ACVP KAS KC SP 800-56 JSON Specification

Russell Hammett
HII Technical Solutions Division
302 Sentinel Drive, Suite #300, Annapolis Junction, MD 20701

July 22, 2021

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

Abstract

This document defines the JSON schema for testing SP800-56 KAS KC implementations with the ACVP specification.

Keywords

The following are keywords to be used by search engines and document catalogues.

ACVP; cryptography

Disclaimer

Any mention of commercial products or reference to commercial organizations is for information only; it does not imply recommendation or endorsement by NIST, nor does it imply that the products mentioned are necessarily the best available for the purpose.

Additional Information

For additional information on NIST's Cybersecurity programs, projects and publications, visit the Computer Security Resource Center. Information on other efforts at NIST and in the Information Technology Laboratory (ITL) is also available.

Feedback

Feedback on this publication is welcome, and can be sent to: code-signing@nist.gov.

1. Introduction

The Automated Crypto Validation Protocol (ACVP) defines a mechanism to automatically verify the cryptographic implementation of a software or hardware crypto module. The ACVP specification defines how a crypto module communicates with an ACVP server, including crypto capabilities negotiation, session management, authentication, vector processing and more. The ACVP specification does not define algorithm specific JSON constructs for performing the crypto validation. A series of ACVP sub-specifications define the constructs for testing individual crypto algorithms. Each sub-specification addresses a specific class of crypto algorithms. This sub-specification defines the JSON constructs for testing SP800-56 KAS KC implementations using ACVP.

2. Supported KAS-KCs

The following key derivation functions **MAY** be advertised by the ACVP compliant cryptographic module:

- KAS-KC / null / Sp800-56

3. Test Types and Test Coverage

The ACVP server performs a set of tests on the KAS protocol in order to assess the correctness and robustness of the implementation. A typical ACVP validation session **SHALL** require multiple tests to be performed for every supported permutation of KAS capabilities. This section describes the design of the tests used to validate implementations of KAS algorithms.

3.1. Test Types

There are two test types for KAS testing:

- "AFT"—Algorithm Function Test. In the AFT test mode, the IUT **SHALL** act as a party in the Key Confirmation with the ACVP server. The server **SHALL** generate and provide all necessary information for the IUT to perform a successful key confirmation; both the server and IUT **MAY** act as party U/V, as well as recipient/provider to key confirmation.

3.2. Test Coverage

The tests described in this document have the intention of ensuring an implementation is conformant to [SP 800-56A Rev. 3] and [SP 800-56B Rev. 2] Key Confirmation.

3.2.1. Requirements Covered

- SP 800-56Ar3 / SP 800-56Br2—5.1 Cryptographic Hash Functions. SHA1, SHA2, and SHA3 hash functions **SHALL** be available for the various pieces of KAS requiring use of a hash function; such as approved MACs and OneStep KDF.
- SP 800-56Ar3 / SP 800-56Br2—5.2 Message Authentication Code (MAC) Algorithms. AES-CMAC, HMAC, and KMAC algorithms **SHALL** be available for testing under Key Confirmation as the specification states.
- SP 800-56Ar3—5.3 Random Number Generation / SP800-56Br2—5.3 Random Bit Generators. Though random values are used, the testing of the construction of those random values **SHALL NOT** be in scope of ACVP testing.
- SP 800-56Ar3 / SP800-56Br2—5.4 Nonces. Though nonces are used, the testing of the construction of those nonces **SHALL NOT** be in scope of ACVP testing.
- SP 800-56Ar3—5.9 KeyConfirmation / SP 800-56Br2—5.6 Key Confirmation. The ACVP server **SHALL** support key confirmation for applicable KAS and KTS schemes.

3.2.2. KAS-FFC Requirements Not Covered

- SP 800-56Ar3 / SP 800-56Br2 Sections that aren't applicable to Key Confirmation **SHALL NOT** be in the scope of testing covered under this document, for this algorithm.

4. Capabilities Registration

ACVP requires crypto modules to register their capabilities. This allows the crypto module to advertise support for specific algorithms, notifying the ACVP server which algorithms need test vectors generated for the validation process. This section describes the constructs for advertising support of KAS KC algorithms to the ACVP server.

The algorithm capabilities **MUST** be advertised as JSON objects within the ‘algorithms’ value of the ACVP registration message. The ‘algorithms’ value is an array, where each array element is an individual JSON object defined in this section. The ‘algorithms’ value is part of the ‘capability_exchange’ element of the ACVP JSON registration message. See the ACVP specification [ACVP] for more details on the registration message.

4.1. Prerequisites

Each algorithm implementation **MAY** rely on other cryptographic primitives. For example, RSA Signature algorithms depend on an underlying hash function. Each of these underlying algorithm primitives must be validated, either separately or as part of the same submission. ACVP provides a mechanism for specifying the required prerequisites:

Prerequisites, if applicable, **MUST** be submitted in the registration as the `prereqVals` JSON property array inside each element of the `algorithms` array. Each element in the `prereqVals` array **MUST** contain the following properties

Table 1 — Prerequisite Properties

JSON Property	Description	JSON Type
algorithm	a prerequisite algorithm	string
valValue	algorithm validation number	string

A “valValue” of “same” **SHALL** be used to indicate that the prerequisite is being met by a different algorithm in the capability exchange in the same registration.

An example description of prerequisites within a single algorithm capability exchange looks like this

```
"prereqVals":
[
  {
    "algorithm": "Alg1",
    "valValue": "Val-1234"
  },
  {
    "algorithm": "Alg2",
    "valValue": "same"
  }
```

]

Figure 1

4.2. Prerequisite Algorithms

Some algorithm implementations rely on other cryptographic primitives. For example, IKEv2 uses an underlying SHA algorithm. Each of these underlying algorithm primitives must be validated, either separately or as part of the same submission. ACVP provides a mechanism for specifying the required prerequisites:

Table 2 — Prerequisite Algorithms JSON Values

JSON Value	Description	JSON type	Valid Values	Optional
algorithm	a prerequisite algorithm	value	CMAC, DRBG, DSA, HMAC, KMAC, SafePrimes, SHA, SP800-108	No
valValue	algorithm validation number	value	actual number or "same"	No
prereqAlgVal	prerequistie algorithm validation	object with algorithm and valValue properties	see above	Yes

KAS has conditional prerequisite algorithms, depending on the capabilities registered:

Table 3 — Prerequisite requirement conditions

Prerequisite Algorithm	Condition
CMAC	CMAC validation **REQUIRED** when IUT is performing KeyConfirmation (KC) or a KDF and utilizing CMAC.
HMAC	HMAC validation **REQUIRED** when IUT is performing KeyConfirmation (KC) or a KDF and utilizing HMAC.
KMAC	KMAC validation **REQUIRED** when IUT is performing KeyConfirmation (KC) or a KDF and utilizing KMAC.

4.3. Algorithm Capabilities JSON Values

Each algorithm capability advertised is a self-contained JSON object using the following values.

Table 4 — Capabilities JSON Values

JSON Value	Description	JSON type	Valid Values	Optional
algorithm	The algorithm under test	string	"KAS-KC"	No
revision	The algorithm testing revision to use.	string	"Sp800-56"	No
prereqVals	Prerequisite algorithm validations	array of prereqAlgVal objects	See Section 4.2	No
kasRole	Roles supported for key agreement	array	initiator and/or responder	No
keyConfirmationMethod	The KeyConfirmation capabilities supported.	object	Section 4.3.1	Yes

4.3.1. Supported KeyConfirmation Method

Table 5 — KAS FFC KeyConfirmation Capabilities JSON Values

JSON Value	Description	JSON type	Valid Values	Optional
macMethods	The MAC methods to use when testing KAS or KTS schemes with key confirmation.	object	Section 4.3.2	No
keyConfirmationDirections	The directions in which key confirmation is supported.	array	unilateral, bilateral	No
keyConfirmationRoles	The roles in which key confirmation is supported.	array	provider, recipient	No

4.3.2. Supported MAC Methods

Note that AT LEAST one mac method must be supplied when making use of Key Confirmation.

Table 6 — MAC Method Options

JSON Value	Description	JSON type	Valid Values	Optional
CMAC	Utilizes CMAC as the MAC algorithm.	object	See Section 4.3.2.1. Note that the keyLen must be 128,	Yes

JSON Value	Description	JSON type	Valid Values	Optional
			192, or 256 for this MAC.	
HMAC-SHA-1	Utilizes HMAC-SHA-1 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA2-224	Utilizes HMAC-SHA2-224 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA2-256	Utilizes HMAC-SHA2-256 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA2-384	Utilizes HMAC-SHA2-384 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA2-512	Utilizes HMAC-SHA2-512 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA2-512/224	Utilizes HMAC-SHA2-512/224 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA2-512/256	Utilizes HMAC-SHA2-512/256 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA3-224	Utilizes HMAC-SHA3-224 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA3-256	Utilizes HMAC-SHA3-256 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA3-384	Utilizes HMAC-SHA3-384 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
HMAC-SHA3-512	Utilizes HMAC-SHA3-512 as the MAC algorithm.	object	See Section 4.3.2.1	Yes
KMAC-128	Utilizes KMAC-128 as the MAC algorithm. Note that a customization string of “KC” is used for the function when KMAC is utilized for Key Confirmation.	object	See Section 4.3.2.1	Yes
KMAC-256	Utilizes KMAC-256 as the MAC algorithm. Note that a customization string of “KC” is used for the function when KMAC is utilized for Key Confirmation.	object	See Section 4.3.2.1	Yes

4.3.2.1. Supported MAC Options

Table 7 — MAC Method Base Options

JSON Value	Description	JSON type	Valid Values	Optional
keyLen	The amount of bits from the DKM to pass into the KeyConfirmation MAC function.	integer	128 — 512. Note that the DKM is **Required** to have at least 8 bits available after subtracting the keyLen specified.	No
macLen	The amount of bits to use as the tag from the MAC function.	integer	64 — 512.	No

4.4. Example Registration

The following is a example JSON object advertising support for KAS FFC.

```
{
  "algorithm": "KAS-KC",
  "revision": "Sp800-56",
  "kasRole": [
    "initiator",
    "responder"
  ],
  "keyConfirmationMethod": {
    "macMethods": {
      "KMAC-128": {
        "keyLen": 128,
        "macLen": 128
      }
    },
    "keyConfirmationDirections": [
      "unilateral",
      "bilateral"
    ],
    "keyConfirmationRoles": [
      "provider",
      "recipient"
    ]
  }
}
```

Figure 2

5. Test Vectors

The ACVP server provides test vectors to the ACVP client, which are then processed and returned to the ACVP server for validation. A typical ACVP validation test session would require multiple test vector sets to be downloaded and processed by the ACVP client. Each test vector set represents an individual algorithm defined during the capability exchange. This section describes the JSON schema for a test vector set used with SP800-56 KAS KC algorithms.

The test vector set JSON schema is a multi-level hierarchy that contains meta data for the entire vector set as well as individual test vectors to be processed by the ACVP client. The following table describes the JSON elements at the top level of the hierarchy.

Table 8 — Top Level Test Vector JSON Elements

JSON Values	Description	JSON Type
acvVersion	Protocol version identifier	string
vsId	Unique numeric vector set identifier	integer
algorithm	Algorithm defined in the capability exchange	string
mode	Mode defined in the capability exchange	string
revision	Protocol test revision selected	string
testGroups	Array of test groups containing test data, see Section 5.1	array

An example of this would look like this

```
{
  "acvVersion": "version",
  "vsId": 1,
  "algorithm": "Alg1",
  "mode": "Mode1",
  "revision": "Revision1.0",
  "testGroups": [ ... ]
}
```

Figure 3

5.1. Test Groups JSON Schema

The testGroups element at the top level in the test vector JSON object is an array of test groups. Test vectors are grouped into similar test cases to reduce the amount of data transmitted in the vector set. For instance, all test vectors that use the same key size would be grouped together. The Test Group JSON object contains meta data that applies to all test vectors within the group. The following table describes the secure hash JSON elements of the Test Group JSON object.

The test group for KAS/KTS FFC is as follows:

Table 9 — Vector Group JSON Object

JSON Value	Description	JSON type	Optional
tgId	Numeric identifier for the test group, unique across the entire vector set.	value	No
testType	The type of test for the group (AFT or VAL).	value	No
kasRole	The group role from the perspective of the IUT.	value	No
keyConfirmationDirection	The key confirmation direction.	value	No
keyConfirmationRole	The key confirmation role.	value	No
keyAgreementMacType	The MAC being used for key confirmation.	value	No
keyLen	The length of the key to be used as the macKey.	value	No
macLen	The length of the MAC to be produced.	value	No
tests	The tests for the group.	Array of objects, See Section 5.2.	No

5.2. Test Case JSON Schema

Each test group contains an array of one or more test cases. Each test case is a JSON object that represents a single test vector to be processed by the ACVP client. The following table describes the JSON elements for each test vector.

Table 10 — Test Case JSON Object

JSON Value	Description	JSON type	Optional
tcId	Numeric identifier for the test case, unique across the entire vector set.	value	No
macDataServer	The partyId and ephemeral data to be used from the ACVP server perspective.	value	No
macDataIut	The partyId and ephemeral data to be used from the IUT perspective.	value	No
macKey	The macKey portion of the key confirmation.	value	No

JSON Value	Description	JSON type	Optional
tag	The tag generated as a part of key confirmation (from the IUT perspective).	value	Yes

5.3. Example Test Vectors JSON Object

The following is a example JSON object for test vectors sent from the ACVP server to the crypto module.


```
{
  "vsId": 0,
  "algorithm": "KAS-KC",
  "revision": "Sp800-56",
  "isSample": true,
  "testGroups": [
    {
      "tgId": 1,
      "testType": "AFT",
      "kasRole": "initiator",
      "keyConfirmationDirection": "bilateral",
      "keyConfirmationRole": "provider",
      "keyAgreementMacType": "CMAC",
      "keyLen": 256,
      "macLen": 64,
      "tests": [
        {
          "tcId": 1,
          "macDataServer": {
            "partyId": "3590EA2B8D8EE994684A0CE4385DD2D2",
            "ephemeralData":
 "3139B09E09434C5F294F20115C7EE97B5716C9188CA39D08807F3809ADD8AD05"
          },
          "macDataIut": {
            "partyId": "910C6FE518C33A22380BCD33EAA34A79",
            "ephemeralData":
 "AA380D7E3E49563B006DE8F224336B421137D3CB50BD69472FDD5299885F9637"
          },
          "macKey":
 "08E276F4BC4EAE5DE47C4DB92402E7338D2373CA4BE9A4B43338635E25C5C212"
        },
        {
          "tcId": 2,
          "macDataServer": {
            "partyId": "C19FE731C14EBB0EDE8ECF2C60086CEA"
          },
```

```
          "macDataIut": {
            "partyId": "88E6C06D57E5EAC600DDE7246AAF7408"
          },
          "macKey":
"234ADECE1B99695BD1E539BED042ABC51C9B0D348ECBCF9C0E46F7B885857D71"
        },
        {
          "tcId": 3,
          "macDataServer": {
            "partyId": "5345535892D86B3BE9C57D57E6EB4EA6"
          },
          "macDataIut": {
            "partyId": "022376FC5CBDE150D754BE6C78D2C653"
          },
          "macKey":
"6A9BFC7FC2E6013CE901D59C1DF7297B61FB6B945FF1D7C55217FA5FB54FC5BB"
        },
        {
          "tcId": 4,
          "macDataServer": {
            "partyId": "F30A8967854FED4C423ABBCAC2190D65"
          },
          "macDataIut": {
            "partyId": "B1B0408807E22EB93EFEF2FAFB418EEB",
            "ephemeralData":
"242FD779A30DAEFE542F6832348640A2A8FC824990CFC5E5F1DA881237C7452D"
          },
          "macKey":
"950E78377B63387216C45BBF8349C4DD536B03B26BF6E4D03E855379E9FA5B79"
        }
      ]
    }
  ]
}
```


Figure 4

6. Test Vector Responses

After the ACVP client downloads and processes a vector set, it **MUST** send the response vectors back to the ACVP server. The following table describes the JSON object that represents a vector set response.

Table 11 — Vector Set Response JSON Object

JSON Value	Description	JSON type	Optional
acvVersion	Protocol version identifier	value	No
vsId	Unique numeric identifier for the vector set	value	No
testGroups	Array of JSON objects that represent each test vector group. See Table 12.	array	No

The testGroups section is used to organize the ACVP client response in a similar manner to how it receives vectors. Several algorithms **SHALL** require the client to send back group level properties in their response. This structure helps accommodate that.

Table 12 — Vector Set Group Response JSON Object

JSON Value	Description	JSON type	Optional
tgId	The test group Id	value	No
tests	Array of JSON objects that represent each test vector group. See Table 13.	array	No

The testCase section is used to organize the ACVP client response in a similar manner to how it receives vectors. Several algorithms **SHALL** require the client to send back group level properties in their response. This structure helps accommodate that.

Table 13 — Vector Set Test Case Response JSON Object

JSON Value	Description	JSON type	Optional
tcId	The test case Id	value	No
tag	The tag generated as a part of key confirmation (from the IUT perspective).	value	No

6.1. Example Test Results JSON Object

The following is a example JSON object for test results sent from the crypto module to the ACVP server.


```
{
  "vsId": 0,
  "algorithm": "KAS-KC",
```

```
  "revision": "Sp800-56",
  "isSample": true,
  "testGroups": [
    {
      "tgId": 1,
      "tests": [
        {
          "tcId": 1,
          "tag": "35FA16A8F7CE4DD6"
        },
        {
          "tcId": 2,
          "tag": "7FD1AF7F1FF82F6C"
        },
        {
          "tcId": 3,
          "tag": "A1ABD89925631AC1"
        },
        {
          "tcId": 4,
          "tag": "BAABCDE5BFA9F3FA"
        }
      ]
    }
  ]
}
```

Figure 5

7. Security Considerations

There are no additional security considerations outside of those outlined in the ACVP document.

Appendix A — Terminology

For the purposes of this document, the following terms and definitions apply.

A.1.

Prompt
JSON sent from the server to the client describing the tests the client performs
Registration
The initial request from the client to the server describing the capabilities of one or several algorithm, mode and revision combinations
Response
JSON sent from the client to the server in response to the prompt
Test Case
An individual unit of work within a prompt or response
Test Group
A collection of test cases that share similar properties within a prompt or response
Test Vector Set
A collection of test groups under a specific algorithm, mode, and revision
Validation
JSON sent from the server to the client that specifies the correctness of the response

Appendix B — Abbreviations and Acronyms

ACVP	Automated Crypto Validation Protocol
JSON	Javascript Object Notation

Appendix C — Revision History

Table C-1

Version	Release Date	Updates
1	2021-07-22	Initial Release

Appendix D — References

S. Bradner (March 1997) *Key words for use in RFCs to Indicate Requirement Levels* (Internet Engineering Task Force), BCP 14, March 1997. RFC 2119. RFC RFC2119. DOI 10.17487/RFC2119. https://www.rfc-editor.org/info/rfc2119.

T. Kivinen, M. Kojo (May 2003) *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)* (Internet Engineering Task Force), RFC 3526, May 2003. RFC 3526. RFC RFC3526. DOI 10.17487/RFC3526. https://www.rfc-editor.org/info/rfc3526.

D. Gillmor (August 2016) *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)* (Internet Engineering Task Force), RFC 7919, August 2016. RFC 7919. RFC RFC7919. DOI 10.17487/RFC7919. https://www.rfc-editor.org/info/rfc7919.

P. Hoffman (December 2016) *The "xml2rfc" Version 3 Vocabulary* (Internet Engineering Task Force), RFC 7991, December 2016. RFC 7991. RFC RFC7991. DOI 10.17487/RFC7991. https://www.rfc-editor.org/info/rfc7991.

B. Leiba (May 2017) *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words* (Internet Engineering Task Force), BCP 14, May 2017. RFC 8174. RFC RFC8174. DOI 10.17487/RFC8174. https://www.rfc-editor.org/info/rfc8174.

Elaine B. Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, Richard Davis (April 2018) *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography* (Gaithersburg, MD), April 2018. SP 800-56A Rev. 3. https://doi.org/10.6028/NIST.SP.800-56Ar3.

Elaine B. Barker, Lily Chen, Allen Roginsky, Apostol Vassilev, Richard Davis, Scott Simon (March 2019) *Recommendation for Pair-Wise Key-Establishment Using Integer Factorization Cryptography* (Gaithersburg, MD), March 2019. SP 800-56B Rev. 2. https://doi.org/10.6028/NIST.SP.800-56Br2.

Fussell B, Vassilev A, Booth H, Celi C, Hammett R (July 01, 2019) *Automatic Cryptographic Validation Protocol* (National Institute of Standards and Technology, Gaithersburg, MD), July 01, 2019.