ETSI TR 104 239-1 V1.2.1 (2026-05)

TECHNICAL REPORT

Cyber Security (CYBER); Quantum-Safe Cryptography (QSC); Secure Implementation Guidance for Key Encapsulation Mechanisms and Digital Signature Schemes; Part 1: General

Reference
RTR/CYBER-QSC-0032

Keywords
cyber security, digital signature, key exchange,
quantum safe cryptography

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM®** and the GSM logo are trademarks registered and owned by the GSM Association.

BLUETOOTH® is a trademark registered and owned by Bluetooth SIG, Inc.

Foreword

This Technical Report (TR) has been produced by ETSI Technical Committee Cyber Security (CYBER).

The present document is part 1 of a multi-part deliverable covering implementation guidance for quantum-safe key encapsulation mechanisms and digital signature schemes, as identified below:

Part 1: "General";

Part 2: "ML-KEM" [i.2];

Part 3: "ML-DSA" [i.3];

Part 4: "SLH-DSA" [i.4].

Later parts of this multi-part deliverable will provide implementation guidance for specific algorithms, building on the general guidance provided in the present document.

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

1 Scope

The present document provides developers with general guidance to aid the secure implementation of quantum-safe algorithms. This includes an overview of the interfaces and expected security properties of quantum-safe algorithms; some general good practice guidance for cryptographic implementations; some background on side-channel and fault attacks; and a brief discussion of testing and formal verification.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding, but are not required for conformance to the present document.

[i.1] ETSI EN 303 645: "CYBER; Cyber Security for Consumer Internet of Things: Baseline Requirements".

[i.2] ETSI TR 104 239-2: "Cyber Security (CYBER); Quantum-Safe Cryptography (QSC); Secure Implementation Guidance for Key Encapsulation Mechanisms and Digital Signature Schemes; Part 2: ML-KEM".

[i.3] ETSI TR 104 239-3: " Cyber Security (CYBER); Quantum-Safe Cryptography (QSC); Secure Implementation Guidance for Key Encapsulation Mechanisms and Digital Signature Schemes; Part 3: ML-DSA".

[i.4] ETSI TR 104 239-4: " Cyber Security (CYBER); Quantum-Safe Cryptography (QSC); Secure Implementation Guidance for Key Encapsulation Mechanisms and Digital Signature Schemes; Part 4: SLH-DSA".

[i.5] IRTF RFC 7748: "Elliptic Curves for Security".

[i.6] IETF RFC 8017: "PKCS #1: RSA Cryptography Specifications version 2.2".

[i.7] IRTF RFC 8032: "Edwards-Curve Digital Signature Algorithm (EdDSA)".

[i.8] IRTF RFC 9180: "Hybrid Public Key Encryption".

[i.9] IRTF RFC 9474: " RSA Blind Signatures".

[i.10] ISO/IEC 14888-3:2018: "IT Security techniques — Digital signatures with appendix — Part 3: Discrete logarithm based mechanisms".

[i.11] ISO/IEC 18033-2:2006: "Information technology — Security techniques — Encryption algorithms — Part 2: Asymmetric ciphers".

[i.12] NCSC: "Timelines for migration to post-quantum cryptography".

[i.13] NIST FIPS 186-5: "Digital Signature Standard (DSS)".

[i.14] NIST FIPS 203: "Module-Lattice-Based Key-Encapsulation Mechanism Standard".

[i.15] NIST FIPS 204: "Module-Lattice-Based Digital Signature Standard".

[i.16] NIST FIPS 205: "Stateless Hash-Based Digital Signature Standard".

[i.17] NIST SP 800-56A Rev. 3: "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography".

[i.18] NIST SP 800-56B Rev. 2: "Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography".

[i.19] NIST SP 800-89: "Recommendation for Obtaining Assurances for Digital Signature Applications".

[i.20] NIST SP 800-90 Series.

[i.21] NIST: "Automated Cryptographic Validation Test System".

[i.22] NIST: "Cryptographic Algorithm Validation Program (CAVP)".

[i.23] NIST: "Announcing Approval of Three Federal Information Processing Standards (FIPS) for Post-Quantum Cryptography".

[i.24] J. B. Almeida et al.: "Formally verifying Kyber Episode IV: Implementation correctness". IACR Transactions on Cryptographic Hardware and Embedded Systems, 2023(3), pp. 164-193, 2023.

[i.25] M. Barbosa et al.: "SoK: Computer-aided cryptography", IEEE™ Symposium on Security and Privacy (SP), pp. 777-795, 2021.

[i.26] G. Becker et al.: "Test Vector Leakage Assessment (TVLA) methodology in practice". International Cryptographic Module Conference 2013.

[i.27] R. Benadjila et al.: "Deep learning for side-channel analysis and introduction to ASCAD database". Journal of Cryptographic Engineering Volume 10 (2020), pp. 163-188, 2020.

[i.28] D. J. Bernstein et al.: "KyberSlash: Exploiting secret-dependent division timings in Kyber implementations". IACR Transactions on Cryptographic Hardware and Embedded Systems, 2025(2), pp. 209-234, 2025.

[i.29] E. Biham and A. Shamir: "Differential fault analysis of secret key cryptosystems". Advances in Cryptology - CRYPTO' 97. Lecture Notes in Computer Science, vol. 1294, 1997.

[i.30] G. Camurati et al.: "Screaming channels: When electromagnetic side channels meet radio transceivers". CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 163-177, 2018.

[i.31] Cybersecurity & Infrastructure Security Agency, US Government: "Secure-by-Design - Shifting the Balance of Cybersecurity Risk: Principles and Approaches for Secure by Design Software".

[i.32] Department of Science, Innovation and Technology, UK Government: "Software Security Code of Practice".

[i.33] fail0verflow: "Console hacking 2010". 27th Chaos Communication Congress, 2010.

[i.34] A. Genêt: "On Protecting SPHINCS+ Against Fault Attacks". IACR Transactions on Cryptographic Hardware and Embedded Systems, pp. 80-114, 2023.

[i.35] M. Hastings et al.: "Weak keys remain widespread in network devices". Proceedings of the 2016 Internet Measurement Conference (IMC 16), pp. 49-63, 2016.

[i.36] N. Heninger et al.: "Mining your Ps and Qs: Detection of widespread weak keys in network devices". 21st USENIX Security Symposium (USENIX Security 12), pp. 205-220.

[i.37] X. Hou et al.: "Fully automated differential fault analysis on software implementations of block ciphers". IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(3), pp. 1-29, 2019.

[i.38] J. Kilgallin and R. Vasko: "Factoring RSA keys in the IoT era". 2019 First IEEE™ International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), pp. 184-189, 2019.

[i.39] Y. Kim et al.: "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors". ACM SIGARCH Computer Architecture News, Volume 42, Issue 3, pp. 361-372.

[i.40] P. Kocher, J. Jaffe and B. Jun: "Differential power analysis". Advances in Cryptology - CRYPTO' 99. Lecture Notes in Computer Science, vol 1666. 1999.

[i.41] A. K. Lenstra et al.: "Public keys". Advances in Cryptology – CRYTPO 2012, pp. 626-642.

[i.42] N. Madden: "CVE-2022-21449: Psychic Signatures in Java".

[i.43] J. Manger: "A chosen ciphertext attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as standardized in PKCS #1 v2.0". Advances in Cryptology - CRYPTO 2001, pp. 230-238, 2001.

[i.44] J. Mattsson, E. Thormaker and S. Ruohomaa: "Hedged ECDSA and EdDSA signatures (work in progress)". IRTF Internet-Draft, draft-irtf-cfrg-det-sigs-with-noise-05.

[i.45] D. McCann, E. Oswald and C. Whitnall: "Towards practical tools for side channel aware software engineering: 'Grey box' modelling for instruction leakages". Proceedings of the 26th USENIX Security Symposium, 2017.

[i.46] Mozilla: "Cryptofuzz".

[i.47] B. Nassi et al.: "Video-Based cryptanalysis: Extracting cryptographic keys from video footage of a device's power LED captured by standard video cameras". Proceedings - 45th IEEE™ Symposium on Security and Privacy, pp. 2442-2440, 2024.

[i.48] M. Polubelova et al.: "HACLxN: Verified generic SIMD crypto (for all your favorite platforms)". Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 899-918, 2020.

[i.49] Post-Quantum Cryptography Alliance: "mlkem-native".

[i.50] T. Roche: "EUCLEAK side-channel attack on the YubiKey 5 series (Revealing and breaking Infineon ECDSA implementation on the way)". Proceedings - IEEE™ Symposium on Security and Privacy, pp. 4026-4043, 2025.

[i.51] C. Thuillet, P. Andouard and O. Ly: "A smart card power analysis simulator". 2009 International Conference on Computational Science and Engineering, pp. 847-852, 2009.

[i.52] Tromer, E., Osvik, D.A. & Shamir, A.: "Efficient Cache Attacks on AES, and Countermeasures". Journal of Cryptology Issue 23, pp. 37–71, 2010.

[i.53] P.W. Shor: "Algorithms for quantum computation: discrete logarithms and factoring". Proceedings 35th Annual Symposium on Foundations of computer Science, 1994.

[i.54] N. Kobeissi: "Verification Theatre: False Assurance in Formally Verified Cryptographic Libraries". IACR ePrint Archive 2026/192.

[i.55] Cryspen: "libcrux".

3 Definition of terms, symbols and abbreviations

3.1 Terms

Void.

3.2 Symbols

For the purposes of the present document, the following symbols apply:

ct	Ciphertext
m	Message
par	Parameter set
pk	Public key
sk	Private key
ss	Shared secret
σ	Signature

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

DHKEM	Diffie-Hellman Key Encapsulation Mechanism
DSS	Digital Signature Scheme
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
EdDSA	Edwards-curve Digital Signature Algorithm
KAS	Key Agreement Scheme
KEM	Key Encapsulation Mechanism
ML-KEM	Module-Lattice-based Key Encapsulation Mechanism
NCSC	National Cyber Security Centre
NIST	National Institute of Standards and Technology
RSA	Rivest-Shamir-Adleman
RSAKPG	RSA Key-Pair Generation
RSASVE	RSA Secret-Value Encapsulation

4 Introduction

Traditional public-key algorithms such as RSA, Elliptic Curve Diffie-Hellman (ECDH) and the Elliptic Curve Digital Signature Algorithm (ECDSA) are known to be vulnerable to quantum attacks using Shor's algorithm [i.53]. In response to this, the US National Institute of Standards and Technology (NIST) are currently standardizing the next generation of public-key algorithms [i.23]. These algorithms include numerous Key Encapsulation Mechanisms (KEMs) and Digital Signature Schemes (DSSs).

To prevent the new standardized algorithms being vulnerable to quantum attacks such as Shor's algorithm, their cryptographic security relies on different mathematical problems to their predecessors. The mathematics can be more complex to implement than the traditional algorithms, and careful consideration will be needed to avoid introducing vulnerabilities based on subtleties in the new standards.

Secure implementations of quantum-safe cryptographic algorithms are only one small component of a larger quantum-safe system. It is important to also consider the way in which the algorithms are used in protocols and to follow current best practice in quantum-safe protocol standards. Introducing quantum-safe algorithms to a secure system is a mitigation to one cyber security threat, but it is important to also ensure other cyber security risks are not created during the transition [i.12]. Therefore, developers implementing quantum-safe algorithms should also ensure they follow best practice for developing secure systems in general [i.32], [i.31].

The present document provides developers with general guidance to aid the secure implementation of quantum-safe algorithms. Many of the principles are not unique to quantum-safe cryptography, and so several illustrative examples are drawn from traditional cryptography. The guidance is primarily aimed at secure implementations in software, although many of the recommendations are also relevant for hardware implementations. To begin, the present document describes the interfaces and general security properties required for quantum-safe algorithms. It then outlines general good practice guidance for secure implementations of cryptographic algorithms, such as validation of inputs and outputs, error handling, and the use of good randomness.

The present document then provides some background about side-channel and fault attacks and describes possible mitigations for each class of attack. Finally, it presents an overview of testing and formal verification considerations for secure implementations. In particular, it emphasizes the need to use a rigorous threat model to consciously decide the extent to which formal verification or side-channel mitigations should be applied to an implementation.

5 Interfaces

5.1 Key encapsulation mechanisms

5.1.1 Key encapsulation interface

A Key Encapsulation Mechanism (KEM) consists of a collection of parameter sets and three algorithms:

- **KEM.KeyGen**
  Input: Parameter set *par*.
  Output: Public encapsulation key *pk* and private decapsulation key *sk*, or an error.

- **KEM.Encaps**
  Input: Parameter set *par* and public encapsulation key *pk*.
  Output: Shared secret *ss* and ciphertext *ct*, or an error.

- **KEM.Decaps**
  Input: Parameter set *par*, private decapsulation key *sk* and ciphertext *ct*.
  Output: Shared secret *ss*, or an error.

NOTE 1: The parameter set *par* is sometimes considered to be a global variable rather than an explicit input to the key generation, encapsulation and decapsulation algorithms.

In a key exchange such as Elliptic Curve Diffie-Hellman (ECDH), the shared secret is derived from keys contributed by both parties in the exchange. However, in a key encapsulation mechanism the two parties have distinct roles: one party generates a key pair, and the other party uses the public encapsulation key to generate a ciphertext that encapsulates a shared secret which the first party can then recover from the ciphertext using their private decapsulation key.

EXAMPLE 1: The RSA Secret-Value Encapsulation (RSASVE) scheme from NIST SP 800-56B Rev. 2 [i.18] is a key encapsulation mechanism based on the basic RSA encryption primitive. One party generates an RSA key pair. The other party selects a random shared secret and encrypts it using the first party's public RSA key to produce the ciphertext. The first party decrypts the ciphertext using their private RSA key to recover the shared secret.

NOTE 2: In contrast to ECDH, the shared secret output by RSASVE is entirely controlled by the encapsulating party. Consequently, when RSASVE is used in the key agreement scheme KAS1 from [i.18], the RSASVE shared secret is combined with a random nonce from the decapsulating party during key derivation to ensure that the final keying material depends on contributions from both parties.

For some key encapsulation mechanisms, decapsulation involves a key derivation step or ciphertext check that needs the public encapsulation key. In those cases, the private key either contains a copy of the public key or can be used to recompute the public key.

EXAMPLE 2: The Diffie-Hellman Key Encapsulation Mechanism (DHKEM) specified in IRTF RFC 9180 [i.8] includes both the ciphertext and public encapsulation key when deriving the KEM shared secret from the Diffie-Hellman shared secret. The public key can be recomputed from the private key.

NOTE 3: Decapsulation in DHKEM can be made more efficient by including a copy of the public key in the private key rather than requiring it to be recomputed each time. This changes the format of the private key and could potentially cause issues if the private key is malformed (see clause 5.1.3).

5.1.2 Errors

Each of the key generation, encapsulation and decapsulation algorithms can return an error. These errors can be caused by a malformed input, including invalid parameters; the failure of an internal algorithm check, potentially due to an implementation mistake elsewhere; or an error in a component relied on by the algorithm such as random bit generation.

EXAMPLE 1: The RSA Key-Pair Generation (RSAKPG) family of key generation algorithms from NIST SP 800-56B Rev. 2 [i.18] include a pair-wise consistency check to detect badly generated key pairs. If the pair-wise consistency check fails, the key generation algorithm will return an error.

NOTE: Valid RSA key pairs will not fail the pair-wise consistency check so a correctly functioning RSA key generation algorithm is not expected to return a pair-wise consistency error. The check is intended to detect and prevent issues caused by a faulty implementation or invalid parameters (see clause 6.3.1).

EXAMPLE 2: The RSASVE decapsulation algorithm (RSASVE.Generate) from [i.18] checks that the ciphertext corresponds to an integer in the correct range. If the range check fails, the decapsulation algorithm will return an error.

EXAMPLE 3: The RSASVE encapsulation algorithm (RSASVE.Recover) from [i.18] uses a random bit generator to derive the shared secret that is encapsulated by the ciphertext. If the random bit generator returns a catastrophic error due to a failure of the entropy source, the encapsulation algorithm can return an error.

5.1.3 Decapsulation failures

The shared secret output by encapsulation might not always match the shared secret output by decapsulation; that is, for a key pair (*sk*, *pk*) it is possible to have (*ss*, *ct*) = KEM.Encaps(*par*, *pk*) and *ss*' = KEM.Decaps(*par*, *sk*, *ct*) with $ss \neq ss'$.

NOTE 1: This is not the same as a decapsulation error where decapsulation does not output a shared secret. In a decapsulation failure, encapsulation and decapsulation both output shared secrets, but they are different.

Decapsulation failures can occur when the inputs to encapsulation or decapsulation are malformed, including invalid parameters, or when there are mistakes in the implementation.

EXAMPLE 1: The RSASVE encapsulation and decapsulation algorithms from [i.18] expect correctly generated keys and can output different shared secrets when used with a mismatched key pair; for example, when the public and private keys use the same RSA modulus *n*, but the private exponent *d* does not correspond to the public exponent *d*.

NOTE 2: The pair-wise consistency check in the RSAKPG key generation algorithms [i.18] detects malformed or mismatched key pairs by testing for decapsulation failures. (See also clause 6.3.1).

EXAMPLE 2: Incorrectly implemented elliptic curve arithmetic can return the wrong ECDH shared secret for specific inputs without necessarily causing an error (see, for example, CVE-2017-8932). This would lead to a decapsulation failure if used in DHKEM [i.8].

The design of some key encapsulation mechanisms means that they can suffer from random decapsulation failures even when they have been correctly implemented and all inputs are valid. For well-chosen parameters, random decapsulation failures will happen with negligible probability.

EXAMPLE 3: The Module-Lattice-based Key Encapsulation Mechanism (ML-KEM) from NIST FIPS 203 [i.14] has a random decapsulation failure rate of 2^{-165} with parameter set ML-KEM-768.

NOTE 3: ML-KEM decapsulation includes a re-encryption check to detect invalid ciphertexts. If the re-encryption check fails, decapsulation will return a different shared secret rather than an explicit decapsulation error. This is the intended behaviour and not a random decapsulation failure.

5.2 Digital signature schemes

5.2.1 Digital signature interface

A Digital Signature Scheme (DSS) consists of a collection of parameter sets and three algorithms:

- **DSS.KeyGen**
  Input: Parameter set *par*.
  Output: Public verification key *pk* and private signing key *sk*, or an error.

- **DSS.Sign**
  Input: Parameter set *par*, private signing key *sk* and message *m*.
  Output: Signature σ, or an error.

- **DSS.Verify**
  Input: Parameter set *par*, public verification key *pk*, message *m* and signature σ.
  Output: Accept or reject, or an error.

NOTE: The parameter set *par* is sometimes considered to be a global variable rather than an explicit input to the key generation, signature generation and verification algorithms.

For critical applications, it is sometimes recommended that the signature generation algorithm checks that the signature verifies correctly and outputs an error if verification fails (see clause 3.2 in [i.13]). In those cases, the private key either contains a copy of the public key or can be used to rederive the public key.

5.2.2 Errors

The key generation and signature generation algorithms can return an error. These errors can be caused by a malformed input, including invalid parameters; the failure of an internal algorithm check, potentially due to an implementation mistake elsewhere; or an error in a component relied on by the algorithm such as random bit generation.

EXAMPLE 1: Key generation for the Elliptic Curve Digital Signature Algorithm (ECDSA) in NIST FIPS 186-5 [i.13] converts the output of a random bit generator into an integer for use as the private key. If the rejection sampling variant is used (clause A.2.2 in [i.13]), this conversion will fail when the integer is not in the expected range and key generation will return an error.

EXAMPLE 2: The ECDSA signature generation algorithm (clause 6.4.1 in [i.13]) includes a step that computes a modular inverse. This can fail if the ECDSA domain parameters are invalid; for example, if the order *n* of the base point is not prime. In that case, the signature generation algorithm will return an error.

Errors that occur in the verification algorithm are often handled by rejecting the signature rather than returning an explicit error. However, it still might be possible for a failure in a subroutine to cause verification to return an explicit error rather than rejecting the signature.

EXAMPLE 3: The ECDSA verification algorithm (clause 6.4.2 in [i.13]) checks that the signature corresponds to a pair of integers in the correct range. If the range check fails, the verification algorithm will reject the signature.

EXAMPLE 4: The ECDSA verification algorithm (clause 6.4.2 in [i.13]) includes a step that computes a modular inverse. This can fail if the ECDSA domain parameters are invalid; for example, if the order *n* of the base point is not prime. In that case, the verification algorithm could return an error rather than rejecting the signature.

NOTE: NIST FIPS 186-5 [i.13] includes a general requirement that any verification failures, including errors, are treated as rejections of the signature. Conversely, ISO/IEC 14888-3 [i.10] does not have this requirement.

5.2.3 Pre-hashing

In general purpose digital signature schemes, the signature generation algorithm takes a variable length message as input. If a hardware device is being used to securely store the private signing key and perform signature generation, it might not be feasible to transfer long messages to the device for signing; for example, if the hardware device has limited resources or bandwidth. In those cases, it can be preferable to use the digital signature scheme to sign a hash of the message rather than the message itself.

Pre-hashing the message is often part of the protocol layer, but some digital signature schemes include variants that integrate pre-hashing at the algorithmic layer.

EXAMPLE 1: The EdDSA digital signature scheme [i.7], [i.13] has a pre-hash variant HashEdDSA. Distinct byte strings are included when computing the internal message hash to provide separation between the pure EdDSA and pre-hash HashEdDSA variants. Specifically, when used with the Ed25519 parameter set, EdDSA computes:

$$digest = \text{Hash}(R \,||\, Q \,||\, m),$$

where R is the encoding of a point from the first half of the signature, and Q is the encoding of the public key, whereas HashEdDSA computes:

$$digest = \text{Hash}(str \,||\, R \,||\, Q \,||\, \text{PreHash}(m))$$

for a specified byte string *str*.

For some digital signature schemes, the internal hash computation in the signature generation algorithm does not involve the private signing key and can be cleanly separated from the rest of the signature calculation. This allows the internal message hash to be performed outside the hardware device, and the result can be passed to the device so that it can complete signature generation using the private signing key stored on the device.

EXAMPLE 2: The first step of the ECDSA signature generation algorithm in [i.13] computes $digest = \text{Hash}(m)$. This can be cleanly separated from the rest of the signature calculation.

NOTE: The EdDSA signature generation algorithm [i.7], [i.13] hashes the message in a way that cannot be cleanly separated from the rest of signature generation so it is not possible to use the same approach in this case.

5.2.4 Context strings

If a protocol reuses the same private signing key to sign messages across different sessions of the protocol or for different purposes within the protocol, this can lead to replay or cross-protocol attacks. One mitigation technique is to include a context string which is specific to a given session or use case when signing the message.

Context strings are often part of the protocol layer, but some digital signature schemes allow context strings as optional input to the signature generation and verification algorithms.

EXAMPLE: Some variants of the EdDSA digital signature scheme [i.7], [i.13] accept a context string *ctx* as optional input to signature generation and verification. Specifically, for the Ed448 parameter set, EdDSA computes:

$$digest = \text{Hash}(str \,||\, \text{len}(ctx) \,||\, ctx \,||\, R \,||\, Q \,||\, m)$$

for a specified byte string *str*, where R is the encoding of a point from the first half of the signature, and Q is the encoding of the public key. The default for *ctx* is the empty string.

NOTE: The pre-hash HashEdDSA variant of EdDSA accepts a context string for the Ed25519 parameter set, but pure EdDSA does not.

6 Best practice guidance

6.1 Reuse existing implementations

If an existing cryptographic implementation has been independently reviewed and rigorously tested, then it is generally better to reuse that implementation than to develop a new implementation. Open-source projects that are well-organized and well-supported will benefit from greater community scrutiny and are more likely to receive regular updates to fix vulnerabilities. Implementations that have been formally verified can increase confidence further.

However, existing implementations might not always be suitable for the intended application.

EXAMPLE 1: Existing implementations require more resources than are available.

EXAMPLE 2: Existing implementations do not support the required parameter sets or algorithm variants.

EXAMPLE 3: Existing implementations do not include the required side-channel mitigations.

6.2 Perform input validation

6.2.1 Check input formats

It is good practice for any implementation to validate input data so that unexpected data does not cause unpredictable behaviour [i.1]. At best, unexpected data can lead to the implementation returning inconsistent outputs. At worst, it can be an exploitable security vulnerability.

The simplest form of input validation checks that the input has the expected format. For cryptographic implementations, this is often part of the algorithm specification.

EXAMPLE 1: The RSASVE decapsulation algorithm (RSASVE.Recover) from [i.18] checks that the ciphertext is a byte string of the expected length. If the check fails, the decapsulation algorithm returns an error.

It is important that an implementation includes all input validation checks from the algorithm specification.

EXAMPLE 2: The ECDSA verification algorithm (clause 6.4.2 in [i.13]) checks that the signature corresponds to a pair of integers in the expected range. If the check fails, the verification algorithm rejects the signature.

NOTE: An implementation of ECDSA verification that does not perform this check could allow an attacker to use (0, 0) as a valid signature for any message with any public key (see, for example, the "psychic signatures" vulnerability CVE-2022-21449 [i.42]).

6.2.2 Check cryptographic inputs

A more sophisticated form of input validation checks that cryptographic inputs such as private keys, public keys and ciphertexts have the expected properties. Cryptographic validation is generally not part of the algorithm specification, but it can often be recommended or required before passing the cryptographic inputs to the algorithm.

EXAMPLE 1: Partial validation of elliptic curve public keys includes checking that the public key corresponds to a non-trivial point on the elliptic curve with the expected encoding. Full validation also checks that the point has the expected order.

NOTE 1: An implementation of a static-ephemeral ECDH key agreement (clause 6.2.2.2 in [i.17]) that does not validate public keys could allow an attacker to use malformed ephemeral public keys to recover the other party's static private key (see, for example, CVE-2024-48930).

EXAMPLE 2: Partial validation of RSA public keys as described in NIST SP 800-89 [i.19] involves checking that the modulus and exponent are odd integers in the expected ranges, checking that the modulus is not a prime power, and checking that the modulus does not have any small factors.

NOTE 2: Full validation of an RSA public key is not possible without the private key.

The disadvantage of cryptographic validation is that it is often computationally expensive and can lead to denial of service attacks when used inappropriately (see, for example, CVE-2023-6237 and CVE-2024-41996). It can be safe to omit cryptographic validation if the impact of invalid input is sufficiently low or if its validity can be assured in other ways.

EXAMPLE 3: Validation of a public key might not be necessary in an ephemeral-ephemeral ECDH key agreement (clause 6.1.2.2 in [i.17]) as the keys will only be used for a very short period of time.

EXAMPLE 4: Validation of a private key before it is used in signature generation might not be needed if the key generation ensures validity and private keys are stored in a way that prevents modification.

6.3 Perform output validation

6.3.1 Include consistency checks

Some cryptographic algorithms include consistency checks on the output as part of the specification. These are often intended to detect and prevent issues caused by mistakes elsewhere in the implementation. They can also include intermediate cryptographic checks that cannot be applied directly on the cryptographic input.

EXAMPLE 1: The RSAKPG family of key generation algorithms from [i.18] check that the private exponent is not unexpectedly small and do not output the key pair if this check fails.

NOTE 1: If the private exponent is sufficiently small, it can lead to practical attacks on algorithms that use it.

EXAMPLE 2: Implementations of the X25519 key exchange in [i.5] are required to accept all public keys that are encoded as byte strings of the correct length, even public keys that correspond to points of small order. Instead, X25519 includes an optional check that the shared secret is not zero.

NOTE 2: Returning zero as the shared secret is not a security issue for the key exchange itself, but the shared secret no longer depends on contributions from both parties which could cause problems in protocols that expect this property. In particular, the shared secret check is required when X25519 is used in DHKEM [i.8].

EXAMPLE 3: The ECDSA signature generation algorithm in [i.13] checks that values of r and s in the resulting signature (r, s) are both non-zero and does not output the signature if this check fails.

NOTE 3: Correctly implemented ECDSA signature verification will reject signatures (r, s) if at least one of r or s is zero.

More generally, it is sometimes recommended that signature generation algorithms check that the signature will verify correctly before output as a way of identifying signature generation mistakes (see, for example, clause 3.2 in [i.13]).

6.3.2 Use expected output formats

Cryptographic algorithms can often have multiple representations or formats for keys, ciphertexts and signatures. Supporting different formats increases the complexity of an implementation. Some formats can be more vulnerable to side-channel or fault attacks than others.

An implementation that returns the output in a format that is not widely supported will limit interoperability. If the output format is not valid, then this can cause issues with other implementations that do not perform input validation correctly.

EXAMPLE: PKCS #1 [i.6] specifies two different representations for RSA private keys. If an RSA private key parser expects keys to use the CRT representation, then input validation can fail unexpectedly when passed a key in the non-CRT representation or in a CRT representation that does not contain all of the expected values (CVE-2025-22865).

NOTE: The RSAPrivateKey type recommended in [i.6] includes components from both representations.

6.4 Prevent leakage of intermediate values

6.4.1 Zeroise intermediate values after use

Intermediate values from key generation, encryption, decryption or signature generation algorithms can potentially reveal sensitive information about private keys or plaintexts. If these values remain in memory after the cryptographic calculation, they might be recoverable by attackers with the right access. This includes local copies of private keys or locally stored portions of the plaintext.

EXAMPLE 1: A non-cryptographic vulnerability such as Heartbleed (CVE-2014-0160) that leaks contents from memory used by a cryptographic implementation can expose sensitive values after they have been used.

Some algorithm specifications recommend or require that all intermediate values are reset to zero before returning the output, but this is not always the case.

EXAMPLE 2: The specification of the RSA-OAEP encryption algorithm in NIST SP 800-56B Rev. 2 [i.18] requires that all intermediate values are reset to zero before returning the ciphertext.

NOTE: This is not required by the specification of RSA-OAEP from IETF RFC 8017 [i.6].

6.4.2 Limit access to intermediate functions

Cryptographic algorithms are often built from lower-level primitives. If an implementation allows direct access to the lower-level primitives, or other intermediate functions, with the same inputs then this will reveal the intermediate values output by those primitives or functions.

In particular, accessing lower-level primitives can avoid cryptographic input validation or other consistency checks.

EXAMPLE 1: The Elliptic Curve Integrated Encryption Scheme (ECIES) [i.11] uses ECDH as a component. If the implementation allows ECDH to be called directly with the same static private key, then this could potentially bypass public key validation and allow an active attacker to use malformed public keys to recover the private key (CVE-2023-49292).

There are some scenarios where access to intermediate values or intermediate functions can be necessary for the deployment model. This will generally be safe provided that the intermediate values are not derived from private data and the intermediate functions do not skip any security-critical checks.

EXAMPLE 2: The computation of the message hash in the ECDSA signature generation algorithm from [i.13] can be separated from the remainder of the signature generation algorithm as it does not involve the private key or skip any security-critical checks (see clause 5.2.3).

6.5 Handle errors gracefully

6.5.1 Include specified error handling

As discussed in clauses 5.1.2 and 5.2.2, cryptographic algorithms can return errors due to malformed inputs, mistakes in the implementation, or other failures. Errors that are not handled carefully can lead to unexpected behaviour or leak sensitive information.

Algorithm specifications often include requirements on how to handle errors or other algorithm failures.

EXAMPLE 1: The RSAKPG family of key generation algorithms from NIST SP 800-56B Rev. 2 [i.18] restart if computation of the private RSA exponent fails or if the check on the exponent size fails.

If the final pair-wise consistency check fails (see clause 5.1.2), the key generation algorithms return an explicit error.

EXAMPLE 2: The ECDSA key generation algorithms from NIST FIPS 186-5 [i.13] return an explicit error if the call to the random bit generator fails.

However, this is not always the case so it is important to consider errors that might arise in addition to those given in the algorithm specifications.

EXAMPLE 3: The EdDSA key generation algorithm from NIST FIPS 186-5 [i.13] does not check that the call to the random bit generator was successful.

Some algorithm specifications assume that errors are indicated through a status value that is part of the algorithm output rather than being raised explicitly. When implementing error handling in this way, it is important that algorithms do not also return the output that might have caused the error since this could be inadvertently used elsewhere.

EXAMPLE 4: The output of the prime generation algorithms in NIST FIPS 186-5 [i.13] is a status indicator and a pair of primes (p, q). If any internal checks fail, the algorithms return an error status and the fixed pair (0, 0).

NOTE: The RSAPKG family of key generation algorithms in NIST SP 800-56B Rev. 2 [i.18] use the prime generation algorithms from NIST FIPS 186-5 [i.13], but do not explicitly check that the prime generation completed successfully. If a prime generation failure also returns (0, 0), then the step in the key generation that attempts to compute the private RSA exponent will always fail.

6.5.2 Avoid leaking information through errors

If an error is caused by a check or other failure that depends on intermediate values derived from the private key or plaintext, then the error itself can reveal sensitive information. It is important to avoid leaking this information via error messages or timing data.

EXAMPLE: The RSA-OAEP decryption algorithm [i.6] checks that the recovered message has the correct encoding and returns an error if this fails. If the error handling leaks information about which specific check failed, then this can be used in an active plaintext-recovery attack [i.43] (see, for example, CVE-2020-26939).

NOTE: IETF RFC 8017 [i.6] contains a warning about being able to distinguish different error conditions.

6.5.3 Indicate the severity of errors

For failures that are expected to occur with reasonable probability through normal use, it is usually safe to retry the computation with different random values where this is possible. On the other hand, failures that would never be expected to occur through normal use are likely to be caused by critical implementation mistakes or malformed inputs, potentially provided by an attacker. In that case, it is important to raise an error that indicates the severity of the failure.

EXAMPLE: The specification of the Blind function from the Blind RSA signature scheme [i.9] checks that the encoded message is co-prime to the modulus and returns an "invalid input" if this check fails. However, this is equivalent to finding a non-trivial factor of the RSA modulus so a failure is likely caused by the signer using a weak or malformed key pair.

6.6 Use good randomness

6.6.1 Use a secure random bit generator

Cryptographic algorithms require a good source of randomness. Randomness that is predictable or biased can critically undermine the security of the algorithm. Recommendations for generating and testing the quality of randomness can be found in [i.20] for example.

Algorithm specifications often include requirements on the choice of random bit generator or explicit instructions on how to derive a sequence of random bits from a seed value. It is important that implementations do not attempt to deviate from these.

EXAMPLE 1: An implementation of ECDSA signature generation that used a fixed ephemeral signing value for all signatures allowed the private key to be recovered from a pair of signatures [i.33].

EXAMPLE 2: The use of the Mersenne Twister random bit generator in two cryptocurrency wallets allowed attackers to recover private keys and steal funds (see CVE-2023-31290 and CVE-2023-39910).

6.6.2 Use good entropy

Even when an implementation uses a cryptographically secure random bit generator, the output can still be predictable if it is not instantiated and reseeded with good entropy.

EXAMPLE 1: A change in the entropy collection for the random number generator in Debian led to predictable output (CVE-2008-0166).

EXAMPLE 2: Reuse of entropy to seed the random bit generator or duplication of the random bit generator's internal state led to predictable output (CVE-2025-7394).

EXAMPLE 3: Limited entropy during RSA key generation led to RSA moduli that shared a prime factor which could then be easily recovered (see, for example, [i.36], [i.41], [i.35] and [i.38]).

6.6.3 Perform de-randomization correctly

A technique that is commonly used to defend against entropy failures is de-randomization: the random bits needed by the algorithm are generated deterministically from the algorithm input.

De-randomization can be part of the algorithm specification (see, for example, [i.7]). If it is being used to replace a call to a random bit generator, then it is important to ensure that de-randomization produces the expected number of bits and does not introduce any bias.

EXAMPLE: A de-randomized implementation of ECDSA signature generation did not use full-sized ephemeral signing values with NIST P-521. This allowed the private key to be recovered from approximately 60 signatures (CVE-2024-31497).

An alternative approach is hedging which derives the random bits deterministically from the algorithm input and fresh entropy (see, for example, [i.44]).

7 Side-Channels and Fault Attacks

7.1 Introduction

7.1.1 Concept

It is possible to closely observe devices performing cryptographic calculations and deduce secret information by analysing the measurements taken. The existence of such an extraction route for secrets is called a "side-channel", and the information is said to be "leaking" from the device. Examples of potential side-channels are the time taken or the amount of power drawn during a calculation.

A related concept to side-channels is fault attacks. In this case, the target device is subjected to some form of interference or interruption while performing cryptographic calculations in the hope of inducing a fault in the device's internal state, such as flipping a bit or zeroing a register. If successful, the resulting output from the faulty calculation might reveal information about the secret key.

Any observable feature can lead to a side-channel if sufficiently coupled to the secret information held within the device. Different adversary threat models permit different classes of side-channel attacks, which in turn will inform the mitigations that a particular implementation might need to employ. For many use cases, side-channel and fault attacks will not be a viable approach for an adversary. It is not possible to describe all possible ways a device might leak information: the present document covers the most common side-channels and potential mitigations.

The present document will cover protocol-level adaptations (such as regular key rotation) and algorithm-level adaptations (altering the implementation of algorithms to a "mathematically equivalent set of steps" as permitted by [i.14]). For some threat models additional system-level engineering countermeasures, such as tamper-proofing, may be appropriate. These are outside the scope of the present document.

Cryptographic operations can be implemented in either software or hardware. Much of this general guidance applies equally to both: it should be stressed that side-channel leakage can be introduced through the cryptographic implementation, any compilation or synthesis process, the characteristics of the device that will perform the cryptographic operations, or a combination of any of these. Therefore, mitigations will need to be evaluated in the context in which they will be deployed.

EXAMPLE 1: Cryptographic software can be written to be "constant time" (see clause 7.2 below), but the implementation of low-level operations varies between architectures.

EXAMPLE 2: Cryptographic implementations can have algorithmic masking applied to reduce the dependence of power draw on secret intermediate values (see clause 7.3 below), but the placing of registers next to each other in physical memory can lead to the "cancelling out" of the masks.

An adversary threat model will need to consider a variety of parameters, including:

- Physical access to the device.
- Potential duration of observation.
- The level of control over the device.

7.1.2 Physical Access

An adversary's physical access to a device may range from remote observation through to full lab instrumentation. A further consideration is whether the adversary can make permanent modifications to the device, such as removing or introducing circuit board components. In general, the more permissive the physical access, the greater the number of side-channel attacks that will be viable.

7.1.3 Duration

The length of time an adversary can monitor a device for directly informs the number of calculations that can be observed. In the worst case, measurements made during a single iteration of a cryptographic algorithm can reveal the full secret key (a "single trace attack"). More typically, multiple iterations will be required together with statistical analysis of the collected traces. Reducing the number of observable iterations for a particular secret key can reduce the likelihood of success for some attacks.

7.1.4 Control

An adversary's ability to trigger calculations, and to observe and control inputs, outputs and cryptographic secrets will impact the potential effectiveness of the attacks they are able to carry out.

If an adversary can only passively observe calculations, this may increase the length of time taken to collect the required traces and can introduce noise into measurements. An adversary able to actively trigger calculations will have an advantage.

Similarly, an adversary able to control the inputs to an algorithm may be able to tune these values in a way that offers an advantage over an adversary passively observing inputs or outputs.

The greatest level of control an adversary may enjoy is being able to alter the cryptographic secret on a device. While not useful for recovering that particular secret, side-channel attacks can be "profiled" or "non-profiled". In a profiled attack, there is a training stage where traces are generated using a key known to the attacker on the target device, or a very similar one, and then an attack stage where an unknown key is recovered. In a non-profiled attack, an unknown key is directly recovered from collected traces. After the training stage, profiled attacks typically require many fewer traces than non-profiled attacks, but the adversary requires significant control over the device used to generate the training data.

7.2 Timing Analysis

7.2.1 Concept

Timing attacks use variations in the execution time of a cryptographic routine or subroutine to recover information about secret values.

EXAMPLE: A password checking subroutine could perform a string comparison between the stored password and an input and terminate as soon as a mismatch is found. An adversary able to repeatedly guess passwords and accurately measure the time taken for the subroutine to return could recover the password character by character. A simple fix is to prevent the early return of the subroutine, eliminating the timing dependency on the secret password.

Timing dependencies in cryptographic code can occur at several levels:

- At the implementation level from secret-dependent branching.
- At the operation level from variable time operations and secret-dependent memory addressing.
- At the processor instruction level from speculative execution and branch prediction.

NOTE: Attacks that include speculative execution and branch prediction are outside the scope of the present document.

The number of traces required to reveal a secret will depend on the magnitude of the timing difference and the signal to noise ratio in the adversary's measurements. Adversaries able to precisely trigger the cryptographic calculation and directly observe when it is finished (e.g. querying a local process) will have a significant advantage over an adversary who infers these events via secondary effects (e.g. remote observation of ClientHello and ServerHello packets during a TLS handshake).

7.2.2 Secret-Dependent Branching

In cryptographic algorithm specifications, there are often branching paths where the control flow is dependent on the secret value. If the branches take differing lengths of time to execute, then an adversary able to observe the duration of calculation may be able to infer information about the secret value.

EXAMPLE 1: In the RSA algorithm, modular exponentiation is often implemented via square-and-multiply, where the base value is repeatedly squared and then multiplied into an accumulator if the secret exponent bit is set at that position. If a multiplication operation is *only* carried out at the positions where the bits are set, the time taken for the calculation can reveal the total number of bits set in the secret exponent. Of course, this is not sufficient to recover the exponent.

More powerful observations can be made if the branching depends on an input value and the secret value. If the adversary is able to collect traces using different inputs, then they may be able to carry out statistical analysis to infer the secret value itself.

EXAMPLE 2: During RSA exponentiation, if modular reductions are carried out only when needed, then the adversary can collect many observations with varying input and recover the secret exponent one bit at a time. This is done by guessing the exponent bit and then dividing input values into groups based on whether a modular reduction would occur. When a statistically significant difference in the distributions of the two timing groups is observed, the exponent bit has been guessed correctly. For example, the median or mean of the two different groups could differ by a statistically significant amount. The attack can proceed bit by bit to recover the full key.

7.2.3 Variable Time Operations

Timing variations can also arise from applying variable time operations to secret data, such as a modular reduction or a division operation where the length of time taken is correlated with the secret value being processed.

EXAMPLE: A secret-dependent timing variation introduced by the division operator occurred in many early implementations of the post-quantum lattice-based scheme Kyber (since standardized by NIST as ML-KEM) [i.28].

7.2.4 Secret-Dependent Memory Addressing

If the adversary is able to monitor and influence the cache usage of a device, then secret-dependent memory addressing can also lead to timing vulnerabilities.

EXAMPLE: AES implementations are often optimized by the introduction of lookup tables to speed up the final two rounds. Purging part of a lookup table from shared memory and measuring whether this has an impact on calculation duration tells an adversary whether that part of the lookup table is used, providing direct information about the AES key bytes [i.52].

7.2.5 Mitigations

Cryptographic code should be *isochronous*, which means it avoids secret-dependent branching or addressing and the application of variable time instructions to secret-dependent data.

NOTE: The property of being *isochronous* does not mean that the entire calculation is required to run in constant time: some algorithms, such as ML-DSA, explicitly include loops with variable numbers of iterations. Critically, nothing sensitive is leaked by the number of iterations required.

This may mean implementing bespoke isochronous implementations for core operations such as modular reduction or the insertion of dummy operations, such as multiplying by 1 when the exponent bit is 0 during a square-and-multiply exponentiation. Isochronous code is unlikely to be the quickest way to implement a given operation and so care should be taken to ensure that compilers do not reintroduce non-isochronous code as part of their optimization process.

Formal analysis has shown benefits in detecting secret-dependent paths or confirming secret independence of cryptographic code.

EXAMPLE: The KyberSlash vulnerability was discovered during formal analysis of an ML-KEM implementation [i.28].

Isochronous code will mitigate many cache-based and microarchitectural timing attacks.

7.3 Power and EM Analysis

7.3.1 Concept

The amount of power drawn by a device is correlated with the operations it is performing, including the data being processed in registers. For data, this correlation is usually with the value's Hamming weight or its Hamming distance to the previous value in the register. If an adversary can gain sufficiently close access to monitor the power consumed by a device, they can learn detailed information about the internal workings of the device including intermediate values of cryptographic algorithms, potentially revealing secret keys [i.40].

This will usually be done by instrumenting a device under laboratory conditions to provide the cleanest possible power traces. An adversary will ideally be able to probe the power line directly and will also remove any capacitors that induce dampening effects on the circuit. However, any measurement related to the power draw of the device can reveal sensitive information. Changes in a component's power draw induce fluctuations in the electromagnetic field around this component. Therefore, traces can also be collected via an EM probe placed near a component, removing the requirement to be able to probe the circuit directly. These requirements can be relaxed even further: attacks have been demonstrated by using high speed video of a power LED on a smartcard reader over a distance of several metres [i.47], and by monitoring Bluetooth® signals from a device where the RF and cryptographic components share a power supply [i.30].

EXAMPLE 1: For modular exponentiation within RSA implemented as described in clause 7.2.2, if an adversary can distinguish between square-only and square-and-multiply operations from the power trace, then they can read the secret exponent off from a single calculation ("simple power analysis").

NOTE: Single trace attacks may also be possible if the Hamming weights of small parts of the cryptographic secret can be inferred from one trace. This may be easier in algorithms where secret coefficients take values in {-1, 0, 1}.

More typically, multiple traces will be collected with varying inputs. These traces will then undergo signal processing and alignment so that the same operation is occurring at the same time point within each trace. The adversary will then guess small parts of the key and carry out statistical tests on the collected traces to see if there is a detectable difference or correlation of the distribution of the observed power draw with the predicted power draw given their key guess. This is referred to as Differential or Correlational Power Analysis.

EXAMPLE 2: In AES the output from a single S-Box is S[P⊕K], where P is a single byte of plaintext and K is a single byte of key. Knowing the plaintext for each trace, the adversary can calculate the output of the S-Box for all 256 possible values of K and then correlate each of these guesses with the power trace values. Due to the non-linearity of the AES S-Box, the correct value of K will demonstrate significantly higher correlation than incorrect values.

For software implementations, this correlation is typically with the Hamming weight (the number of bits set) of a value in a register. For hardware implementations, the correlation is typically with the Hamming distance between a value in a register and the value it is overwriting (the number of bits that differ between the two values).

Identifying the correct leakage model (what value is leaking and locating it in the power traces) can be a major difficulty for an adversary. When masking techniques have been employed (see clause 7.3.2 Mitigations below), it can be necessary to combine multiple samples within each trace, exponentially increasing the potential search space. The application of machine learning techniques such as neural networks has shown promise both in reducing the level of trace pre-processing required and automatically identifying the correct leakage model and locations [i.27].

The number of traces required for a key recovery is dependent on an enormous variety of factors outside of any specific countermeasures or analysis techniques employed. These include the calibre of the probing and recording equipment, the proximity of the adversary, the complexity of the microprocessor architecture and noise induced by environmental factors such as temperature variation.

7.3.2 Mitigations

As mentioned in clause 7.1.1, the present document will focus on protocol and algorithmic level mitigations. Protocol level mitigations can reduce the number of traces an adversary can collect for a particular secret by enforcing regular key rotation after a certain period of time or number of algorithm iterations using that secret key. If applicable, this can be an effective and cheap mitigation.

Algorithmic level mitigations attempt to limit the adversary's ability to correlate the power draw of the device at a particular time from the sensitive values it is processing. Some generic protections include randomizing the timing of operations or decoupling the power dependence from the secret values via masking, but the application of these to a specific algorithm requires careful analysis of the secret-dependent parts. Different analysis is required for Simple Power Analysis and Differential/Correlational Power Analysis attacks.

Additionally, it can be difficult to verify that mitigations have been effective, even when leakage simulations are employed. The overall strength of mitigation is dependent on the weakest protected part of the algorithm [i.50] and device-specific leakage can arise from architecture quirks. This means expert analysis of devices under laboratory conditions may be required to provide full assurance of the effectiveness of deployed mitigations.

7.3.3 Randomization

Altering the order of operations can be an effective way to mitigate some Simple Power Analysis attacks by removing the adversary's ability to identify which value is being operated on at a particular time. This can be done by randomly permuting a loop or array, where the algorithm functionally permits this.

EXAMPLE: AES S-Box lookups are done bytewise and independently and so can be performed in any order.

Randomization does not fully protect against Differential/Correlational Power Analysis, but it does increase trace requirements by a factor of $O(\sqrt{n})$ where n is the number of iterations in the loop.

7.3.4 Masking

The dependence of a device's power consumption on sensitive intermediate values can be mitigated algorithmically by rewriting the algorithm to operate on shares of these values, or equivalently "masking" the values. Masking introduces significant overheads to the complexity of implementing and testing an algorithm and to the memory, time and randomness requirements of the deployed implementation. Fresh masks, unknown to the adversary, are generated for each iteration of the algorithm and may need to be refreshed during the algorithm's running.

Different masking approaches are required to most efficiently protect different operations within a cryptographic algorithm. Additional operations and randomness will be required to safely transition between different types of masking and to maintain the functional correctness of the algorithm.

EXAMPLE 1: Boolean masking: $x \rightarrow (s_1, s_2)$ where $s_1 \oplus s_2 = x$, or, equivalently, $x \rightarrow (x \oplus m, m)$.

EXAMPLE 2: Additive masking: $x \rightarrow (s_1, s_2)$ where $s_1 + s_2 = x$, or, equivalently, $x \rightarrow (x - m, m)$.

EXAMPLE 3: Multiplicative masking: $x \rightarrow (s_1, s_2)$ where $s_1 \times s_2 = x$, or, equivalently, $x \rightarrow (x/m, m)$.

The number of random masks applied to each value is called the masking order d, and the algorithm will now operate on at least $d + 1$ values. An implementation using d^{th} order masking should require $O(n^{d+1})$ traces to recover the secret key (where n is the number of traces required for an attack on an unmasked implementation) and an adversary will need to identify the correct d samples to combine and correlate against putative leakage models. For even small masking orders (d = 1 or 2), this will often make the number of traces required infeasible.

NOTE: Care should be taken to ensure that compilers do not optimize out functionally redundant steps in masked operations. Additionally, device specific physical effects can cause adjacent registers to "interfere" with each other, leading to the cancellation of masks and hence the reappearance of lower order leakage.

It is possible to simulate the leakage of an implementation [i.51], but to account for more subtle physical effects, more accurate simulations will be based on leakage models specific to the device the implementation will be deployed on [i.45]. Lab testing of a device is necessary to fully assure that leakage has been successfully mitigated up to a certain number of traces. The t-test [i.26] is often used to demonstrate that masking countermeasures have been effectively implemented, up to the order of masking that has been employed.

7.4 Fault Attacks

7.4.1 Concept

Inducing a fault in a cryptographic calculation can reveal information about the secret key. This is typically most useful if the output of several faulted calculations on the same input can be collected and compared, which is called *Differential Fault Analysis* [i.29].

An adversary can induce faults in a variety of ways, typically requiring a high degree of physical access and some level of sophistication in the device's instrumentation. They could glitch the power supply to the device, either dropping and raising the voltage sharply or lowering it gradually below the comfortable operating level of the device. They could fire electromagnetic or laser pulses targeted at a specific component on the device. They may be able to interfere with the device's clock, inducing occasional fast or slow cycles. They could change the ambient temperature the device is operating at, causing stress on components. They could repeatedly access memory locations adjacent to where cryptographic variables are being stored (a "Row hammer" attack [i.39]).

It is hard to predict the impact that a particular technique will have on a given algorithm or device. The levels of both temporal and spatial precision required will depend on the target fault, which will be informed by the specifics of the algorithm. If targeting a specific bit or operation in the cryptographic calculation, the adversary may require extremely precise triggering mechanisms, entailing instrumentation at least as sophisticated as lab-based power analysis. Some algorithms, such as SLH-DSA, may be susceptible to less precisely-targeted faults [i.34], so the threshold for considering faults a viable attack vector may be lower.

Once an adversary can successfully induce faults, further analysis may be required to diagnose the type of fault that is being induced and its impact on the cryptographic algorithm.

NOTE: Using fault attacks to recover cryptographic information may not be the most efficient way to defeat protections. For example, if a fault attack can be used to induce a skipped instruction, then it will likely be more efficient to defeat a password check directly.

7.4.2 Mitigations

Tools exist to analyse implementations and identify where carefully targeted faults could reveal cryptographic secrets [i.37]. In practice, this sort of precision is hard for an adversary to achieve.

Where fault attacks are considered part of a threat model, a simple mitigation is to iterate the vulnerable calculations two or more times. The output values should then be compared and only used if they match. The overheads of inducing an identical fault in two or more places are likely to be prohibitive for an adversary. For additional security, the iterations could use independent implementations of an algorithm.

8 Testing and Formal Verification

8.1 Introduction

Typically, implementations of cryptographic algorithms are deemed functionally correct by checking they conform to a series of known-answer tests. This testing occurs on a very limited subset of all possible input-output pairs and provides no guarantee that other inputs do not map to an incorrect output according to the algorithm specification.

Formal verification tools can be used to prove the functional correctness of a cryptographic algorithm implementation for all possible inputs with respect to the algorithm specification. For cryptographic algorithm implementations, this is typically achieved by proving that the implementation is functionally equivalent to a reference implementation. That is, for the same input, the two implementations always produce the same output. Alternatively, it is possible to use verified code generation to produce formally verified code from a specification of the algorithm.

Formal verification can also be used to provide additional assurances in other parts of cryptographic algorithm and protocol design. For example, computer-aided security proofs can provide additional mathematical rigour around traditional pen-and-paper security proofs. Protocol verification tools can be used to reason about the security properties of protocols which use cryptographic algorithms as a component. These are both interesting applications of formal methods, however they are more relevant to the theoretical design rather than the implementation of cryptography and therefore are out of scope for the present document.

The present clause focuses on implementations of post-quantum cryptography in software. For hardware implementations, different considerations around testing and formal verification will apply. Commercially available formal verification tools for hardware are on average very expensive when compared to their software counterparts and require years of deep expertise to use effectively. This expense may be worthwhile for the level of assurance provided, however depending on the threat model for the implementation it may not be necessary. Further research and tool development is required before formal verification of hardware implementations will be viable for general use in production, and therefore it is out of scope of the present document.

8.2 Testing

8.2.1 Concept

Known-answer tests are used to gain confidence in algorithm implementations. Validation tests are typically provided as a small number of known input-output pairs for an algorithm. These will include pairs sampled uniformly at random from the input space and ideally pairs that cover corner cases such as testing the error handling. NIST have provided test data [i.21] for FIPS 203 [i.14], 204 [i.15] and 205 [i.16] which can be used to validate implementations.

The specifications of ML-KEM [i.14], ML-DSA [i.15] and SLH-DSA [i.16] contain external functions which form the interfaces for each algorithm. In addition to this the specifications also define internal subroutines which provide the underlying functionality. Implementations may choose to apply optimizations to these internal subroutines to meet different requirements. For example, implementations that are optimized for efficiency will have a different internal structure from those with strong side-channel protections. Verifying known-answer tests for the external interfaces provides a way to ensure each of these implementations are correct at the end even if their internal procedures differ.

Any internal functions should not be made available for external use except for testing purposes. It is possible to validate each of these internal functions against a list of known-answer tests to provide additional confidence in the implementation, although NIST have not provided these.

In addition to validation testing, it is possible to carry out fuzzing of an implementation. Fuzzing typically tests for errors or faults in the code by providing random or carefully crafted malicious inputs. This means fuzzing can be more targeted than validation testing and can be used to test for known vulnerabilities in the algorithm, or to trigger known edge cases or specific code paths. It is important to ensure that testing occurs for all code paths within an implementation to prevent vulnerabilities being introduced due to untested code.

EXAMPLE: Cryptofuzz [i.46] is an open source fuzzing tool that has found vulnerabilities in over 10 different cryptographic libraries, such as invalid Elliptic Curve signatures that successfully verify and AES-GCM implementations that allow IVs of length 0.

8.2.2 Benefits and Limitations

Testing can provide additional assurance of cryptographic algorithm implementations, by identifying errors in implementations and helping find vulnerabilities. In addition, validation tests can be used to certify that the implementation conforms to the standards, such as through NIST's Cryptographic Algorithm Validation Program (CAVP) [i.22]. Known-answer tests can be easily automated, meaning it is repeatable and fast to achieve results, even when changes are made to the implementation.

However, testing only provides limited coverage of the algorithm and does not rule out all types of attack. Conducting known-answer tests for a small number of inputs may not explore all code paths and is likely to miss edge cases that have been misimplemented.

EXAMPLE: CVE-2022-21449 is an example of where a simple test vector was missed causing a high severity vulnerability in the implementation of ECDSA in Java 15, 16, 17 and 18. Recall that an ECDSA signature is a pair of values (r, s) where both r and s need to be in the range [1, n-1] where n is the order of the base point. The implementation did not check this bound and permitted (0, 0) as a valid signature on any message, for any key. The inclusion of a simple test vector for the signature verification function with (r, s) = (0, 0), coupled with an expected outcome of verification failure, would have been sufficient to avoid this vulnerability altogether.

8.3 Functional Correctness

8.3.1 Concept

Cryptographic algorithms are based on years of analysis, including writing precise specifications, accompanying mathematical security proofs which are in some cases complemented by computer-aided security proofs, to ensure the security of their algorithms. To achieve high levels of assurance that an implementation conforms to the specification, it can be formally verified to prove functional correctness.

Formal verification occurs within a logic-based setting where the system is modelled under a set of assumptions and then tools are used to reason about the model. The assumptions made and type of logic model in use will vary depending on the type of verification being carried out and the tools in use. Some examples of assumptions [i.25] made when formally verifying functional correctness might include perfect hardware that does not experience faults; functional correctness of any libraries integrated into the codebase; guarantees that the reference implementation exactly matches the specification; and expectations that the compiler is not changing any of the functionality of the code that has been verified.

For cryptographic algorithms, functional correctness is typically proven by asserting functional equivalence of the test implementation with a reference implementation. This reference implementation is assumed to have already been scrutinized to ensure it conforms to the specification. Primarily, the assurance is provided about the source code, since the analysis is usually carried out either at the source code level or some intermediary representation of the code, prior to compilation. The reference implementation used for comparison might be written in a low-level programming language such as C, or in a cryptographic domain specific language such as Cryptol.

EXAMPLE: A 2026 paper [i.54] provided evidence that the libcrux [i.55] library did not meet the claimed level of assurance, when vulnerabilities were found in code that was reportedly formally verified. One of these vulnerabilities was due to the developers missing an error in the algorithm specification that was being used to prove functional correctness, emphasizing the need for rigorous scrutiny and auditing. This vulnerability was not discovered until approximately 18 months after it was introduced to the library.

Many tools for proving functional correctness use SMT solvers to prove that a test implementation produces the same output as a reference implementation for all given inputs. This means that the test implementation can often be automatically proven functionally equivalent to the reference implementation even if there are differences in the way the code is implemented. These tools do not always have to be used in conjunction with a reference implementation, for example they can be used to compare an implementation before and after optimizations are applied, or implementations in two different languages.

Another method for producing implementations of cryptographic algorithms that are proven to be functionally correct is to use verified code generation. This method starts with writing and verifying an algorithm specification in a proof-oriented programming language such as F*, then using associated tools compile this into executable code that is verifiably correct by construction [i.48].

8.3.2 Benefits and limitations

Proving functional correctness provides a much higher level of assurance of an implementation than testing alone, by testing the entire input space rather than a small number of inputs. This assurance is increased further if the proof is used in conjunction with a formally verified security proof for the algorithm design. However, this approach does come with several costs and limitations.

Formal verification of an implementation is expensive, both in terms of the knowledge and effort required to conduct the proof, but may also impact the performance of the resulting implementation. Implementations that are closely aligned with the specification will be easier to formally verify, but likely will lack the efficiency required for many applications.

Reducing the proof of functional correctness to a proof of functional equivalence with a reference implementation relies on the reference implementation conforming exactly to the algorithm specification. Furthermore, when changes are made to the implementation being tested, it will not necessarily be straightforward to reproduce the proof. This lack of repeatability means that often formal verification tools cannot form part of an automated testing framework, meaning manual intervention will be required each time the implementation is updated to ensure the proof of functional correctness is still valid.

It is possible to only prove full functional correctness of small parts of the implementation, to reduce the cost of carrying out the verification. This can improve proof automation and may reduce the performance overhead, however it will also reduce the guarantees provided by the proof. These trade-offs need to be made carefully and will depend on the use cases and threat model for the implementation [i.49].

EXAMPLE 1: A 2023 paper [i.24] reported that it took 12 people almost 3 years to develop a formally verified implementation of Kyber. Though the paper acknowledges that this was in part due to lack of familiarity with the tools being used and the availability of the researchers, it also notes that a similar project undertaken by the same individuals would "still require significant resource investment". Furthermore, the resulting fully verified implementation is reportedly significantly slower than alternatives written in C and assembly, taking approximately twice as many cycles to complete when tested on three different Intel CPUs.

EXAMPLE 2: The mlkem-native [i.49] project offers performance-critical components written in assembly-language for both the AArch64 and x86_64 instruction sets that are both formally verified and highly optimized. The formal verification has been used to test aggressive optimization of this code. However, the same level of assurance and optimization does not apply to the project as a whole. The code written directly in assembly-language is formally verified to be functionally correct, constant time and memory-safe, whereas the less performance-critical C code is formally verified to be memory-safe and type-safe but not functionally correct or constant time.

The alternative approach of using verified code generation is repeatable and can be automated when small changes to the specification need to be made. However, if the specification language does not provide all the required functionality, then developers may be forced to modify the code after it has been generated, losing the guarantees of the formal verification carried out pre-generation. An additional proof could be constructed to prove equivalence of this modified code and the original code, however the generated code is unlikely to be easily human-readable or straightforward to refactor.

Tools for proving functional correctness typically provide assurances about source code, which then needs to be compiled before it can be run. General-purpose modern-day compilers can produce fast optimized code when presented with well-verified source code, but are not designed or proven to maintain functional correctness of formally verified code. Any guarantees about the functional correctness are lost during compilation unless a formally verified compiler is used to preserve the confidence in the implementation. However, at the time of writing verified compilers are only available for a small number of special purpose languages, such as Jasmin, and do not provide many optimizations. This will often mean that the resulting code will be much slower than alternatives, although it may be beneficial where side-channel mitigations have been implemented. When the source code of an implementation has been formally verified as functionally correct, then using a formally verified compiler will prevent the loss of this assurance in the resulting executable code.

Finally, it is worth noting that the tools for proving functional correctness are themselves software that may contain vulnerabilities, so it is impossible to be completely confident in the results of these proofs.

8.4 Memory-safety and Type-safety

8.4.1 Concept

Memory-safe code prevents the occurrence of errors and vulnerabilities due to issues with memory management, for example buffer overflows. Type-safety is a stronger property than memory-safety, which in addition ensures that all variables are handled as the correct type of information, for example not trying to store a 64-bit integer in a 32-bit register, and verifies the absence of undefined behaviour as a result of arithmetic defects, such as overflow of signed integers and division-by-zero.

Both memory-safety and type-safety can be carried out either statically at compile time, or dynamically at runtime. Static verification prevents unsafe code from even compiling in the first place, whereas dynamic checking will prevent unsafe behaviour as and when it happens while the program is running. This can cause issues with the performance of an implementation, for example by causing undesirable exceptions that need handling or aborting the run altogether.

When writing implementations of cryptographic algorithms, using memory-safe languages can mitigate common software vulnerabilities relating to memory management. Different languages offer different memory-safety guarantees, for example Javascript is dynamically memory-safe whereas Rust offers a hybrid approach where almost all memory-safety issues are checked statically apart from Buffer Overflow, which is subject to a dynamic check and a "panic" at run-time on failure.

Static memory-safety and type-safety can also be achieved by using function contracts, which specify a set of preconditions and postconditions that the code needs to satisfy and then proves that these are always met. This approach can be applied to languages that are not typically memory-safe (e.g. C), however it is fragile to small changes in the implementation.

EXAMPLE: Heartbleed (CVE-2014-0160) is one of the most famous vulnerabilities that affected OpenSSL's implementation of TLS. The client and server can exchange identical TLS heartbeat extensions which include a length field and a payload of that length. The OpenSSL implementation allocated the memory buffer based on the value in the length field, but did not check that this length matched the length of the corresponding payload. This meant an attacker could send a heartbeat with a large length value but a small payload, and trick the victim into sending them up to 64 kB of memory. Using a memory-safe language would have prevented this attack.

8.4.2 Benefits and Limitations

Writing memory-safe or type-safe cryptographic code provides a good way to eliminate vulnerabilities that are easy to introduce and difficult to find through code review or testing.

However, memory-safe languages can come with significant impact on the efficiency of the implementation. This may be at compilation time in which case it may be a worthwhile trade-off for the additional assurances. On the other hand, the performance overheads of using some memory-safe languages at runtime are likely to be undesirable for many implementations, especially those designed for high throughput applications.

Using a memory-safe language for cryptography requires compatibility with the rest of the codebase and may require the developers to upskill in a new programming language. Furthermore, there are no programming languages that can completely eliminate memory management vulnerabilities without the use of additional tools to provide assurance.

EXAMPLE: In Rust there is an unsafe keyword, and if used, there are no guarantees about memory-safety. This keyword exists to allow the developers additional control over the code that compromises some of the memory-safety guarantees. It is often used to integrate libraries into Rust code that are written in languages that are not considered memory-safe.

History

Version	Date	Status
V1.1.1	March 2026	Publication
V1.2.1	May 2026	Publication