Guidelines for Self-Assessment of Cryptographic Algorithms Performance on x86 Architecture

Institute of Commercial Cryptography Standards

June 2026

Contents

1 Notes on Algorithm Implementation

1.1 Purpose

These Guidelines are intended to standardize the self-assessment process for implementations of the next-generation public-key cryptographic algorithms performance, including key encapsulation mechanisms (KEMs), digital signature schemes, and key exchange protocols, as well as cryptographic hash algorithms, on the x86 architecture. They define the self-assessment metrics and requirements for the correctness, comparability, and efficiency of algorithm implementations.

Based on the characteristics of the x86 architecture, these Guidelines provide algorithm submitters with a technical reference and procedural guidance for preparing the Reference Implementation Version and Optimization Version on the x86 architecture. They do not, by itself, impose any additional implementation requirements. Algorithm submitters may, in accordance with the algorithm submission requirements, submit corresponding reference implementation versions and optimized implementation versions (including performance-optimized and resource-optimized versions).

The self-assessment serve as checkable technical evidence in the algorithm submission materials and are used solely to describe the implementation environment, testing process, and test results.

1.2 Scope

These Guidelines apply to the self-assessment of two categories of cryptographic algorithms performance on x86 architecture: public-key cryptographic algorithms and cryptographic hash algorithms. They cover the Reference Implementation Version and the optimization versions, namely the Performance Optimization Version and the Resource Optimization Version. The versions are described as follows:

(1) Reference Implementation Version: This version focuses on the correctness of algorithmic logic, implementation simplicity, and ease of development. It does not require additional designs of performance or resource optimization, nor does it introduce specific optimization techniques, thereby facilitating understanding of the implementation principles of the algorithm.

(2) Performance Optimization Version: This version is primarily intended to maximize the performance of the algorithm implementation. Optimization techniques, such as those based on x86 instruction set extensions, may be used to reduce latency and improve throughput. A moderate increase in storage resource consumption is permitted to support high-performance and high-throughput software application scenarios.

(3) Resource Optimization Version: This version is primarily intended to minimize storage resource consumption in the algorithm implementation. Techniques such as streamlining data structures, reusing storage resources, and simplifying computational processes may be used to reduce storage resource consumption and support resource-constrained software application scenarios.

1.3 Assessment Guidelines

The guidelines for self-assessment of cryptographic algorithms performance on x86 architecture are as follows:

(1) Unified Requirements: The self-assessment of public-key cryptographic algorithms and cryptographic hash algorithms should be conducted based on the implementation self-assessment metrics provided in Chapter 3.

(2) Categorized Submission: Submission materials should be submitted separately by algorithm category and implementation version, and the differences among different versions should be reflected. For public-key cryptographic algorithms, the three functions (namely key encapsulation mechanisms (KEMs), digital signature schemes, and key exchange protocols) should be separately assessed.

(3) Full Metric Coverage: The assessment results for every implementation version should cover metrics including functionality, performance, resource consumption, and transmission and storage overhead.

(4) Authenticity and Reproducibility: The execution results in the implementation self-assessment test report should be authentic and reproducible.

1.4 Material Preparation

(1) Algorithm submitters should organize the materials separately by algorithm category and implementation version, and submit them as a complete package for each version. Each version should include at least the following materials:

No.	Material Name	Supplementary Notes
1	Algorithm Source Code	Complete source code of the algorithm implementation, including build configuration files, implementation documentation, and other relevant materials.
2	Self-Assessment Code and Data	Code and data that invoke the algorithm implementation source code and are used to perform self-assessment operations for functionality, performance, resource consumption, and transmission and storage overhead.
3	Self-Assessment Report	Test results.
4	Dependency Information	List of third-party library dependencies and their version information.

(2) Material naming rule: The material files should be named in the format “Algorithm Category-Algorithm Name-x86-Implementation Version”, such as “Key Encapsulation-SM2-x86-Resource Optimization Version”.

2 Algorithm Implementation Guidelines

2.1 Hardware and Software Environment

To ensure the comparability of self-assessment results, it is recommended to adopt the following baseline environment configurations for implementations on the x86 architecture.

The hardware environment configurations are recommended as follows:

No.	Component	Configuration Description
1	CPU Model	No specific model is required. The CPU should be based on a 64-bit x86 architecture and be compatible with the x86 instruction set.
2	Clock Frequency	Recommended to be more than 2 GHz.
3	x86 Instruction Set	64-bit base instruction set, with support for the AVX2 instruction set.
4	Core Count	Only a single core should be used for testing on the x86 architecture.
5	Memory	Recommended to be more than 4 GB.

The software environment configurations are recommended as follows:

No.	Component	Configuration Description
1	Operating System	Linux operating system. The report should specify the distribution name and version actually used. Kernel version 4.19 or later is recommended.
2	Compiler	GNU GCC 8.5.0 or later.
3	Build Tool	CMake 3.11.4 or later.

2.2 Algorithm Implementation

The guidelines for algorithm implementation on the x86 architecture are as follows:

(1) Interface Specification: The two algorithm categories shall be implemented in accordance with the programming interfaces specified in the *Public-Key Cryptographic Algorithm Submission Requirements* and *Cryptographic Hash Algorithm Submission Requirements*, respectively.

(2) Code Specification: The algorithm implementation code shall be written in ISO/IEC 9899:1999 (C99). The performance optimization version and resource optimization version may use inline assembly or standalone assembly source files targeting the x86 architecture as needed. In addition, the algorithm implementation code shall be clear, well-structured, and fully commented, and shall avoid common vulnerabilities such as superfluous code, infinite loops, and buffer overflows.

3 Self-Assessment Metrics

3.1 Metric Definitions

The self-assessment content is divided into three categories: basic verification items, record information items, and quantitative comparison metrics. Basic verification items include interface conformance, correctness of Known Answer Test (KAT) validation, buildability of the program, reproducibility of test scripts, and completeness of submission materials. Record information items cover third-party dependencies, random number generation methods, compilation parameters, target instruction sets, optimization methods, the actual test environment, and related information. Quantitative comparison metrics include clock cycles, throughput, memory usage, data size, interaction rounds, and related dimensions. Basic verification items and record information items are mainly used to determine whether the assessment results are authentic and valid and to ensure that the data can be reviewed and verified. Quantitative comparison metrics support subsequent analysis of performance, resource consumption, data transmission, and storage overhead.

The implementation self-assessment metrics are defined as follows:

Level-1 Metric	Level-2 Metric	Description
Functional Metric	Correctness	Use Known Answer Test (KAT) vectors to verify whether the algorithm functionality is correctly implemented.
Performance Metrics	Clock Cycles	The number of CPU clock cycles required to complete one core algorithm operation, including key generation, signature generation, signature verification, encapsulation, decapsulation, shared secret derivation, or hash computation, measured in cycles.

Level-1 Metric	Level-2 Metric	Description
	Throughput	For public-key cryptographic algorithms: the number of algorithm operations completed per unit time, including key encapsulation, digital signature, and key exchange operations, measured in ops/s. For cryptographic hash algorithms: the number of data Bytes processed per unit time, measured in MB/s.
Resource Consumption Metrics	Static Memory Usage	Baseline memory usage after the algorithm is loaded but before any operation is executed, including the code segment, constants, global variables, and related static resources, measured in Bytes.
	Peak Memory Usage	The maximum memory usage observed during algorithm execution, including runtime resources such as heap and stack usage, as well as static resources such as the code segment, constants, and global variables, measured in Bytes.
Transmission and Storage Overhead Metrics	Data Size	For public-key cryptographic algorithms: the sizes of public keys, private keys, signatures, ciphertexts, key exchange messages, and related data, measured in Bytes.
	Rounds	For public-key cryptographic algorithms: the number of interaction rounds for key exchange and related operations.

3.2 Algorithm Implementation Versions

These Guidelines provide algorithm submitters with implementation guidance for three versions on the x86 architecture: the Reference Implementation Version, the Performance Optimization Version, and the Resource Optimization Version. In addition, the submission of other versions targeting specific application scenarios and having practical value is encouraged.

3.2.1 Reference Implementation Version

The Reference Implementation Version should be written in pure C and should conform to the ISO C99 standard. The use of assembly language, compiler-specific extensions, and hardware-specific instruction sets, should be avoided.

The Reference Implementation Version should use the following build configuration:

gcc -std=c99 -Wpedantic -Wall -Wextra -O2.

The options are described as follows:

- -std=c99: specifies that the compiler should follow the ISO C99 standard, ensuring syntactic consistency and cross-platform portability.
- -Wpedantic: enforces strict compliance with the ISO C99 standard and prohibits the use of non-standard, compiler-specific extension.
- -Wall: enables the commonly used set of compiler warnings to help identify and fix obvious logic issues in the code.

- -Wextra: enables additional checks beyond the basic warning set to capture more subtle code defects.
- -O2: performs moderate performance optimization to improve execution efficiency without significantly increasing program size.

For any third-party libraries used by the implementation, the same build configuration is recommended.

3.2.2 Performance Optimization Version

The Performance Optimization Version may be implemented in C, inline assembly targeting the x86 architecture, or standalone assembly source files, with the objective of maximizing algorithm performance. The following features of the x86 architecture may be fully utilized:

- 64-bit base instruction set: x86-64.
- SIMD vectorization: AVX2.
- Implementation optimizations: loop unrolling, pipelining, inline functions, the -O3 compilation option, and related techniques.

The Performance Optimization Version should use the following build configuration:

gcc -O3 -march=x86-64 -mavx2 -mtune=native -flto -fomit-frame-pointer -std=c99 -Wpedantic -Wall -Wextra.

The options are described as follows:

- -O3: enables a high level of performance optimization. Based on -O2, it adds automatic vectorization, loop unrolling, and more aggressive function inlining, with the objective of maximizing program execution speed.
- -march=x86-64: specifies x86-64 as the target instruction set architecture.
- -mavx2: enables the AVX2 instruction set.
- -mtune=native: tunes code generation according to the microarchitecture of the build machine, using the hardware characteristics of the target core to improve execution efficiency.
- -flto: enables link-time optimization, allowing the compiler to perform global analysis and optimization across source files, thereby further reducing code size and improving performance.
- -fomit-frame-pointer: omits the frame pointer when it is not needed, thereby freeing a general-purpose register for program logic.

For any third-party libraries used by the implementation, the same build configuration is recommended.

3.2.3 Resource Optimization Version

The Resource Optimization Version may be implemented in C, inline assembly targeting the x86 architecture, or standalone assembly source files, with the objective of reducing storage consumption. To achieve this optimization objective, code may be written using techniques such as streaming processing instead of loading the entire data set at once, timely release of unused memory, data structure optimization, and memory reuse to reduce allocation and deallocation overhead.

The Resource Optimization Version should use the following build configuration:

gcc -Os -march=x86-64 -mavx2 -flto -fomit-frame-pointer -std=c99 -Wpedantic -Wall -Wextra.

The options are described as follows:

- -Os: enables size optimization, with the objective of reducing code size.
- -march=x86-64: specifies x86-64 as the target instruction set architecture.
- -mavx2: enables the AVX2 instruction set.
- -flto: enables link-time optimization, allowing the compiler to perform global analysis and optimization across source files, thereby further reducing code size and improving performance.
- -fomit-frame-pointer: omits the frame pointer when it is not needed, thereby freeing a general-purpose register for program logic.

For any third-party libraries used by the implementation, the same build configuration is recommended.

3.3 Self-Assessment Testing

The implementation self-assessment on the x86 architecture should include the following issues:

(1) Functional Testing: Use Known Answer Test (KAT) vectors as inputs to the algorithm. Compare the actual outputs of the algorithm with the standard known outputs to verify whether the algorithm functionality is correctly implemented.

(2) Performance Testing: Execute multiple times, and calculate performance metrics based on the execution time and the volume of data processed.

(3) Resource Consumption Testing: Execute multiple times, and measure and record static memory usage and peak memory usage.

(4) Transmission and Storage Overhead Testing: Measure and record the data sizes of public keys, private keys, signatures, ciphertexts, key exchange messages, and related data for public-key cryptographic algorithms.

The self-assessment test data should cover all algorithm instances, all declared security strength levels, and all specified programming interfaces.

Note: For scenarios requiring repeated execution and statistical analysis, the recommended number of measurements is no less than 100.

3.4 Self-Assessment Test Data

All test vectors for functional testing should be recorded in the report. Performance and resource consumption tests should use reproducible inputs, and the source or generation method of the input data should be specified in the test report.

For tests requiring random numbers, it is recommended to use a fixed test set or to record the random seed and generation rules so that the results are reproducible. For random number generation rules, the method specified in the programming interface files provided in the algorithm submission requirements is recommended.

3.4.1 Test Data for Public-Key Cryptographic Algorithms

For the key encapsulation function, functional test data for key generation, encapsulation, and decapsulation should be prepared, and the sizes of the public key, private key, ciphertext, and encapsulated key should be recorded.

For the digital signature function, functional test data for key generation, signature generation, and signature verification should be prepared, and the sizes of the public key, private key, and signature should be recorded.

For the key exchange function, functional test data for initialization, message generation, and shared secret

key derivation should be prepared, and the message length, number of interaction rounds, and sizes of the long-term public key, if any, long-term private key, if any, and shared secret key should be recorded.

For public-key cryptographic algorithm test data, performance test results should be provided separately for the three functions of key encapsulation, digital signature, and key exchange under different security strength levels.

Function	Metrics	Remarks
Key Encapsulation	Key generation performance; encapsulation performance; decapsulation performance	Only the performance metrics of this function are tested; the effect of differences in encapsulated key length is not considered.
Digital Signature	Key generation performance; signature generation performance; signature verification performance	A 64-byte message should be used as the message input to be signed for testing, so as to avoid the impact of differences in hash computation overhead on the comparability of test results.
Key Exchange	Key exchange performance	Network transmission time is not counted.

3.4.2 Test Data for Cryptographic Hash Algorithms

For cryptographic hash algorithms, functional test data for hash computation should be prepared. If the algorithm supports extendable output or variable-length output, functional testing should cover different output lengths.

For cryptographic hash algorithms, eight levels of data, S1 to S8, should be selected as inputs for performance testing. The data range from the byte level to the kilobyte level. The specific levels are as follows:

Level	Recommended Data Length
S1	32 Bytes
S2	128 Bytes
S3	512 Bytes
S4	1024 Bytes
S5	4096 Bytes
S6	8192 Bytes
S7	16384 Bytes
S8	65536 Bytes

3.5 Self-Assessment Test Report

After completing the self-assessment testing, the algorithm designers should prepare a self-assessment test report containing the following information:

(1) Basic Algorithm Information: algorithm category, algorithm name, subdivided function, implementation version, parameter set/security level.

(2) Assessment Environment Information: runtime environment, including processor model, clock frequency, memory, operating system distribution and kernel version, compiler version, build tool version,

start time, end time, and related information.

(3) Functional Testing: test results for each test case.

(4) Performance Testing: average execution time and average clock cycles.

(5) Resource Consumption Testing: peak memory, static memory, and related information.

(6) Transmission and Storage Overhead Testing: the sizes of public keys, private keys, signatures, ciphertexts, key exchange messages, and other relevant data for public-key cryptographic algorithms, as well as the number of key exchange rounds.

(7) Raw Evidence Index: a list of the test commands, input data, raw logs, result CSV/JSON files, and calculation scripts corresponding to each summary metric.

Test results should be clear, explicit, unambiguous, and checkable.

4 Miscellaneous

For algorithm submission requirements and programming interfaces, please refer to the website of the Institute of Commercial Cryptography Standards at www.niccs.org.cn. The following platform provides self-assessment code and data, which are available free of charge: https://vm-manager.xyz. The source code directory structure may refer to the above platform. This platform is unrelated to this solicitation activity and has no affiliation with the Institute of Commercial Cryptography Standards. Use of this platform is not mandatory. Please handle the protection of intellectual property rights related to cryptographic algorithms on your own.