S + E) \in \mathbb{Z}_q^n \)
- Search LWE: Given several pairs \((A, T)\), find \( S \).
- Decisional LWE: Distinguish between valid LWE pairs \((A, T)\) from uniformly random samples in \((\mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^n)\).
- Computations over matrices and Vectors were mapped to polynomials in the more efficient variants of LWE such as Ring-LWE (RLWE) and Module-LWE (MLWE).
- Ring LWE: \( \mathbb{R}_q = \mathbb{Z}_q[X]/(X^n + 1) \) with \( A, S, E \in \mathbb{R}_q \).
- Module LWE: \( \mathbb{R}_q^{k \times l} = (\mathbb{Z}_q[X]/(X^n + 1))^{k \times l} \) with \( A \in \mathbb{R}_q^{k \times l} \), \( S \in \mathbb{R}_q^l \), \( E \in \mathbb{R}_q^k \).
Learning With Errors (LWE) problem

- Let $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ and $\mathbf{S}, \mathbf{E} \in \mathbb{Z}_q^n \leftarrow D_\sigma$
- $\mathbf{T} = (\mathbf{A} \times \mathbf{S} + \mathbf{E}) \in \mathbb{Z}_q^n$
- Search LWE: Given several pairs $(\mathbf{A}, \mathbf{T})$, find $\mathbf{S}$.
- Decisional LWE: Distinguish between valid LWE pairs $(\mathbf{A}, \mathbf{T})$ from uniformly random samples in $(\mathbb{Z}_q^{n \times n} \times \mathbb{Z}_q^n)$.
- Computations over matrices and vectors were mapped to polynomials in the more efficient variants of LWE such as Ring-LWE (RLWE) and Module-LWE (MLWE).
- Ring LWE: $\mathbb{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $\mathbf{A}, \mathbf{S}, \mathbf{E} \in \mathbb{R}_q$.
- Module LWE: $\mathbb{R}_q^{k \times \ell} = (\mathbb{Z}_q[X]/(X^n + 1))^{k \times \ell}$ with $\mathbf{A} \in \mathbb{R}_q^{k \times \ell}$, $\mathbf{S} \in \mathbb{R}_q^\ell$, $\mathbf{E} \in \mathbb{R}_q^k$.
- Learning With Rounding (LWR): Error deterministically generated by rounding to a lower modulus.
The Importance of Error

- Error component $E$ is essential to hardness guarantees
The Importance of Error

- Error component $E$ is essential to hardness guarantees
- Without $E$, LWE instance becomes solvable modular linear equations $T = A \ast S$
The Importance of Error

• Error component \( E \) is essential to hardness guarantees
• Without \( E \), LWE instance becomes solvable modular linear equations \( T = A \ast S \)
• An attack reducing (or bounding) \( E \) could potentially compromise the security of the scheme
The Importance of Error

• Error component $E$ is essential to hardness guarantees
• Without $E$, LWE instance becomes solvable modular linear equations $T = A \ast S$
• An attack reducing (or bounding) $E$ could potentially compromise the security of the scheme
• Several insecure instantiations of LWE:
The Importance of Error

- Error component $E$ is essential to hardness guarantees.
- Without $E$, LWE instance becomes solvable modular linear equations $T = A \ast S$.
- An attack reducing (or bounding) $E$ could potentially compromise the security of the scheme.
- Several insecure instantiations of LWE:
  - Distribution always outputs zero error.
The Importance of Error

- Error component $E$ is essential to hardness guarantees.
- Without $E$, LWE instance becomes solvable modular linear equations $T = A \ast S$.
- An attack reducing (or bounding) $E$ could potentially compromise the security of the scheme.
- Several insecure instantiations of LWE:
  - Distribution always outputs zero error.
  - Distribution always outputs an error in the interval $z + \left[ -\frac{1}{2}, \frac{1}{2} \right)$.
  - Sum of a specific set of error co-ordinates is always zero.
  - Secret is same as the Error.
The Importance of Error

- Error component $E$ is essential to hardness guarantees
- Without $E$, LWE instance becomes solvable modular linear equations $T = A \ast S$
- An attack reducing (or bounding) $E$ could potentially compromise the security of the scheme
- Several insecure instantiations of LWE:
  - Distribution always outputs zero error
  - Distribution always outputs an error in the interval $z + \left[ -\frac{1}{2}, \frac{1}{2} \right)$
  - Sum of a specific set of error co-ordinates is always zero
The Importance of Error

• Error component $E$ is essential to hardness guarantees
• Without $E$, LWE instance becomes solvable modular linear equations $T = A \ast S$
• An attack reducing (or bounding) $E$ could potentially compromise the security of the scheme
• Several insecure instantiations of LWE:
  • Distribution always outputs zero error
  • Distribution always outputs an error in the interval $z + \left[ -\frac{1}{2}, \frac{1}{2} \right)$
  • Sum of a specific set of error co-ordinates is always zero
  • Secret is same as the Error
Table of Contents

1. Context
2. Lattice based Crypto: Background
3. Fault Vulnerability
4. Key Recovery Attacks
5. Message Recovery Attacks
6. Experimental Validation
7. Countermeasures
8. Conclusion
Fault Vulnerability

- Certain amount of randomness required to generate $S$ and $E$.
- The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
Fault Vulnerability

- Certain amount of randomness required to generate $S$ and $E$.
- The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
- $S = \text{Sample}(\sigma_S)$, $E = \text{Sample}(\sigma_E)$
Fault Vulnerability

- Certain amount of randomness required to generate $S$ and $E$.
- The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
- $S = \text{Sample}(\sigma_S)$, $E = \text{Sample}(\sigma_E)$
Fault Vulnerability

- Certain amount of randomness required to generate $S$ and $E$.
- The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
- $S = \text{Sample}(\sigma_S)$, $E = \text{Sample}(\sigma_E)$
- Ideally, for every fresh instance of $\text{Sample}$, one should use a newly generated random seed.
Fault Vulnerability

• Certain amount of randomness required to generate $S$ and $E$.
• The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
• $S = \text{Sample}(\sigma_S)$, $E = \text{Sample}(\sigma_E)$
• Ideally, for every fresh instance of $\text{Sample}$, one should use a newly generated random seed.
• But, we observed...
Fault Vulnerability

- Certain amount of randomness required to generate $S$ and $E$.
- The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
  - $S = \text{Sample}(\sigma_S)$, $E = \text{Sample}(\sigma_E)$
- Ideally, for every fresh instance of $\text{Sample}$, one should use a newly generated random seed.
- But, we observed...
  - $S = \text{Sample}(\sigma, nonce_S)$, $E = \text{Sample}(\sigma, nonce_E)$
Fault Vulnerability

- Certain amount of randomness required to generate $S$ and $E$.
- The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
- $S = \text{Sample}(\sigma_S)$, $E = \text{Sample}(\sigma_E)$
- Ideally, for every fresh instance of $\text{Sample}$, one should use a newly generated random seed.
- But, we observed...
  - $S = \text{Sample}(\sigma, nonce_S)$, $E = \text{Sample}(\sigma, nonce_E)$
- In need for efficiency, the same seed appended with \textbf{one byte of nonce} is used across multiple instances of the $\text{Sample}$ function.
Fault Vulnerability

- Certain amount of randomness required to generate $S$ and $E$.
- The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
- $S = \text{Sample}(\sigma_S)$, $E = \text{Sample}(\sigma_E)$
- Ideally, for every fresh instance of Sample, one should use a newly generated random seed.
- But, we observed...
  - $S = \text{Sample}(\sigma, nonce_S)$, $E = \text{Sample}(\sigma, nonce_E)$
- In need for efficiency, the same seed appended with one byte of nonce is used across multiple instances of the Sample function.
- If this nonce could be faulted to realize reuse, then same seed is used to sample both $S$ and $E$ resulting in $S = E$. 
Fault Vulnerability

- Certain amount of randomness required to generate $S$ and $E$.
- The secret $S$ and error $E$ are sampled from the same distribution and utilize the same functions for sampling.
- $S = \text{Sample}(\sigma_S)$, $E = \text{Sample}(\sigma_E)$
- Ideally, for every fresh instance of Sample, one should use a newly generated random seed.
- But, we observed...
  - $S = \text{Sample}(\sigma, \text{nonce}_S)$, $E = \text{Sample}(\sigma, \text{nonce}_E)$
- In need for efficiency, the same seed appended with one byte of nonce is used across multiple instances of the Sample function.
- If this nonce could be faulted to realize reuse, then same seed is used to sample both $S$ and $E$ resulting in $S = E$. 
Fault Vulnerability

- Assume a Ring LWE instance

\[ T = A \times S + E \in \mathbb{R}_q \]
Fault Vulnerability

- Assume a Ring LWE instance

\[ T = A \times S + E \in R_q \]

- Inject fault such that \( E = S \).
Fault Vulnerability

- Assume a Ring LWE instance

\[ T = A \times S + E \in \mathbb{R}_q \]

- Inject fault such that \( E = S \).
- Ring-LWE instance is faulted to:

\[ T = A \times S + S \in \mathbb{R}_q \]
Fault Vulnerability

• Assume a Ring LWE instance

\[ T = A \times S + E \in \mathbb{R}_q \]

• Inject fault such that \( E = S \).
• Ring-LWE instance is faulted to:

\[ T = A \times S + S \in \mathbb{R}_q \]

• Modular linear system of equations with \( n \) equations and \( n \) unknowns which is trivially solvable.
Fault Vulnerability

- Assume a Ring LWE instance

\[ T = A \times S + E \in \mathbb{R}_q \]

- Inject fault such that \( E = S \).

- Ring-LWE instance is faulted to:

\[ T = A \times S + S \in \mathbb{R}_q \]

- Modular linear system of equations with \( n \) equations and \( n \) unknowns which is trivially solvable.

- Applies to all variants of LWE (General LWE, Ring-LWE, Module-LWE)
Fault Vulnerability

- These faulty LWE instances can be used to perform key recovery and message recovery attacks.
- Key recovery attacks are performed by faulting the key generation procedure.
- Key recovery attacks applicable to NewHope, Frodo, Kyber and Dilithium.
- Message recovery attacks are performed by faulting the encapsulation procedure.
- Message recovery attacks only applicable over NewHope, Frodo and Kyber KEM schemes.
# Table of Contents

1. Context
2. Lattice based Crypto: Background
3. Fault Vulnerability
4. Key Recovery Attacks
5. Message Recovery Attacks
6. Experimental Validation
7. Countermeasures
8. Conclusion
NewHope KEM

• NewHope is a suite of KEM (NewHope-CPA/CCA-KEM)
NewHope KEM

- NewHope is a suite of KEM (NewHope-CPA/CCA-KEM)
- Based on RLWE problem
- NewHope-CPA KEM is derived from the NewHope-CPA Public Key Encryption (PKE) scheme.

Sample operation:

- Sample takes input as a 32-byte seed and 1-byte of nonce
- It uses SHAKE256 (SHA-3 family) as an Extendable Output Function (XOF) to deterministically generate more random bits and subsequently generate $S$ and $E$.

In NIST submission, designers use nonce=(0,1).
NewHope KEM

- NewHope is a suite of KEM (NewHope-CPA/CCA-KEM)
- Based on RLWE problem
- NewHope-CPA KEM is derived from the NewHope-CPA Public Key Encryption (PKE) scheme.
- Further, NewHope-CCA KEM is obtained through application of FO transformation on NewHope-CPA KEM.
NewHope KEM

- NewHope is a suite of KEM
- Based on RLWE problem
- NewHope-CPA KEM is derived from the NewHope-CPA Public Key Encryption (PKE) scheme.
- Further, NewHope-CCA KEM is obtained through application of FO transformation on NewHope-CPA KEM.
- $S$ and $E$ are generated using a Sample operation.
NewHope KEM

- NewHope is a suite of KEM (NewHope-CPA/CCA-KEM)
- Based on RLWE problem
- NewHope-CPA KEM is derived from the NewHope-CPA Public Key Encryption (PKE) scheme.
- Further, NewHope-CCA KEM is obtained through application of FO transformation on NewHope-CPA KEM.
- $S$ and $E$ are generated using a Sample operation
- Sample takes input as a 32-byte seed and 1-byte of nonce
NewHope KEM

• NewHope is a suite of KEM (NewHope-CPA/CCA-KEM)
• Based on RLWE problem
• NewHope-CPA KEM is derived from the NewHope-CPA Public Key Encryption (PKE) scheme.
• Further, NewHope-CCA KEM is obtained through application of FO transformation on NewHope-CPA KEM.
• $S$ and $E$ are generated using a Sample operation
• Sample takes input as a 32-byte seed and 1-byte of nonce
• It uses SHAKE256 (SHA-3 family) as an Extendable Output Function (XOF) to deterministically generate more random bits and subsequently generate $S$ and $E$. 
NewHope KEM

- NewHope is a suite of KEM (NewHope-CPA/CCA-KEM)
- Based on RLWE problem
- NewHope-CPA KEM is derived from the NewHope-CPA Public Key Encryption (PKE) scheme.
- Further, NewHope-CCA KEM is obtained through application of FO transformation on NewHope-CPA KEM.
- S and E are generated using a Sample operation
- Sample takes input as a 32-byte seed and 1-byte of nonce
- It uses SHAKE256 (SHA-3 family) as an Extendable Output Function (XOF) to deterministically generate more random bits and subsequently generate $S$ and $E$.
- In NIST submission, designers use nonce=$(0,1)$. 
1: procedure NEWHOPE.CPAPKE.GEN()
2:  
3:   \( \hat{a} \leftarrow \text{GenA}(\text{publicseed}) \)
4:   \( s \leftarrow \text{PolyBitRev}(\text{Sample}(\text{noiseseed}, 0)) \)
5:   \( \hat{s} = \text{NTT}(s) \)
6:   \( e \leftarrow \text{PolyBitRev}(\text{Sample}(\text{noiseseed}, 1)) \)
7:   \( \hat{e} = \text{NTT}(e) \)
8:   \( \hat{b} = \hat{a} \ast \hat{s} + \hat{e} \)
9:  Return (\( pk = \text{EncodePK}(\hat{b}, \text{publicseed}), sk = \text{EncodePolynomial}(s) \))
10: end procedure
NEWHOPE CPA-PKE

1: procedure NEWHOPE.CPAPKE.GEN()

2: 

3: \( \hat{a} \leftarrow \text{GenA}(\text{publicseed}) \)

4: \( s \leftarrow \text{PolyBitRev}(\text{Sample}(\text{noiseseed}, 0 \rightarrow R)) \)

5: \( \hat{s} = \text{NTT}(s) \)

6: \( e \leftarrow \text{PolyBitRev}(\text{Sample}(\text{noiseseed}, 1 \rightarrow R)) \)

7: \( \hat{e} = \text{NTT}(e) \)

8: \( \hat{b} = \hat{a} \ast \hat{s} + \hat{e} \)

9: Return \( (pk = \text{EncodePK}(\hat{b}, \text{publicseed}), sk = \text{EncodePolynomial}(s)) \)

10: end procedure
NEWHOPE CPA-PKE

1: procedure NEWHOPE.CPAPKE.GEN()

2: 

3: \( \hat{a} \leftarrow \text{GenA}(publicseed) \)

4: \( s \leftarrow \text{PolyBitRev}(\text{Sample}(noiseseed, 0\rightarrow R)) \)

5: \( \hat{s} = \text{NTT}(s) \)

6: \( e \leftarrow \text{PolyBitRev}(\text{Sample}(noiseseed, 1\rightarrow R)) \)

7: \( \hat{e} = \text{NTT}(e) \)

8: \( \hat{b} = \hat{a} \ast \hat{s} + \hat{e} \)

9: Return \((pk = \text{EncodePK}(\hat{b}, publicseed), sk = \text{EncodePolynomial}(s))\)

10: end procedure
Frodo KEM

- Frodo, similar to NewHope is a suite of KEM (NewHope-CPA/CCA-KEM) based on the General LWE problem.
- We identify the same vulnerable usage of nonce for sampling $S$ and $E$. 
Frodo CPA-PKE

1: \textbf{procedure} FRODO.CPAPKE.GEN() \\
2: \hspace{1em} seed_A \leftarrow U(\{0, 1\}^{\text{len}_A}) \\
3: \hspace{1em} A \leftarrow \text{Frodo.Gen}(\text{seed}_A) \in \mathbb{Z}_{q}^{n \times n} \\
4: \hspace{1em} seed_E \leftarrow U(\{0, 1\}^{\text{len}_E}) \\
5: \hspace{1em} S \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 1) \in \mathbb{Z}_{q}^{n \times \bar{n}} \\
6: \hspace{1em} E \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 2) \in \mathbb{Z}_{q}^{n \times \bar{n}} \\
7: \hspace{1em} B = A \times S + E \\
8: \hspace{1em} \text{Public key } pk \leftarrow (\text{seed}_A, B) \text{ and Secret key } sk \leftarrow S \\
9: \textbf{end procedure}
Frodo CPA-PKE

1: procedure FRODO.CPAPKE.GEN()
2: \(seed_A \leftarrow U(\{0, 1\}^{\text{len}_A})\)
3: \(A \leftarrow \text{Frodo.Gen}(seed_A) \in \mathbb{Z}_{q}^{n \times n}\)
4: \(seed_E \leftarrow U(\{0, 1\}^{\text{len}_E})\)
5: \(S \leftarrow \text{Frodo.SampleMatrix}(seed_E, 1 \rightarrow R) \in \mathbb{Z}_{q}^{n \times \bar{n}}\)
6: \(E \leftarrow \text{Frodo.SampleMatrix}(seed_E, 2 \rightarrow R) \in \mathbb{Z}_{q}^{n \times \bar{n}}\)
7: \(B = A \times S + E\)
8: Public key \(pk \leftarrow (seed_A, B)\) and Secret key \(sk \leftarrow S\)
9: end procedure
Frodo CPA-PKE

1: function FRODO.CPAPKE.GEN()
2: \( \text{seed}_A \leftarrow U(\{0, 1\}^{\text{len}_A}) \)
3: \( A \leftarrow \text{Frodo.Gen}(\text{seed}_A) \in \mathbb{Z}_q^{n \times n} \)
4: \( \text{seed}_E \leftarrow U(\{0, 1\}^{\text{len}_E}) \)
5: \( S \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 1 \rightarrow R) \in \mathbb{Z}_q^{n \times \bar{n}} \)
6: \( E \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 2 \rightarrow R) \in \mathbb{Z}_q^{n \times \bar{n}} \)
7: \( B = A \times S + E \)
8: Public key \( pk \leftarrow (\text{seed}_A, B) \) and Secret key \( sk \leftarrow S \)
9: end function
Kyber KEM

- Kyber is a suite of KEM (NewHope-CPA/CCA-KEM) based on the MLWE problem.
- $S \in \mathbb{R}_q^k$ and $E \in \mathbb{R}_q^\ell$ are sampled from a Centered Binomial distribution.
- Same seeds appended with fixed nonces are yet again used in sampling $S$ and $E$. 
Kyber KEM

• Kyber is a suite of KEM (NewHope-CPA/CCA-KEM) based on the MLWE problem
• \( S \in \mathbb{R}_q^k \) and \( E \in \mathbb{R}_q^\ell \) are sampled from a Centered Binomial distribution.
• Same seeds appended with fixed nonces are yet again used in sampling \( S \) and \( E \).
• In NIST submission, designers use nonce=(0 to k-1) for \( S \) and nonce=(k to 2k-1) for \( E \).
Kyber CPA-PKE

1: procedure KYBER.CPAPKE.GEN()
2: \[ d \leftarrow \{0, 1\}^{256}, (\rho, \sigma) := G(d), N := 0 \]
3: For \( i \) from 0 to \( k - 1 \)
4: For \( j \) from 0 to \( k - 1 \)
5: \[ a[i][j] \leftarrow \text{Parse}(\text{XOF}(\rho||j||i)) \]
6: EndFor
7: EndFor
8: For \( i \) from 0 to \( k - 1 \)
9: \[ s[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, N)) \]
10: \[ N := N + 1 \]
11: EndFor
12: For \( i \) from 0 to \( k - 1 \)
13: \[ e[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, N)) \]
14: \[ N := N + 1 \]
15: EndFor
16: \[ \hat{s} \leftarrow \text{NTT}(s) \]
17: \[ t = \text{NTT}^{-1}(\hat{a} \ast \hat{s}) + e \]
18: \[ pk := (\text{Encode}_{d_t}(\text{Compress}_q(t, d_t))||\rho) \]
19: \[ \text{Secret Key} := \text{Encode}_{13}(\hat{s} \mod^+ q) \]
20: Return (Public Key, Secret Key)
21: end procedure
Kyber CPA-PKE

1: procedure KYBER.CPAPKE.GEN()
2: \[ d \leftarrow \{0, 1\}^{256}, (\rho, \sigma) := G(d), N := 0 \]
3: For \( i \) from 0 to \( k - 1 \)
4: For \( j \) from 0 to \( k - 1 \)
5: \[ a[i][j] \leftarrow \text{Parse}(XOF(\rho||j||i)) \]
6: EndFor
7: EndFor
8: For \( i \) from 0 to \( k - 1 \)
9: \[ s[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, N \rightarrow R)) \]
10: \( N := N + 1 \)
11: EndFor
12: For \( i \) from 0 to \( k - 1 \)
13: \[ e[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, N \rightarrow R)) \]
14: \( N := N + 1 \)
15: EndFor
16: \( \hat{s} \leftarrow \text{NTT}(s) \)
17: \[ t = \text{NTT}^{-1}(\hat{a} \ast \hat{s}) + e \]
18: \( pk := (\text{Encode}_{d_t}(\text{Compress}_q(t, d_t))||\rho) \)
19: Secret Key := Encode_{13}(\hat{s} mod^+ q)
20: Return (Public Key, Secret Key)
21: end procedure
Kyber CPA-PKE

1: procedure KYBER.CPAPKE.GEN()
2: \( d \leftarrow \{0, 1\}^{256}, (\rho, \sigma) := G(d), N := 0 \)
3: For \( i \) from 0 to \( k - 1 \)
4: For \( j \) from 0 to \( k - 1 \)
5: \( a[i][j] \leftarrow \text{Parse}(XOF(\rho||j||i)) \)
6: EndFor
7: EndFor
8: For \( i \) from 0 to \( k - 1 \)
9: \( s[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, N \rightarrow R)) \)
10: \( N := N + 1 \)
11: EndFor
12: For \( i \) from 0 to \( k - 1 \)
13: \( e[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, N \rightarrow R)) \)
14: \( N := N + 1 \)
15: EndFor
16: \( \hat{s} \leftarrow \text{NTT}(s) \)
17: \( t = \text{NTT}^{-1}(\hat{a} * \hat{s}) + e \)
18: \( pk := \text{Encode}_{dt}(\text{Compress}_q(t, dt)||\rho) \)
19: Secret Key := \text{Encode}_{13}(\hat{s} mod^+ q) \)
20: Return (Public Key, Secret Key)
21: end procedure
**Kyber CPA-PKE**

1: `procedure KYBER.CPAPKE.GEN()`
2: \(d \leftarrow \{0, 1\}^{256}, (\rho, \sigma) := G(d), N := 0\)
3: For \(i\) from 0 to \(k - 1\)
4: For \(j\) from 0 to \(k - 1\)
5: \(a[i][j] \leftarrow \text{Parse}(XOF(\rho||j||i))\)
6: EndFor
7: EndFor
8: For \(i\) from 0 to \(k - 1\)
9: \(s[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, N \rightarrow R))\)
10: \(N := N + 1\)
11: EndFor
12: For \(i\) from 0 to \(k - 1\)
13: \(e[i] \leftarrow \text{CBD}_\eta(\text{PRF}(\sigma, N \rightarrow R))\)
14: \(N := N + 1\)
15: EndFor
16: \(\hat{s} \leftarrow \text{NTT}(s)\)
17: \(t = \text{NTT}^{-1}(\hat{a} \ast \hat{s}) + e\)
18: Public Key := (Encode_{d_t}(\text{Compress}_q(t, d_t))||\rho) **** Adds more error
19: Secret Key := Encode_{13}(\hat{s} \mod + q)
20: Return (Public Key, Secret Key)
21: `end procedure`
Key Recovery Attack on Kyber

• The Compress function rounds each coefficient to a lower modulus thereby inherently introducing additional deterministic error.

• Though the induced fault nullified the error in the LWE instance, the LWR hardness might still not be possible to break.
Key Recovery Attack on Kyber

- The Compress function rounds each coefficient to a lower modulus thereby inherently introducing additional deterministic error.
- Though the induced fault nullified the error in the LWE instance, the LWR hardness might still not be possible to break.
- The authors have only considered rounding for efficiency and not for security.
Key Recovery Attack on Kyber

- The Compress function rounds each coefficient to a lower modulus thereby inherently introducing additional deterministic error.
- Though the induced fault nullified the error in the LWE instance, the LWR hardness might still not be possible to break.
- The authors have only considered rounding for efficiency and not for security.
- The authors state that “we believe that the compression technique adds some security”, but it has not been quantified.
Key Recovery Attack on Kyber

- The Compress function rounds each coefficient to a lower modulus thereby inherently introducing additional deterministic error.
- Though the induced fault nullified the error in the LWE instance, the LWR hardness might still not be possible to break.
- The authors have only considered rounding for efficiency and not for security.
- The authors state that “we believe that the compression technique adds some security”, but it has not been quantified.
- Thus, our fault does not result in direct key recovery attack, but brings down the hardness to solving the corresponding LWR problem.
Dilithium Signature Scheme

- Dilithium is a Fiat-Shamir Abort-based lattice signature scheme.
- Indistinguishability of the Public key is based on the MLWE problem.
- Here again, nonces appended with domain separators are used to sample $S \in R_q^\ell$ and $E \in R_q^k$. 
Dilithium Signature Scheme

1: `procedure DILITHIUM.KEYGEN()`
2: \( \rho, \rho' \leftarrow \{0, 1\}^{256}, K \leftarrow \{0, 1\}^{256}, N := 0 \)
3: For \( i \) from 0 to \( \ell - 1 \)
4: \( s_1[i] := \text{Sample}(PRF(\rho', N)) \)
5: \( N := N + 1 \)
6: EndFor
7: For \( i \) from 0 to \( k - 1 \)
8: \( s_2[i] := \text{Sample}(PRF(\rho', N)) \)
9: \( N := N + 1 \)
10: EndFor
11: \( A \sim R_q^{k \times \ell} := \text{ExpandA}(\rho) \)
12: Compute \( t = A \times s_1 + s_2 \)
13: Compute \( t_1 := \text{Power2Round}_q(t, d) \)
14: \( tr \in \{0, 1\}^{384} := \text{CRH}(\rho || t_1) \)
15: Return \( pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0) \)
16: `end procedure`
Dilithium Signature Scheme

1: procedure DILITHIUM.KEYGEN()
2: \(\rho, \rho' \leftarrow \{0, 1\}^{256}, K \leftarrow \{0, 1\}^{256}, N := 0\)
3: For \(i\) from 0 to \(\ell - 1\)
4: \(s_1[i] := \text{Sample}(\text{PRF}(\rho', N \rightarrow R))\)
5: \(N := N + 1\)
6: EndFor
7: For \(i\) from 0 to \(k - 1\)
8: \(s_2[i] := \text{Sample}(\text{PRF}(\rho', N \rightarrow R))\)
9: \(N := N + 1\)
10: EndFor
11: \(A \sim R_q^{k \times \ell} := \text{ExpandA}(\rho)\)
12: Compute \(t = A \times s_1 + s_2\)
13: Compute \(t_1 := \text{Power2Round}_q(t, d)\)
14: \(tr \in \{0, 1\}^{384} := \text{CRH}(\rho||t_1)\)
15: Return \(pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0)\)
16: end procedure
Dilithium Signature Scheme

1: procedure DILITHIUM.KEYGEN()
2: \( \rho, \rho' \leftarrow \{0, 1\}^{256}, K \leftarrow \{0, 1\}^{256}, N := 0 \)
3: For \( i \) from 0 to \( \ell - 1 \)
4: \( s_1[i] := \text{Sample}(PRF(\rho', N \rightarrow R)) \)
5: \( N := N + 1 \)
6: EndFor
7: For \( i \) from 0 to \( k - 1 \)
8: \( s_2[i] := \text{Sample}(PRF(\rho', N \rightarrow R)) \)
9: \( N := N + 1 \)
10: EndFor
11: \( A \sim R_q^{k \times \ell} := \text{ExpandA}(\rho) \)
12: Compute \( t = A \times s_1 + s_2 \)
13: Compute \( t_1 := \text{Power2Round}_q(t, d) \)
14: \( tr \in \{0, 1\}^{384} := \text{CRH}(\rho||t_1) \)
15: Return \( pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0) \)
16: end procedure
Dilithium Signature Scheme

1: procedure DILITHIUM.KEYGEN()
2: \( \rho, \rho' \leftarrow \{0, 1\}^{256}, K \leftarrow \{0, 1\}^{256}, N := 0 \)
3: For \( i \) from 0 to \( \ell - 1 \)
4: \( s_1[i] := Sample(PR\{\rho', N \rightarrow R\}) \)
5: \( N := N + 1 \)
6: EndFor
7: For \( i \) from 0 to \( k - 1 \)
8: \( s_2[i] := Sample(PR\{\rho', N \rightarrow R\}) \)
9: \( N := N + 1 \)
10: EndFor
11: \( A \sim R_q^{k \times \ell} := \text{ExpandA}(\rho) \)
12: Compute \( t = A \times s_1 + s_2 \)
13: Compute \( t_1 := \text{Power2Round}_q(t, d) \) **** Only the top \( d \) bits of \( t \)
14: \( tr \in \{0, 1\}^{384} := \text{CRH}(\rho || t_1) \)
15: Return \( pk = (\rho, t_1), sk = (\rho, K, tr, s_1, s_2, t_0) \)
16: end procedure
Key Recovery Attack on Dilithium

- Only the higher order bits of the LWE instance $t$ are declared as the public key.
Key Recovery Attack on Dilithium

- Only the higher order bits of the LWE instance $t$ are declared as the public key.
- Some rounding error is introduced on top of the LWE instance $t$. 
Key Recovery Attack on Dilithium

• Only the higher order bits of the LWE instance $t$ are declared as the public key.

• Some rounding error is introduced on top of the LWE instance $t$.

• Security Analysis of Dilithium assumes that the whole of $t$ is known to the adversary. The original LWE instance $t$ can be derived just through observation of a large number of signatures.
Key Recovery Attack on Dilithium

- Only the higher order bits of the LWE instance $t$ are declared as the public key.
- Some rounding error is introduced on top of the LWE instance $t$.
- Security Analysis of Dilithium assumes that the whole of $t$ is known to the adversary. The original LWE instance $t$ can be derived just through observation of a large number of signatures.
- If the whole of $t$ can be derived by the adversary, our induced faults results in a key recovery attack.
# Table of Contents

1. Context
2. Lattice based Crypto: Background
3. Fault Vulnerability
4. Key Recovery Attacks
5. Message Recovery Attacks
6. Experimental Validation
7. Countermeasures
8. Conclusion
NEWHOPE CPA-PKE

1: procedure
   NEWHOPE.CPAPKE.ENC(pk \in \{0, \ldots, 255\}^{7.n/4+32}, \mu \in \{0, \ldots, 255\}^{32}, coin \in \{0, \ldots, 255\}^{32})
2:   :
3:   \hat{s} \leftarrow PolyBitRev(Sample(coin, 0))
4:   \hat{e} \leftarrow PolyBitRev(Sample(coin, 1))
5:   \hat{e} \leftarrow Sample(coin, 2)
6:   \hat{t} = NTT(\hat{s})
7:   \hat{u} = \hat{a} \ast \hat{t} + NTT(\hat{e})
8:   v = Encode(\mu)
9:   \hat{v} = NTT^{-1}(\hat{b} \ast \hat{t}) + \hat{e} + v
10:  h = Compress(\hat{v})
11:  Return c = EncodeC(\hat{u}, h)
12: end procedure
NEWHOPE CPA-PKE

1: \textbf{procedure} \\
NEWHOPE.CPAPKE.ENC\left(pk \in \{0, \ldots, 255\}^{7.n/4+32}, \mu \in \{0, \ldots, 255\}^{32}, coin \in \{0, \ldots, 255\}^{32}\right) \\
2: \colon \\
3: \hat{s} \leftarrow \text{PolyBitRev}\left(\text{Sample}(\text{coin}, 0 \rightarrow R)\right) \\
4: \hat{e} \leftarrow \text{PolyBitRev}\left(\text{Sample}(\text{coin}, 1 \rightarrow R)\right) \\
5: \hat{\epsilon} \leftarrow \text{Sample}(\text{coin}, 2) \\
6: \hat{t} = \text{NTT}(\hat{s}) \\
7: \hat{u} = \hat{a} \ast \hat{t} + \text{NTT}(\hat{e}) \\
8: \hat{v} = \text{Encode}(\mu) \\
9: \hat{v} = \text{NTT}^{-1}(\hat{b} \ast \hat{t}) + \hat{\epsilon} + \hat{v} \\
10: h = \text{Compress}(\hat{v}) \\
11: \text{Return } c = \text{EncodeC}(\hat{u}, h) \\
12: \textbf{end procedure}
NEWHOPE CPA-PKE

1: procedure
   NEWHOPE.CPAPKE.ENC\((pk \in \{0, \ldots, 255\}^{7.n/4+32}, \mu \in \{0, \ldots, 255\}^{32}, coin \in \{0, \ldots, 255\}^{32}\))

2: :

3: \(s \leftarrow \text{PolyBitRev}(\text{Sample}(coin, 0 \rightarrow R))\)
4: \(e \leftarrow \text{PolyBitRev}(\text{Sample}(coin, 1 \rightarrow R))\)
5: \(e \leftarrow \text{Sample}(coin, 2)\)
6: \(t = \text{NTT}(s)\)
7: \(\hat{u} = \hat{a} \ast \hat{t} + \text{NTT}(e)\)
8: \(v = \text{Encode}(\mu)\)
9: \(\hat{v} = \text{NTT}^{-1}(\hat{b} \ast \hat{t}) + \hat{e} + v\)
10: \(h = \text{Compress}(\hat{v})\)
11: Return \(c = \text{EncodeC}(\hat{u}, h)\)
12: end procedure
NEWHOPE CPA-PKE

1: procedure
NEWHOPE.CPAPKE.ENC\((pk \in \{0, \ldots, 255\}^{7.n/4+32}, \mu \in \{0, \ldots, 255\}^{32}, coin \in \{0, \ldots, 255\}^{32}\))

2: :
3: \(s \leftarrow \text{PolyBitRev}(\text{Sample}(coin, 0 \rightarrow R))\)
4: \(e \leftarrow \text{PolyBitRev}(\text{Sample}(coin, 1 \rightarrow R))\)
5: \(\hat{e} \leftarrow \text{Sample}(coin, 2)\)
6: \(\hat{t} = \text{NTT}(s)\)
7: \(\hat{u} = \hat{a} \ast \hat{t} + \text{NTT}(\hat{e})\)
8: \(v = \text{Encode}(\mu)\)
9: \(\hat{v} = \text{NTT}^{-1}(\hat{b} \ast \hat{t}) + \hat{e} + v\)
10: \(h = \text{Compress}(\hat{v})\)
11: Return \(c = \text{EncodeC}(\hat{u}, h)\)
12: end procedure
NEWHOPE CPA-PKE

1: procedure
   NEWHOPE.CPAPKE.ENC(pk ∈ {0, ..., 255}^{7.n/4+32}, μ ∈ {0, ..., 255}^{32}, coin ∈ {0, ..., 255}^{32})
2:   : 
3:   s ← PolyBitRev(Sample(coin, 0 → R))
4:   e ← PolyBitRev(Sample(coin, 1 → R))
5:   e ← Sample(coin, 2)
6:   t = NTT(s)
7:   ü = â * ť + NTT(é)
8:   v = Encode(μ)
9:   ṽ = NTT^{-1}(b̂ * ť) + é + v
10:  h = Compress(ṽ)
11:  Return c = EncodeC(û, h)
12: end procedure
NEWHOPE CPA-PKE

1:  procedure
   NEWHOPE.CPAPKE.ENC(pk ∈ \{0, \ldots, 255\}^{7.n/4+32}, \mu ∈ \{0, \ldots, 255\}^{32}, coin ∈ \{0, \ldots, 255\}^{32})
2:   :
3:   s ← PolyBitRev(Sample(coin, 0 → R))
4:   e ← PolyBitRev(Sample(coin, 1 → R))
5:   é ← Sample(coin, 2)
6:   t = NTT(\hat{s})
7:   \hat{u} = \hat{a} \ast \hat{t} + NTT(\hat{e})
8:   v = Encode(\mu)
9:   \hat{v} = NTT^{-1}(\hat{b} \ast \hat{t}) + \hat{e} + v
10:  h = Compress(\hat{v})
11:  Return c = EncodeC(\hat{u}, h)
12:  end procedure
NEWHOPE CPA-PKE

1: procedure
NEWHOPE.CPAPKE.ENC\( (pk \in \{0, \ldots, 255\}^{7.n/4+32}, \mu \in \{0, \ldots, 255\}^{32}, coin \in \{0, \ldots, 255\}^{32}) \)

2: 

3: \( \hat{s} \leftarrow \text{PolyBitRev}(\text{Sample}(\text{coin}, 0 \rightarrow R)) \)

4: \( \hat{e} \leftarrow \text{PolyBitRev}(\text{Sample}(\text{coin}, 1 \rightarrow R)) \)

5: \( \hat{e} \leftarrow \text{Sample}(\text{coin}, 2) \)

6: \( \hat{t} = \text{NTT}(\hat{s}) \)

7: \( \hat{u} = \hat{a} \ast \hat{t} + \text{NTT}(\hat{e}) \)

8: \( \nu = \text{Encode}(\mu) \)

9: \( \hat{v} = \text{NTT}^{-1}(\hat{b} \ast \hat{t}) + \hat{e} + \nu \)

10: \( \hat{h} = \text{Compress}(\hat{v}) \)

11: Return \( c = \text{EncodeC}(\hat{u}, \hat{h}) \)

12: end procedure
NEWHOPE CPA-PKE

1: procedure
   NEWHOPE.CPAPKE.ENC(pk ∈ \{0, \ldots, 255\}^{7 \cdot n/4 + 32}, \mu \in \{0, \ldots, 255\}^{32}, coin \in \{0, \ldots, 255\}^{32})

2: :

3:   \hat{s} \leftarrow \text{PolyBitRev}(\text{Sample}(\text{coin}, 0 \rightarrow R))

4:   \hat{e} \leftarrow \text{PolyBitRev}(\text{Sample}(\text{coin}, 1 \rightarrow R))

5:   \hat{e} \leftarrow \text{Sample}(\text{coin}, 2)

6:   \hat{t} = \text{NTT}(\hat{s})

7:   \hat{u} = \hat{a} \ast \hat{t} + \text{NTT}(\hat{e})

8:   \hat{v} = \text{Encode}(\mu)

9:   \hat{v} = \text{NTT}^{-1}(\hat{b} \ast \hat{t}) + \hat{e} + \hat{v}

10:  h = \text{Compress}(\hat{v})

11:  Return c = \text{EncodeC}(\hat{u}, h)

12: end procedure
FRODO CPA-PKE

1: **procedure** FRODO.CPAPKE.ENC()
2: \( \text{seed}_E \leftarrow U(\{0, 1\}^{\text{len}_E}) \)
3: \( \hat{S} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 4) \in \mathbb{Z}_q^{\tilde{m} \times n} \)
4: \( \hat{E} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 5) \in \mathbb{Z}_q^{\tilde{m} \times n} \)
5: \( \hat{E} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 6) \in \mathbb{Z}_q^{n \times \tilde{n}} \)
6: Compute \( \hat{B} = \hat{S} \times A + \hat{E} \)
7: Compute \( V = \hat{S} \times B + \hat{E} + \text{Frodo.Encode}(\mu) \)
8: Ciphertext \( C \leftarrow (C_1, C_2) = (\hat{B}, V) \)
9: **end procedure**
FRODO CPA-PKE

1: procedure FRODO.CPAPKE.ENC()
2: \( seed_E \leftarrow U(\{0, 1\}^{\text{len}_E}) \)
3: \( \hat{S} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 4 \rightarrow R) \in \mathbb{Z}_q^{m \times n} \)
4: \( \hat{E} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 5 \rightarrow R) \in \mathbb{Z}_q^{m \times n} \)
5: \( \hat{E} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 6) \in \mathbb{Z}_q^{n \times \bar{n}} \)
6: Compute \( \hat{B} = \hat{S} \times A + \hat{E} \)
7: Compute \( V = \hat{S} \times B + \hat{E} + \text{Frodo.Encode}(\mu) \)
8: Ciphertext \( C \leftarrow (C_1, C_2) = (\hat{B}, V) \)
9: end procedure
FRODO CPA-PKE

1: procedure FRODO.CPAPKE.ENC()
2: \text{seed}_E \leftarrow U(\{0, 1\}^{\text{len}_E})
3: \hat{S} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 4 \rightarrow R) \in \mathbb{Z}_q^{\bar{m} \times n}
4: \hat{E} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 5 \rightarrow R) \in \mathbb{Z}_q^{\bar{m} \times n}
5: \hat{E} \leftarrow \text{Frodo.SampleMatrix}(\text{seed}_E, 6) \in \mathbb{Z}_q^{n \times \bar{n}}
6: \text{Compute } \hat{B} = \hat{S} \times A + \hat{E}
7: \text{Compute } V = \hat{S} \times B + \hat{E} + \text{Frodo.Encode}(\mu)
8: \text{Ciphertext } C \leftarrow (C_1, C_2) = (\hat{B}, V)
9: end procedure
FRODO CPA-PKE

1: procedure FRODO.CPAPKE.ENC()
2: \[seed_E \leftarrow U(\{0, 1\}^{len_E})\]
3: \[\hat{S} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 4 \rightarrow R) \in \mathbb{Z}_q^{m \times n}\]
4: \[\hat{E} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 5 \rightarrow R) \in \mathbb{Z}_q^{m \times n}\]
5: \[\hat{E} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 6) \in \mathbb{Z}_q^{n \times \bar{n}}\]
6: Compute \[\hat{B} = \hat{S} \times A + \hat{E}\]
7: Compute \[V = \hat{S} \times B + \hat{E} + \text{Frodo.Encode}(\mu)\]
8: Ciphertext \[C \leftarrow (C_1, C_2) = (\hat{B}, V)\]
9: end procedure
FRODO CPA-PKE

1: procedure FRODO.CPAPKE.ENC()
2: \[ seed_E \leftarrow U(\{0, 1\}^{\text{len}_E}) \]
3: \[ S \leftarrow \text{Frodo.SampleMatrix}(seed_E, 4 \rightarrow R) \in \mathbb{Z}_{q}^{m \times n} \]
4: \[ E \leftarrow \text{Frodo.SampleMatrix}(seed_E, 5 \rightarrow R) \in \mathbb{Z}_{q}^{m \times n} \]
5: \[ \hat{E} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 6) \in \mathbb{Z}_{q}^{n \times \bar{n}} \]
6: Compute \[ \hat{B} = S \times A + \hat{E} \]
7: Compute \[ V = S \times B + \hat{E} + \text{Frodo.Encode}(\mu) \]
8: Ciphertext \[ C \leftarrow (C_1, C_2) = (\hat{B}, V) \]
9: end procedure
**FRODO CPA-PKE**

1: **procedure** FRODO.CPAPKE.ENC()
2: \[ seed_E \leftarrow U(\{0, 1\}^{\text{len}_E}) \]
3: \[ \hat{S} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 4 \rightarrow R) \in \mathbb{Z}_q^{m \times n} \]
4: \[ \hat{E} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 5 \rightarrow R) \in \mathbb{Z}_q^{m \times n} \]
5: \[ \hat{E} \leftarrow \text{Frodo.SampleMatrix}(seed_E, 6) \in \mathbb{Z}_q^{n \times \bar{n}} \]
6: Compute \[ \hat{B} = \hat{S} \times A + \hat{E} \]
7: Compute \[ V = \hat{S} \times B + \hat{E} + \text{Frodo.Encode}(\mu) \]
8: Ciphertext \[ C \leftarrow (C_1, C_2) = (\hat{B}, V) \]
9: **end procedure**
KYBER CPA-PKE

1: procedure KYBER.CPAPKE.ENC\( (pk \in \mathcal{B}^{dt \cdot k \cdot n/8 + 32}, m \in \mathcal{B}^{32}, r \in \mathcal{B}^{32}) \)
2: \( N = 0 \)
3: For \( i \) from 0 to \( k - 1 \)
4: \( r[i] \leftarrow \text{CBD}_\eta(\text{PRF}(r, N)) \)
5: \( N := N + 1 \)
6: EndFor
7: For \( i \) from 0 to \( k - 1 \)
8: \( e_1 \leftarrow \text{CBD}_\eta(\text{PRF}(r, N)) \)
9: \( N := N + 1 \)
10: EndFor
11: For \( i \) from 0 to \( k - 1 \) \( e_2 \leftarrow \text{CBD}_\eta(\text{PRF}(r, N)) \)
12: EndFor
13: \( \hat{r} = \text{NTT}(r) \)
14: \( u = \text{NTT}^{-1}(\hat{a}^T \ast \hat{r}) + e_1 \)
15: \( v = \text{NTT}^{-1}(\hat{t}^T \ast \hat{r}) + e_2 + \text{Decode}_1(\text{Decompose}_q(m, 1)) \)
16: \( c_1 = \text{Encode}_{d_u}(\text{Compress}_q(u, d_u)) \)
17: \( c_2 = \text{Encode}_{d_v}(\text{Compress}_q(v, d_v)) \)
18: \( c = (c_1, c_2) \)
19: end procedure
KYBER CPA-PKE

1: procedure KYBER.CPAPKE.ENC(pk ∈ \mathcal{B}_{d \cdot k \cdot n/8 + 32}, m ∈ \mathcal{B}^{32}, r ∈ \mathcal{B}^{32})
2:     N = 0
3: For i from 0 to k − 1
4:     r[i] ← CBD_{\eta}(PRF(r, N → R))
5:     N := N + 1
6: EndFor
7: For i from 0 to k − 1
8:     e_{1} ← CBD_{\eta}(PRF(r, N → R))
9:     N := N + 1
10: EndFor
11: For i from 0 to k − 1 e_{2} ← CBD_{\eta}(PRF(r, N))
12: EndFor
13: \hat{r} = NTT(r)
14: u = NTT^{-1}(\hat{a}^T \ast \hat{r}) + e_{1}
15: v = NTT^{-1}(\hat{t}^T \ast \hat{r}) + e_{2} + \text{Decode}_1(\text{Decompose}_q(m, 1))
16: c_1 = \text{Encode}_{d_u}(\text{Compress}_q(u, d_u))
17: c_2 = \text{Encode}_{d_v}(\text{Compress}_q(v, d_v))
18: c = (c_1, c_2)
19: end procedure
KYBER CPA-PKE

1: procedure KYBER.CPAPKE.ENC(pk ∈ \mathcal{B}^{d_t \cdot k \cdot n/8+32}, m ∈ \mathcal{B}^{32}, r ∈ \mathcal{B}^{32})
2: \quad N = 0
3: \quad For i from 0 to k − 1
4: \quad r[i] ← CBD_\eta(\text{PRF}(r, N \rightarrow R))
5: \quad N := N + 1
6: \quad EndFor
7: \quad For i from 0 to k − 1
8: \quad e_1 ← CBD_\eta(\text{PRF}(r, N \rightarrow R))
9: \quad N := N + 1
10: \quad EndFor
11: \quad For i from 0 to k − 1 e_2 ← CBD_\eta(\text{PRF}(r, N))
12: \quad EndFor
13: \quad \hat{r} = \text{NTT}(r)
14: \quad u = \text{NTT}^{-1}(\hat{a}^T \ast \hat{r}) + e_1
15: \quad v = \text{NTT}^{-1}(\hat{t}^T \ast \hat{r}) + e_2 + \text{Decode}_1(\text{Decompose}_q(m, 1))
16: \quad c_1 = \text{Encode}_{d_u}(\text{Compress}_q(u, d_u))
17: \quad c_2 = \text{Encode}_{d_v}(\text{Compress}_q(v, d_v))
18: \quad c = (c_1, c_2)
19: end procedure
KYBER CPA-PKE

1: procedure KYBER.CPAPKE.ENC:pk ∈ B^{d_k \cdot k \cdot n/8 + 32}, m ∈ B^{32}, r ∈ B^{32})
2: N = 0
3: For i from 0 to k − 1
4: r[i] ← CBD_{\eta}(PRF(r, N → R))
5: N := N + 1
6: EndFor
7: For i from 0 to k − 1
8: e_1 ← CBD_{\eta}(PRF(r, N → R))
9: N := N + 1
10: EndFor
11: For i from 0 to k − 1 e_2 ← CBD_{\eta}(PRF(r, N))
12: EndFor
13: \hat{r} = NTT(r)
14: u = NTT^{-1}(\hat{\alpha}^T * \hat{r}) + e_1
15: v = NTT^{-1}(\hat{\tau}^T * \hat{r}) + e_2 + \text{Decode}_1(\text{Decompose}_q(m, 1))
16: c_1 = \text{Encode}_{d_u}(\text{Compress}_q(u, d_u)) **** Adds more error
17: c_2 = \text{Encode}_{d_v}(\text{Compress}_q(v, d_v))
18: c = (c_1, c_2)
19: end procedure
Translating Message Recovery Attack to CCA-KEM schemes

• CPA-secure PKE is transformed to CCA-secure KEM using the Quantum-Fujisaki Okamoto transformation.

• A **re-encapsulation** is performed in the decapsulation procedure to check for the validity of ciphertexts.

How do we bypass this?

We observe that a fault attacker in a Man-In-The-Middle (MITM) setting can still mount the attack without being detected during decapsulation.
Translating Message Recovery Attack to CCA-KEM schemes

- CPA-secure PKE is transformed to CCA-secure KEM using the Quantum-Fujisaki Okamoto transformation.
- A re-encapsulation is performed in the decapsulation procedure to check for the validity of ciphertexts.
- Thus, faults injected into the encapsulation procedure are detected during decapsulation.

How do we bypass this?

We observe that a fault attacker in a Man-In-The-Middle (MITM) setting can still mount the attack without being detected during decapsulation.
Translating Message Recovery Attack to CCA-KEM schemes

- CPA-secure PKE is transformed to CCA-secure KEM using the Quantum-Fujisaki Okamoto transformation.
- A re-encapsulation is performed in the decapsulation procedure to check for the validity of ciphertexts.
- Thus, faults injected into the encapsulation procedure are detected during decapsulation.
- How do we bypass this?
Translating Message Recovery Attack to CCA-KEM schemes

- CPA-secure PKE is transformed to CCA-secure KEM using the Quantum-Fujisaki Okamoto transformation.
- A re-encapsulation is performed in the decapsulation procedure to check for the validity of ciphertexts.
- Thus, faults injected into the encapsulation procedure are detected during decapsulation.
- How do we bypass this?
- We observe that a fault attacker in a Man-In-The-Middle (MITM) setting can still mount the attack without being detected during decapsulation.
Message Recovery Attack over CCA-KEM schemes

Figure: Fault assisted MITM attack on CCA Secure KEM scheme
# Table of Contents

1. Context
2. Lattice based Crypto: Background
3. Fault Vulnerability
4. Key Recovery Attacks
5. Message Recovery Attacks
6. Experimental Validation
7. Countermeasures
8. Conclusion
We target reference implementations from the \textit{pqm4} benchmarking framework for PQC candidates on the ARM Cortex-M4 microcontroller.

Implementations were ported to the STM32F4DISCOVERY board (DUT) housing the STM32F407 microcontroller.

Clock Frequency: 24 MHz.
Analysis of implementation for Fault Vulnerability

- We target the usage (not generation) of nonce in all reference implementations.
Analysis of implementation for Fault Vulnerability

- We target the usage (not generation) of nonce in all reference implementations.
- The seed to the `Sample` function along with the nonce is input as an array \( A \) to an XOF function.
We target the usage (not generation) of nonce in all reference implementations.

The seed to the `Sample` function along with the nonce is input as an array $A$ to an XOF function.

The nonce is stored as the last element of the array.
Analysis of implementation for Fault Vulnerability

- We target the usage (not generation) of nonce in all reference implementations.
- The seed to the `Sample` function along with the nonce is input as an array $A$ to an XOF function.
- The nonce is stored as the last element of the array.

```
[ ] [ ] [ ] [ ] [ ] ... [ ]
```
Analysis of implementation for Fault Vulnerability

• We target the usage (not generation) of nonce in all reference implementations.
• The seed to the Sample function along with the nonce is input as an array $A$ to an XOF function.
• The nonce is stored as the last element of the array.
Analysis of implementation for Fault Vulnerability

- We target the usage (not generation) of nonce in all reference implementations.
- The seed to the `Sample` function along with the nonce is input as an array $A$ to an XOF function.
- The nonce is stored as the last element of the array.

\[
\begin{array}{cccccc}
\text{\color{blue}{\ }} & \text{\color{blue}{\ }} & \text{\color{blue}{\ }} & \text{\color{blue}{\ }} & \text{\color{blue}{\ }} & \ldots & \text{\color{blue}{\ }} & \text{\color{yellow}{\ }}
\end{array}
\]
• We target the usage (not generation) of nonce in all reference implementations.
• The seed to the Sample function along with the nonce is input as an array $A$ to an XOF function.
• The nonce is stored as the last element of the array.

\[
\begin{array}{cccccc}
\text{blue} & \text{blue} & \text{blue} & \text{blue} & \text{…} & \text{red}
\end{array}
\]
Analysis of implementation for Fault Vulnerability

- We target the usage (not generation) of nonce in all reference implementations.
- The seed to the Sample function along with the nonce is input as an array $A$ to an XOF function.
- The nonce is stored as the last element of the array.

```
  A  A  A  A  ...  A  A  A  A
```

- For all the call instances to this XOF function, all the elements of the array $A$ are the same except the nonce value.
Analysis of implementation for Fault Vulnerability

- We target the usage (not generation) of nonce in all reference implementations.
- The seed to the Sample function along with the nonce is input as an array $A$ to an XOF function.
- The nonce is stored as the last element of the array.

```plaintext
  A A A A A ... A A A A A N
```

- For all the call instances to this XOF function, all the elements of the array $A$ are the same except the nonce value.
- If this nonce-store to the array is skipped, we are essentially using the same randomness to sample both $S$ and $E$. 
Analysis of implementation for Fault Vulnerability

<table>
<thead>
<tr>
<th>ldr r3,[r5,#28]</th>
<th>movs r1,#1</th>
</tr>
</thead>
<tbody>
<tr>
<td>stmia r4!,{r0,r1,r2,r3}</td>
<td>add r0,sp,#52</td>
</tr>
<tr>
<td>strb.w r7,[r6,#-132]!</td>
<td>strb.w r9,[r6,#32]</td>
</tr>
<tr>
<td>movs r1,#1</td>
<td>movs r2,#33</td>
</tr>
<tr>
<td>mov r0,r6</td>
<td>movs r3,#0</td>
</tr>
</tbody>
</table>

(a) Target operation in NewHope

| lsrs r2,r7,#8 | movs r1,#128 |
|               | ldr r0,[pc,#208] |
| ldr r3,[pc,#264] | strb.w r7,[sp,#44] |
| strb.w r2,[sp,#7] | add r1,sp,#12 |
| movw r2,#4097 | add r0,sp,#48 |
| mov r1,sp | |

(c) Target operation in Frodo

| |
|---|---|
| (d) Target operation in Dilithium |
Experimental Setup

Figure: Description of our EMFI setup
Experimental Setup

Figure: (1) EM Pulse Generator (2) USB-Microscope (3) STM32M4F Discovery Board (DUT) (4) Arduino based Relay Shield (5) Controller Laptop (6) Oscilloscope (7) EM Pulse Injector (8) XYZ Motorized Table
Experimental Setup

Figure: (a) Hand-made probe used for our EMFI setup (b) Probe placed over the DUT
Results on ARM Cortex-M4

- **Required Fault:** Skip the store instruction to a particular memory location.
- We profiled the ARM chip using a sample load and store program to find a "sweet spot" to skip the store to a particular memory location.

Fault sensitive region is the area near the ARM logo of the STM32M4F07 microcontroller.

Fault repeatability is (almost) 100% at the identified location for a specific set of voltage pulse parameters.

- Voltage: 150V-200V, Pulse Width = 12nsec, Rise-Time = 2 nsec.

Faults were synchronized with the target operation using an internally generated trigger.
Results on ARM Cortex-M4

- **Required Fault:** Skip the store instruction to a particular memory location.

- We profiled the ARM chip using a sample load and store program to find a "sweet spot" to skip the store to a particular memory location.

- Fault sensitive region is the area near the ARM logo of the STM32M4F07 microcontroller.

- Fault repeatability is (almost) 100% at the identified location for a specific set of voltage pulse parameters.

  - Voltage: 150V-200V, Pulse Width = 12nsec, Rise-Time = 2nsec.

- Faults were synchronized with the target operation using an internally generated trigger.
Results on ARM Cortex-M4

- **Required Fault:** Skip the store instruction to a particular memory location.
- We profiled the ARM chip using a sample load and store program to find a "sweet spot" to skip the store to a particular memory location.
- Fault sensitive region is the area near the ARM logo of the STM32M4F07 microcontroller.
- Fault repeatability is (almost) 100% at the identified location for a specific set of voltage pulse parameters.
Results on ARM Cortex-M4

- **Required Fault:** Skip the store instruction to a particular memory location.

- We profiled the ARM chip using a sample load and store program to find a ”sweet spot” to skip the store to a particular memory location.

- Fault sensitive region is the area near the ARM logo of the STM32M4F07 microcontroller.

- Fault repeatability is (almost) 100% at the identified location for a specific set of voltage pulse parameters.

- **Voltage:** 150V-200V, **Pulse Width** = 12nsec, **Rise-Time** = 2 nsec.
Results on ARM Cortex-M4

- **Required Fault**: Skip the store instruction to a particular memory location.
- We profiled the ARM chip using a sample load and store program to find a "sweet spot" to skip the store to a particular memory location.
- Fault sensitive region is the area near the ARM logo of the STM32M4F07 microcontroller.
- Fault repeatability is (almost) 100% at the identified location for a specific set of voltage pulse parameters.
  - Voltage: 150V-200V, Pulse Width = 12nsec, Rise-Time = 2 nsec.
- Faults were synchronized with the target operation using an internally generated trigger.
Fault Complexity

<table>
<thead>
<tr>
<th>Attack Objective</th>
<th>Fault Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NEWHOPE</td>
</tr>
<tr>
<td></td>
<td>NEWHOPE512</td>
</tr>
<tr>
<td>Key Recovery</td>
<td>1</td>
</tr>
<tr>
<td>Message Recovery</td>
<td>1</td>
</tr>
</tbody>
</table>

|                      | KYBER            | DILITHIUM        |
|                      | KYBER512         | KYBER768         | KYBER1024  | Weak | Med. | Rec. | High |
| Key Recovery         | 2                | 3                | 4          | 2    | 3    | 4    | 5    |
| Message Recovery     | 2                | 3                | 4          | -    | -    | -    | -    |
Fault Complexity

- Security of Kyber is weakened because the induced fault has removed the hardness from the LWE problem.
Fault Complexity

- Security of Kyber is weakened because the induced fault has removed the hardness from the LWE problem.
- If enough number of signatures corresponding to the same public-private key pair can be observed, then it can lead to a successful key recovery attack on Dilithium.

<table>
<thead>
<tr>
<th>Attack Objective</th>
<th>Fault Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NEWHOPE</td>
</tr>
<tr>
<td></td>
<td>NEWHOPE512</td>
</tr>
<tr>
<td>Key Recovery</td>
<td>1</td>
</tr>
<tr>
<td>Message Recovery</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>KYBER512</th>
<th>KYBER768</th>
<th>KYBER1024</th>
<th>Weak</th>
<th>Med.</th>
<th>Rec.</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Recovery</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Message Recovery</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Context  
2. Lattice based Crypto: Background  
3. Fault Vulnerability  
4. Key Recovery Attacks  
5. Message Recovery Attacks  
6. Experimental Validation  
7. Countermeasures  
8. Conclusion
Countermeasures and Future Directions

- Usage of separate seeds for $S$ and $E$
- Frodo has updated its specifications as part of its second round submission by using separate seeds for $S$ and $E$.
- Synchronization of faults with vulnerable operations.
- Study on weakened LWE instances in Kyber and Dilithium.
Table of Contents

1 Context
2 Lattice based Crypto: Background
3 Fault Vulnerability
4 Key Recovery Attacks
5 Message Recovery Attacks
6 Experimental Validation
7 Countermeasures
8 Conclusion
Conclusion

• We identified fault-vulnerabilities due to usage of nonces in multiple LWE-based lattice schemes.

• We proposed key recovery attacks over NewHope, Frodo, Kyber and Dilithium and message recovery attacks over NewHope, Frodo and Kyber KEM schemes.

• Practical Validation of our attack through EMFI on implementations from pqm4 library on the ARM Cortex-M4 microcontroller.

• We hope that nonces either be avoided or be used more carefully in the future.
Thank you!

Any questions?