Guidelines for Self-Assessment of Cryptographic Algorithms Performance on ASIC

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 ASIC 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. They define the self-assessment metrics and requirements for the correctness, comparability, and efficiency of algorithm implementations.

Based on the characteristics of the ASIC Implementation, these Guidelines provide algorithm submitters with a technical reference and procedural guidance for preparing the Reference Implementation Version and Optimization Version for the ASIC Implementation. 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 results 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 ASIC: public-key cryptographic algorithms and cryptographic hash algorithms, including the reference implementation and optimized implementation versions. The optimized implementation version can be further classified into the resource optimization and performance optimization versions. 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 parallel processing, pipelining, and loop unrolling may be employed to improve operating frequency and data throughput, thereby enabling deployment in high-performance, high-throughput hardware application scenarios.

(3) Resource Optimization Version: This version is primarily intended to minimize resource consumption in the algorithm implementation, while also taking the throughput-to-area ratio into consideration. Optimization techniques such as reducing parallel processing paths, decreasing data-path width, and simplifying computational units may be employed to reduce hardware resource consumption, thereby enabling deployment in hardware application scenarios with limited resources.

1.3 Assessment Guidelines

The guidelines for self-assessment of cryptographic algorithms performance on ASIC 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, and resource consumption.

(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 Hardware Implementation Source Code	Complete RTL source code (Verilog/SystemVerilog) satisfying synthesizable specifications; if memory macros are used, provide the corresponding generation; if third-party IP is used, provide IP configuration and instantiation-related files.
2	Compilation/Synthesis Configuration Files	SDC constraint files, EDA tool Tcl run scripts, process library/macro configuration files, etc.
3	Self-Assessment Code and Data	EDA tool run-log summaries, file-generation instructions, test cases, performance statistics scripts, normalization calculation scripts, etc.
4	Algorithm Implementation Description Document	Core information including ASIC implementation principles, logic architecture, design descriptions for implementation versions, optimization methods, and module boundary definitions; normalization formulas, parameter values, and calculation procedures should be specified.
5	Self-Assessment Report	Complete raw results of functional correctness testing and synthesis/performance/area/power tests; report of normalization calculation results.
6	Dependency Description	List of dependencies, including process libraries, memory macros (SRAM/ROM, etc.), and third-party IP, with clear identifiers (name/version/configuration, etc.); the area of the two-input NAND gate (NAND2) should be indicated for area normalization.

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

The hardware environment configurations are recommended as follows:

No.	Component	Configuration Description
1	Process node	Use a standard cell library dedicated to synthesis assessment. No mandatory process node is specified; submitters may select an appropriate node according to actual requirements, such as 28 nm/40 nm/55 nm.
2	Memory macros	If SRAM/ROM or other memory macros are used, they should match the process library, and the macro model and configuration (depth/width/number of ports) should be specified.
3	PVT conditions	Provide assessment results under typical conditions and clearly indicate voltage, temperature, and process-corner information. Normalization calculations are based on raw metrics under typical conditions.

No.	Component	Configuration Description
4	Normalization core parameters	The NAND2 gate area should use the area of the minimum-size two-input NAND gate in the process library, and the specific value should be clearly indicated in the dependency description (area unit: μm²).

The software environment configurations are recommended as follows:

No.	Component	Configuration Description
1	Logic synthesis tool	Empyrean Technology ZenSyn V2025.2 or Synopsys Design Compiler V2019.03 (used to obtain raw metrics such as area, operating frequency, and power consumption, and to provide baseline data for normalization calculations).

2.2 Algorithm Implementation

The guidelines for algorithm implementation on the ASIC implementation are as follows:

(1) Interface Specification: The top-level module interface should be clearly defined, including the module name and port list (control signals/data signals/clock signals). A SRAM-like interface is recommended for data interfaces.

(2) Synthesizability: RTL source code should satisfy EDA tool synthesizability requirements and use standard Verilog/SystemVerilog syntax. Non-synthesizable statements (such as delay statements and dynamic arrays) should be avoided to ensure that logic synthesis can be completed directly and that raw metrics such as post-synthesis area and frequency can be accurately extracted.

(3) Logic Specification: The implementation logic should be fully consistent with the algorithm standard, without logic deviations. Common hardware design issues such as combinational logic loops, potential timing violations, and redundant logic should be avoided to ensure functional correctness. Redundant logic should also be avoided to prevent interference with raw area and power metrics, ensuring that normalization results truly reflect the core efficiency of the algorithm.

3 Self-Assessment Metrics

3.1 Metric Definitions

The self-assessment content is divided into three major categories: basic verification items, record information items, and quantitative comparison metrics. Basic verification items include RTL source-code compliance, KAT verification correctness, synthesis script reproducibility, and completeness of submitted materials. Record information items cover process libraries, memory macros, third-party IP, normalization core parameters, optimization methods, and test environment. Quantitative comparison metrics include clock cycles, computational throughput, computational latency, area, power consumption, throughput-to-area ratio, normalized metrics, and other dimensions. Basic verification items and record information items are mainly used to determine the authenticity and validity of assessment results and to ensure that data can be re-verified. Quantitative comparison metrics support subsequent analysis of performance, resource consumption, and power consumption.

The implementation self-assessment metric definitions are as follows:

Level-1 Metric	Level-2 Metric	Raw Unit	Normalized Unit	Description (Including Normalization Formula)
Raw performance metrics	Clock Cycles	cycles	-	Number of clock cycles required to complete one core algorithm operation (key generation, signature generation, signature verification, encapsulation, decapsulation, shared secret derivation, or hash computation).
	Operating Frequency	MHz	-	Highest frequency satisfying STA convergence, obtained by frequency sweeping under the same constraint template (recommended step: 1-2 MHz; final WNS close to 0 but non-negative).

Level-1 Metric	Level-2 Metric	Raw Unit	Normalized Unit	Description (Including Normalization Formula)
	Computational Throughput	ops/s	-	Public-key cryptographic algorithms: number of algorithm operations completed per unit time, including key generation, signature generation, signature verification, encapsulation, decapsulation, shared secret derivation, etc.
		Mb/s	-	Cryptographic hash algorithms: number of data bits processed per unit time, including hash-value generation.
	Computational Latency	ns	-	Actual time required to complete one core algorithm operation (such as key generation, signature generation, signature verification, encapsulation, decapsulation, shared secret derivation, or hash computation).
Raw area metrics	Total Synthesized Area	μm²	-	The total logical area of the ASIC synthesized netlist, including core modules and necessary interfaces.
Raw power metrics	Static Power Consumption	mW	-	Static/leakage power consumed under typical PVT conditions at zero workload.
	Dynamic Power Consumption	mW	-	Dynamic power consumed under typical PVT conditions at full workload.
	Total Power Consumption	mW	-	Sum of static power consumption and dynamic power consumption.
Normalized metrics	Normalized Area	-	NAND2-equivalent gates	Ratio of total post-synthesis area (μm²) to NAND2 gate area (μm²).
	Normalized Performance	-	ops/cycle	Public-key cryptographic algorithms: ratio of computational throughput (ops/s) to operating frequency (Hz), i.e., throughput efficiency.
		-	bit/cycle	Cryptographic hash algorithms: ratio of computational throughput (bit/s) to operating frequency (Hz), i.e., throughput efficiency.
	Normalized Power Consumption	-	ops/s/mW	Public-key cryptographic algorithms: ratio of computational throughput (ops/s) to total power (mW), i.e., energy efficiency.
		-	Mb/s/mW	Cryptographic hash algorithms: ratio of computational throughput (Mb/s) to total power (mW), i.e., energy efficiency.
	Normalized Throughput-to-Area Ratio	-	ops/s/NAND2-equivalent gates	Public-key cryptographic algorithms: ratio of computational throughput (ops/s) to NAND2-equivalent gates, i.e., throughput-to-area ratio.
		-	Mb/s/NAND2-equivalent gates	Cryptographic hash algorithms: ratio of computational throughput (Mb/s) to NAND2-equivalent gates, i.e., throughput-to-area ratio.

3.2 Algorithm Implementation Versions

These guidelines provide algorithm submitters with implementation guidance for three versions on the ASIC 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.

Algorithm submitters can follow the algorithm implementation guidelines specified in this guide to complete the ASIC implementation of cryptographic algorithms. Please refer to the appendix for the source code directory structure.

3.2.1 Reference Implementation Version

The reference implementation self-assessment focuses on logical correctness, implementation simplicity, and portability across technologies, without pursuing extreme performance and area optimization.

(1) Implementation Plan: No optimizations such as parallelism or pipelining; use the native logic architecture of the algorithm; no resource trimming; logical modules correspond one-to-one with the theoretical algorithm architecture.

(2) Synthesis Constraints: A basic constraint template is used, without additional performance optimization constraints. Clock uncertainty, maximum transition, maximum capacitance, and other parameters are configured according to the process-library defaults.

(3) Assessment Focus: Verify synthesizability and functional correctness of the algorithm logic.

(4) Source Code Implementation: RTL source code should be concise and readable, with complete comments, clear module partitioning, and no redundant logic, so that the organizer can reproduce raw metrics and normalization results.

3.2.2 Performance Optimization Version

The performance optimization self-assessment focuses on throughput maximization and operating frequency improvement as the core goals and is suitable for high-performance and high-throughput scenarios.

(1) Implementation Plan: High-performance optimization methods such as parallel computation, pipelining, loop unrolling, and reuse of multiple arithmetic units may be adopted to improve algorithm processing speed and support high-frequency, high-throughput operation, with performance maximization prioritized.

(2) Synthesis Constraints: Performance-optimization-related constraints are used, and parameters such as clock uncertainty, maximum transition, and maximum capacitance are strictly controlled. Timing convergence at high frequencies should be ensured, and performance-optimization metrics should be prioritized.

(3) Assessment Focus: Focus on collecting raw computational throughput and operating-frequency data, obtaining raw performance metrics for the ASIC implementation, and completing performance normalization calculations.

(4) Source Code Implementation: RTL source code should be concise and readable, with complete comments, clear module partitioning, and no redundant logic, so that the organizer can reproduce raw metrics and normalization results.

(5) Optimization Description: The report should clearly specify the concrete performance-optimization methods and the increase in hardware resources used, and should explain the differences between performance normalization results and those of the reference implementation to demonstrate the practical effect of performance optimization.

3.2.3 Resource Optimization Version

The resource optimization self-assessment focuses on resource usage optimization as the core goal and is suitable for resource optimization scenarios.

(1) Implementation Plan: Hardware resources may be pruned by removing parallel paths, reducing bit widths, simplifying arithmetic units, adopting iterative implementations, and other methods. High-performance optimization methods are not used, and throughput-to-area optimization is achieved on the basis of area optimization.

(2) Synthesis Constraints: On the premise of ensuring timing convergence, constraints related to the throughput-to-area ratio are configured with priority, and resource-optimization metrics are prioritized.

(3) Assessment Focus: Focus on collecting raw data for the total post-synthesis area, obtaining raw area metrics for the ASIC implementation, and completing area normalization calculations.

(4) Source Code Implementation: RTL source code should be concise and readable, with complete comments, clear module partitioning, and no redundant logic, so that the organizer can reproduce raw metrics and normalization results.

(5) Optimization Description: The report should clearly specify the concrete area-optimization methods and the scope of resource trimming, and should explain the differences between area normalization results and those of the reference implementation to demonstrate the practical effect of resource optimization.

3.3 Self-Assessment Testing

The ASIC implementation self-assessment includes the following issues:

(1) Functional Testing: Load test vectors using a simulation tool to verify the functional correctness of the RTL source code.

(2) Logic Synthesis and Report Generation: Use the synthesis EDA tool to synthesize the RTL code and generate raw assessment data for area, performance, and power consumption.

(3) Performance Metric Calculation: Calculate computational throughput based on the operating frequency obtained from logic synthesis and the number of clock cycles obtained from functional simulation.

(4) Normalized Metric Calculation: Complete the calculations of normalized metrics for area, performance, power consumption, and throughput-to-area ratio according to the formulas specified in these Guidelines.

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, at least 100 measurements are recommended.

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: process library identifier (node/name/version), EDA tool version, PVT conditions (voltage/temperature/process corner), SDC constraint template version, and normalization core parameters (NAND2 gate area).

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

(4) Performance Testing: raw performance data (operating frequency, clock cycles, computational latency, computational throughput) and normalized performance metrics.

(5) Area Testing: raw area data (total post-synthesis area of the algorithm) and normalized area metrics.

(6) Power Testing: raw power data (static power, dynamic power, total power) and normalized power metrics.

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

The test results should be clear, explicit, unambiguous, and checkable.

4 Miscellaneous

Algorithm submitters are responsible for securing lawful sources and authorizations for use of process libraries and EDA tools, and for the compliance of submitted materials. The organizer does not provide process library or EDA tool resources. Basic process-library parameters used for normalization calculations, such as NAND2-equivalent gate area, should be extracted from lawfully authorized process libraries, and the basis for parameter extraction should be attached in the report.

Appendix: Source Code Directory Structure

Algorithm submitters should prepare complete RTL source code (Verilog/SystemVerilog), including operation-core modules, necessary input/output interface logic (wrapper), and control logic (start/reset/done). The source code should satisfy synthesizable requirements and contain no non-synthesizable statements.

Algorithm source files should be placed according to a unified directory structure so that EDA tool scripts can automatically identify and compile them. Source code for different implementation versions should be independent and stored separately. The directory structure is as follows:

(1) PKC public-key cryptographic algorithms:

```
├── API_PKC # ASIC implementation directory for public-key cryptographic algorithms (to be replaced by the designer implementation)
│   ├── Implementations # Source code for different implementation forms of public-key cryptographic algorithms
│   │   ├── Basic_Implementation/ # RTL source code of the reference implementation
│   │   │   ├── SIG/ # Digital signature
│   │   │   ├── KEM/ # Key encapsulation
│   │   │   ├── KEX/ # Key exchange
│   │   ├── Lowsize_Implementation/ # RTL source code of the resource-optimized implementation
│   │   │   ├── SIG/ # Digital signature
│   │   │   ├── KEM/ # Key encapsulation
│   │   │   ├── KEX/ # Key exchange
│   │   ├── HighPerformance_Implementation/ # RTL source code of the performance-optimized implementation
│   │   │   ├── SIG/ # Digital signature
│   │   │   ├── KEM/ # Key encapsulation
│   │   │   ├── KEX/ # Key exchange
│   ├── Test_Vector/ # Functional correctness test vectors for public-key cryptographic algorithms
│   │   ├── KAT_SIG.txt # Digital signature test vectors
│   │   ├── KAT_KEM.txt # Key encapsulation test vectors
│   │   ├── KAT_KEX.txt # Key exchange test vectors
│   ├── Constraints/ # SDC constraint files for public-key cryptographic algorithms
│   │   ├── pkc_algorithm.sdc # Unified constraint template; may be fine-tuned for different implementation forms
│   ├── Normalized/ # Files dedicated to normalization calculations
│   │   ├── norm_param.txt # Normalization core-parameter configuration (NAND2 gate area, etc.)
│   │   ├── norm_calc.tcl # Automatic normalization-metric calculation script (optional)
│   ├── Scripts/ # EDA tool run scripts for public-key cryptographic algorithms
│   │   ├── run_dc.tcl # Synthesis script
│
├── build/ # Synthesis/test output directory
│   ├── netlist/ # Exported netlist files (organized by sub-function)
│   ├── report/ # All EDA tool report files
```

```
│   │   ├── dc/ # Synthesis reports (organized by sub-function/implementation version)
│   │   ├── normalized/ # Normalization-metric calculation reports (organized by sub-function/implementation version)
│   ├── log/ # EDA tool run-log summaries
│   │   ├── norm_log/ # Logs from normalization calculation scripts (optional)
│   ├── result/ # assessment result summaries (Excel/Word)
│
├── README.txt # Implementation description, directory structure, reproduction steps, and optimization methods
```

(2) CryptHash cryptographic hash algorithms:

```
├── API_CryptHash # ASIC implementation directory for cryptographic hash algorithms (to be replaced by the designer implementation)
│   ├── Implementations # Source-code directories for different implementation versions (independent of each other)
│   │   ├── Basic_Implementation/ # RTL source code of the reference implementation (.v/.sv)
│   │   ├── LowSize_Implementation/ # RTL source code of the resource-optimized implementation (.v/.sv)
│   │   ├── HighPerformance_Implementation/ # RTL source code of the performance-optimized implementation (.v/.sv)
│   ├── Test_Vector/ # Functional correctness test vectors for cryptographic hash algorithms
│   │   ├── KAT_ShortMsg.txt # Short-message hash test vectors (S1 level)
│   │   ├── KAT_LongMsg.txt # Long-message hash test vectors (S3-S4 levels)
│   │   ├── KAT_Incremental.txt # Incremental hash test vectors
│   ├── Constraints/ # SDC constraint files (unified template, may be fine-tuned)
│   │   ├── hash_algorithm.sdc # General constraint file for cryptographic hash algorithms
│   ├── Normalized/ # Files dedicated to normalization calculations
│   │   ├── norm_param.txt # Normalization core-parameter configuration (NAND2 gate area, etc.)
│   │   ├── norm_calc.tcl # Automatic normalization-metric calculation script (optional)
│   ├── Scripts/ # EDA tool Tcl run scripts (one-click run)
│   │   ├── run_dc.tcl # Logic synthesis script
│
├── build/ # Synthesis/test output directory (generated automatically by scripts)
│   ├── netlist/ # Exported netlist files (organized by implementation version)
│   ├── report/ # EDA tool report files (organized by implementation version)
│   │   ├── dc/ # Synthesis reports
│   │   ├── normalized/ # Normalization-metric calculation reports
│   ├── log/ # EDA tool run-log summaries
│   │   ├── norm_log/ # Logs from normalization calculation scripts (optional)
│   ├── result/ # assessment result summaries (Excel/Word)
│
├── README.txt # Implementation description, directory structure, reproduction steps, and optimization methods
```