ACVP SHA3 and SHAKE JSON Specification

Christopher Celi
Information Technology Laboratory
Computer Security Division

November 01, 2018

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

Abstract

This document defines the JSON schema for testing SHA3 and SHAKE implementations with the ACVP specification.

Keywords

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

ACVP; cryptography

Foreword

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Audience

This document is intended for the users and developers of ACVP.

Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 of [RFC 2119] and [RFC 8174] when, and only when, they appear in all capitals, as shown here.

Acknowledgements

This document is produced by the Security Testing, Validation and Measurement group under the Automated Cryptographic Validation Testing (ACVT) program.

Executive Summary

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 SHA3 and SHAKE implementations using ACVP.

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 SHA3 and SHAKE implementations using ACVP.

2. Supported Hash Algorithms

The following SHA3-based hash algorithms **MAY** be advertised by this ACVP compliant crypto module:

- SHA3-224 / null / 1.0
- SHA3-256 / null / 1.0
- SHA3-384 / null / 1.0
- SHA3-512 / null / 1.0
- SHA3-224 / null / 2.0
- SHA3-256 / null / 2.0
- SHA3-384 / null / 2.0
- SHA3-512 / null / 2.0
- SHAKE-128 / null / 1.0
- SHAKE-256 / null / 1.0

Other hash algorithms **MAY** be advertised by the ACVP module elsewhere.

3. Test Types and Test Coverage

This section describes the design of the tests used to validate implementations of SHA3 and SHAKE.

3.1. Test Types

There are four types of tests for these hash functions: functional tests, Monte Carlo tests, Variable Output tests and Large Data tests. Each has a specific value to be used in the testType field. The testType field definitions are:

- "AFT" — Algorithm Functional Test. These tests can be processed by the client using a normal 'hash' operation. AFTs cause the implementation under test to exercise normal operations on a single block, multiple blocks, or partial blocks. In all cases, random data is used. The functional tests are designed to verify that the logical components of the hash function or extenable output function (block chunking, block padding etc.) are operating correctly.
- "MCT" — Monte Carlo Test. These tests exercise the implementation under test under stenuous circumstances. The implementation under test must process the test vectors according to the correct algorithm and mode in this document. MCTs can help detect potential memory leaks over time, and problems in allocation of resources, addressing variables, error handling, and generally improper behavior in response to random inputs. Each MCT processes 100 pseudorandom tests. Each algorithm and mode SHOULD have at least one MCT group. See Section 3.2 for implementation details.
- "VOT" — Variable Output Test. These tests are for SHAKE only, and ensure that an IUT can properly perform the extendable output function to produce digests of specifc lengths. These tests differ from the AFTs for SHAKE in that the AFTs all produce a single digest size, matching the security strength of the extendable output function. The VOTs SHALL produce varying digest sizes based on the capabilities of the IUT.
- "LDT" — Large Data Test. These tests are for SHA3 only. This test performs the hash function on a message that is multiple gigabytes in length. This pushes the bounds of 32-bit data types to ensure an implementation can handle all types of data. See [LDT] for more information motivating the LDT. As a multiple gigabyte message cannot be communicated naturally via ACVP, a specific structure is outlined in Section 3.3.

3.2. Monte Carlo tests for SHA3 and SHAKE

3.2.1. SHA3 Monte Carlo Test

The MCTs start with an initial condition (SEED which is a single message) and perform a series of chained computations. The algorithm is shown below.

```
For j = 0 to 99
MD[0] = SEED;
    For i = 1 to 1000
```

```
        MSG[i] = MD[i-1]
        MD[i] = SHA3(MSG[i])
    Output MD[1000]
    SEED = MD[1000]
```

Figure 1 — SHA-3 Monte Carlo Test

3.2.2. SHAKE Monte Carlo Test

The MCTs start with an initial condition (SEED which is a single message) and perform a series of chained computations. Some values used in the algorithm are based on properties provided during the registration. They are as follows.

- minOutBytes = smallest number of bytes supported
- maxOutBytes = largest number of bytes supported

The SHAKE function used in the pseudocode takes in a bitstring and a desired output length in bits. The M[i] input to SHAKE MUST always contain at least 128 bits. If this is not the case as the previous digest was too short, append empty bits to the rightmost side of the digest. The MCT algorithm is shown below.

```
Range = maxOutBytes - minOutBytes + 1
OutputLen = maxOutBytes
For j = 0 to 99
    MD[0] = SEED
    For i = 1 to 1000
        MSG[i] = 128 leftmost bits of MD[i-1]
        if (MSG[i] < 128 bits)
            Append 0 bits on rightmost side of MSG[i] til MSG[i] is 128 bits
        MD[i] = SHAKE(M[i], OutputLen * 8)

        if (i != 1000)
            RightmostOutputBits = 16 rightmost bits of MD[i] as an integer
            OutputLen = minOutBytes + (RightmostOutputBits % Range)

    Output MD[1000], OutputLen
    SEED = MD[1000]
```

Figure 2 — SHAKE Monte Carlo Test

3.3. Large Data tests for SHA-3

The large data tests are intended to test the ability of a module to hash multiple gigabytes of data at once. This much information cannot be communicated via the JSON files as a normal message property. Instead a new type is defined as a large data type. It is an object that contains a small

content hex string, a content length in bits, a full length in bits and an expansion technique string. The following is an example of this structure.

```
"largeMsg": {
    "content": "DE26",
    "contentLength": 16,
    "fullLength": 42949672960,
    "expansionTechnique": "repeating"
}
```

Figure 3

The ‘contentLength’ property describes the number of bits in the ‘content’ property. The ‘content’ property is the hex string that can be expanded to the full large message. The ‘expansionTechnique’ describes the process used to obtain the full large message. The ‘fullLength’ is the final length of the full large message.

There may be multiple ‘expansionTechnique’ types defined. Here are the types defined for SHA-3 testing.

- “repeating”—Append the number of content bits specified in ‘contentLength’ to itself as many times as needed until a hex string of exactly ‘fullLength’ bits is acquired. In the example shown, the final large message would have the form “DE26DE26DE26…DE26”.

There are multiple ways hash functions can be implemented in an IUT. The most common are via a single Hash() call on the message or via a series of Init(), any number of Update(), Final() calls. As noted in [LDT], the difference between these hashing techniques can have consequences in the cryptographic module. If both interfaces are offered and accessible for testing, the IUT **MUST** only utilize a single Update() call for the large message.

3.4. Test Coverage

The tests described in this document have the intention of ensuring an implementation is conformant to [FIPS 202].

3.4.1. SHA3 Requirements Covered

Sections 3 through 6 in [FIPS 202] outline the Keccak construction and strucures needed to form a valid SHA3 implementation. Normal AFTs test these operations by running the algorithm.

Section 7 in [FIPS 202] states that any input sizes may be used within a SHA3 hash implementation. The input sizes tested in this document range from 0 bits to 65536 bits. In addition a large data test is available by special request which tests messages that are multiple gigabytes in size.

3.4.2. SHA3 Requirements Not Covered

It is noted that [FIPS 202] states that "different procedures that produce the correct output for every input are permitted" and thus the internal states discussed in Sections 3 through 6 **SHALL NOT** be tested or tracked.

3.4.3. SHAKE Requirements Covered

Sections 3 through 6 in [FIPS 202] outline the Keccak construction and strucures needed to form a valid SHAKE implementation. Normal AFTs test these operations by running the algorithm. VOTs exercise the ability of the implementation to perform the algorithm as well by focusing on the sponge construction.

Section 7 in [FIPS 202] states that any input sizes or output sizes may be used within a SHAKE implementation. The input sizes tested in this document range from 0 bits to 65536 bits. The output sizes tested in this document range from 16 bits to 65536 bits.

3.4.4. SHAKE Requirements Not Covered

Again, the internal states discussed in Sections 3 through 6 **SHALL NOT** be tested or tracked.

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 SHA3 and SHAKE 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 4

4.2. SHA3 and SHAKE Algorithm Capabilities Registration

Each SHA3 and SHAKE algorithm capability advertised **SHALL** be a self-contained JSON object. The following JSON values are used for hash algorithm capabilities:

Table 2 — SHA3 and SHAKE Algorithm Capabilities JSON Values

JSON Value	Description	JSON type
algorithm	The hash algorithm and mode to be validated.	string
revision	The algorithm testing revision to use.	string
inBit	Implementation does accept bit-oriented messages	boolean
inEmpty	Implementation does accept null (zero-length) messages	boolean
outputLen	Output length for SHAKE. The value for the outputLen property must consist either of a single range object or a single literal value. This restriction is made to simplify the implementation of the Monte Carlo Tests (see Section 3.2.2).	domain
outBit	SHAKE can output bit-oriented messages	boolean
messageLength	The message lengths in bits supported by the IUT. The current MCT implementation requires that the digest size and 3x the digest size of the registered hash algorithm must be a valid value within the registered domain.	domain
performLargeDataTest	Determines if the server should include the large data test group defined in Section 3.3. This property is **OPTIONAL**, and shall include the lengths in GiB	integer array

JSON Value	Description	JSON type
	being tested. Valid options are {1, 2, 4, 8}.	
NOTE – The lengths provided in the 'performLargeDataTest' property are in gibibytes. 1GiB is equivalent to 2^30 bytes.		

The following grid outlines which properties are **REQUIRED**, as well as all the possible values a server **MAY** support for SHA3 and SHAKE algorithms:

Table 3 — SHA3 and SHAKE Capabilities Applicability Grid

algorithm	revision	inBit	inEmpty	outputLen	outBit	messageLength	performLargeDataTest
SHA3-224	1.0	true, false	true, false				[1, 2, 4, 8]
SHA3-256	1.0	true, false	true, false				[1, 2, 4, 8]
SHA3-384	1.0	true, false	true, false				[1, 2, 4, 8]
SHA3-512	1.0	true, false	true, false				[1, 2, 4, 8]
SHA3-224	2.0					{"Min": 0, "Max": 65536, "Inc": any}	[1, 2, 4, 8]
SHA3-256	2.0					{"Min": 0, "Max": 65536, "Inc": any}	[1, 2, 4, 8]
SHA3-384	2.0					{"Min": 0, "Max": 65536, "Inc": any}	[1, 2, 4, 8]
SHA3-512	2.0					{"Min": 0, "Max": 65536, "Inc": any}	[1, 2, 4, 8]
SHAKE-128	1.0	true, false	true, false	{"Min": 16, "Max": 65536, "Inc": any}	true, false		
SHAKE-256	1.0	true, false	true, false	{"Min": 16, "Max": 65536, "Inc": any}	true, false		

The following is a example JSON object advertising support for SHA3-256 for testing revision 1.0.

```
{
```

```
    "algorithm": "SHA3-256",
    "revision": "1.0",
    "mode": null,
    "inBit": true,
    "inEmpty": true,
    "performLargeDataTest": [1, 2, 4, 8]
}
```

Figure 5

The following is an example JSON object advertising support for SHAKE-128.

```
{
    "algorithm": "SHAKE-128",
    "revision": "1.0",
    "mode": null,
    "inBit": true,
    "inEmpty": true,
    "outBit": true,
    "outputLen": [
        {
        "min": 16,
        "max": 1024
        }
    ]
}
```

Figure 6

The following is a example JSON object advertising support for SHA3-256 for testing revision 2.0.

```
{
    "algorithm": "SHA3-256",
    "revision": "2.0",
    "mode": null,
    "messageLength": [{"min": 0, "max": 65536, "increment": 1}],
    "performLargeDataTest": [1, 2, 4, 8]
}
```

Figure 7

NOTE 1 – Since the increment is 1 in the above, and the minimum value within the message length is zero, this is effectively an “inBit” and “inEmpty” registration from the 1.0 revision testing. If the implementation supports only byte length messages, you could use an increment of 8.

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 SHA3 and SHAKE 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 4 — 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 8

5.1. Test Groups

Test vector sets **MUST** contain one or many test groups, each sharing similar properties. For instance, all test vectors that use the same key size would be grouped together. The testGroups element at the top level of the test vector JSON object SHALL be the array of test groups. The Test Group JSON object **MUST** contain meta-data that applies to all test cases within the group. The following table describes the JSON elements that **MUST** appear from the server in the Test Group JSON object:

Table 5 — Test Group JSON Object

JSON Value	Description	JSON type
tgId	Numeric identifier for the test group, unique across the entire vector set.	integer
testType	Test category type. AFT, MCT or VOT as defined in Section 3	value
tests	Array of individual test case JSON objects, which are defined in Section 5.2	array of testCase objects

5.2. Test Case JSON Schema

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

Table 6 — Test Case JSON Object

JSON Value	Description	JSON type
tcId	Numeric identifier for the test case, unique across the entire vector set.	integer
len	Length of the message or seed	integer
outLen	Length of the digest	integer
msg	Value of the message or seed. Messages are represented as little-endian hex for all SHA3 variations.	string (hex)
largeMsg	Object describing the message for an LDT group	large data object, see Section 3.3 for more information

NOTE 1 – All properties listed in the above table are **REQUIRED** except for outLen which is only **REQUIRED** when the algorithm is SHAKE-128 or SHAKE-256.

NOTE 2 – The maximum value for SHAKE-128 for 'len' is 65,904 bits when the inBit parameter is set to true in the capabilities registration and 66,240 bits when it is set to false. The maximum value for SHAKE-256 for 'len' is 66,428 bits when the inBit parameter is set to true in the capabilities registration and 65,752 bits when it is set to false.

The following are example JSON objects for secure hash test vectors sent from the ACVP server to the crypto module. Notice that the single bit message is represented as "01". This complies with the little-endian nature of SHA3. All hex displayed is little-endian bit order when associated with SHA3 or any of its variations.

```
[
    { "acvVersion": <acvp-version> },
```

```
{
    "vsId": 1564,
    "algorithm": "SHA3-512",
    "revision": "1.0",
    "testGroups": [
    {
        "tgId": 1,
        "testType": "AFT",
        "tests": [
        {
            "tcId": 1,
            "len": 0,
            "msg": "00"
        },
        {
            "tcId": 2,
            "len": 1,
            "msg": "01"
        }]
    },
    {
        "tgId": 2,
        "testType": "MCT",
        "tests": [
        {
            "tcId": 3,
            "len": 512,
            "msg": "30850bd984b14ff5aff18993329...",
        }]
    },
    {
        "tgId": 3,
        "testType": "LDT",
        "tests": [
            {
                "tcId": 4,
                "largeMsg": {
                    "content": "DE26",
                    "contentLength": 16,
                    "fullLength": 42949672960,
                    "expansionTechnique": "repeating"
                }
            }
        ]
    }]
}
```

```
]

[
    { "acvVersion": <acvp-version> },
    {
        "vsId": 1565,
        "algorithm": "SHA3-512",
        "revision": "2.0",
        "testGroups": [
        {
            "tgId": 1,
            "testType": "AFT",
            "tests": [
            {
                "tcId": 1,
                "len": 0,
                "msg": "00"
            },
            {
                "tcId": 2,
                "len": 1,
                "msg": "01"
            }]
        },
        {
            "tgId": 2,
            "testType": "MCT",
            "tests": [
            {
                "tcId": 3,
                "len": 512,
                "msg": "30850bd984b14ff5aff18993329...",
            }]
        },
        {
            "tgId": 3,
            "testType": "LDT",
            "tests": [
                {
                    "tcId": 4,
                    "largeMsg": {
                        "content": "DE26",
                        "contentLength": 16,
                        "fullLength": 42949672960,
                        "expansionTechnique": "repeating"
                    }
```

```
                }
            ]
        }]
    }
]
```

Figure 9

The following is an example JSON object for SHAKE.

```
[
    { "acvVersion": <acvp-version> },
    {
        "vsId": 1564,
        "algorithm": "SHAKE-128",
        "revision": "1.0",
        "testGroups": [
        {
            "tgId": 1,
            "testType": "AFT",
            "tests": [
            {
                "tcId": 1,
                "len": 0,
                "msg": "00"
            },
            {
                "tcId": 2,
                "len": 1,
                "msg": "01"
            }]
        },
        {
            "tgId": 2,
            "testType": "MCT",
            "inBit": true,
            "outBit": true,
            "inEmpty": false,
            "maxOutLen": 4096,
            "minOutLen": 128,
            "tests": [
            {
                "tcId": 3,
                "len": 512,
                "msg": "30850bd984b14ff5aff18993329...",
            }]
        },
```

```
        {
            "tgId": 3,
            "testType": "VOT",
            "tests": [
            {
                "tcId": 4,
                "len": 128,
                "msg": "7a4c48eb710299e4ff2be3f446327a6f",
                "outLen": 16
            },
            {
                "tcId": 5,
                "len": 128,
                "msg": "b16f331b3a0cf4507124b4358f9d15f5",
                "outLen": 144
            }]
        }]
    }
]
```


Figure 10

5.3. Test Vector Responses

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

Table 7 — Vector Set Response JSON Object

JSON Value	Description	JSON Type
acvVersion	Protocol version identifier	string
vsId	Unique numeric identifier for the vector set	integer
testGroups	Array of JSON objects that represent each test vector result, which uses the same JSON schema as defined in Section 5.2	array of testGroup objects

The testGroup Response 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 its response. This structure helps accommodate that.

Table 8 — Vector Set Group Response JSON Object

JSON Value	Description	JSON Type
tgId	The test group identifier	integer

JSON Value	Description	JSON Type
tests	The tests associated to the group specified in tgId	array of testCase objects

Each test case is a JSON object that represents a single test object to be processed by the ACVP client. The following table describes the JSON elements for each test case object.

Table 9 — Test Case Results JSON Object

JSON Value	Description	JSON Type
tcId	Numeric identifier for the test case, unique across the entire vector set.	integer
md	The IUT's digest response to a VOT, AFT or LDT	string (hex)
resultsArray	Array of JSON objects that represent each iteration of a Monte Carlo Test. Each iteration will contain the msg and md (and outLen for SHAKE-128 and SHAKE-256)	array of objects containing the md (and potentially outLen)
NOTE – The 'tcId' **MUST** be included in every test case object sent between the client and the server.		

The following are examples of JSON objects for secure hash test results sent from the crypto module to the ACVP server. The group identified by tgId 1 is a group of AFTs. The group identified by tgId 2 is a group of MCTs. The group identified by tgId 3 is a group of LDTs.

```
{
    "vsId": 0,
    "algorithm": "SHA3-224",
    "revision": "1.0",
    "testGroups": [
        {
            "tgId": 1,
            "tests": [
                {
                    "tcId": 1,
                    "md":
 "D14A028C2A3A2BC9476102BB288234C415A2B01F828EA62AC5B3E42F"
                },
                {
                    "tcId": 2,
                    "md":
 "D14A028C2A3A2BC9476102BB288234C415A2B01F828EA62AC5B3E42F"
                }
```

```
            ]
        },
        {
            "tgId": 2,
            "tests": [
                {
                    "tcId": 1028,
                    "resultsArray": [
                        {
                            "md":
 "E82B1FA3D510C2E423D03CEDF01F66C995CDD573EB63D5A33FDAE640"
                        },
                        {
                            "md":
 "00179AE4CE57DCBD156B981A414370B5089FA8E1098A05443DF3CD62"
                        },
                        {
                            "md":
 "8F6C7F546940352613E8389D4F4B87473A57CACD7E289A27E4F51965"
                        }
                    ]
                }
            ]
        },
        {
            "tgId": 3,
            "tests": [
                {
                    "tcId": 1029,
                    "md":
 "E4F8B44B32F5A25D1F4784601BF095CF5F7C6F4E9EAA1005AD19F09A"
                }
            ]
        }
    ]
}

{
    "vsId": 0,
    "algorithm": "SHA3-224",
    "revision": "2.0",
    "testGroups": [
        {
            "tgId": 1,
            "tests": [
                {
```

```
                    "tcId": 1,
                    "md":
"D14A028C2A3A2BC9476102BB288234C415A2B01F828EA62AC5B3E42F"
                },
                {
                    "tcId": 2,
                    "md":
"D14A028C2A3A2BC9476102BB288234C415A2B01F828EA62AC5B3E42F"
                }
            ]
        },
        {
            "tgId": 2,
            "tests": [
                {
                    "tcId": 1028,
                    "resultsArray": [
                        {
                            "md":
"E82B1FA3D510C2E423D03CEDF01F66C995CDD573EB63D5A33FDAE640"
                        },
                        {
                            "md":
"00179AE4CE57DCBD156B981A414370B5089FA8E1098A05443DF3CD62"
                        },
                        {
                            "md":
"8F6C7F546940352613E8389D4F4B87473A57CACD7E289A27E4F51965"
                        }
                    ]
                }
            ]
        },
        {
            "tgId": 3,
            "tests": [
                {
                    "tcId": 1029,
                    "md":
"E4F8B44B32F5A25D1F4784601BF095CF5F7C6F4E9EAA1005AD19F09A"
                }
            ]
        }
    ]
```

```
}
```

Figure 11

The following is an example JSON object response for SHAKE-128. The group identified by tgId 1 is a group of AFTs. The group identified by tgId 2 is a group of MCTs. The group identified by tgId 3 is a group of VOTs.


```
{
    "vsId": 0,
    "algorithm": "SHAKE-128",
    "revision": "1.0",
    "testGroups": [
        {
            "tgId": 1,
            "tests": [
                {
                    "tcId": 1,
                    "md": "D14A028C2A3A2BC9476102BB288234C4"
                },
                {
                    "tcId": 2,
                    "md": "D14A028C2A3A2BC9476102BB288234C4"
                }
            ]
        },
        {
            "tgId": 2,
            "tests": [
                {
                    "tcId": 1028,
                    "resultsArray": [
                        {
                            "md": "E82B1FA3D510C2E423D03CEDF01F66C9",
                            "outputLen": 128
                        },
                        {
                            "md": "00179AE4CE57DCBD156B981A414370B5",
                            "outputLen": 128
                        },
                        {
                            "md": "8F6C7F546940352613E8389D4F4B8747",
                            "outputLen": 128
                        }
                    ]
                }
            ]
```

```
        },
        {
            "tgId": 3,
            "tests": [
                {
                    "tcId": 1029,
                    "md": "E4F8"
                }
            ]
        }
    ]
}
```


Figure 12

6. 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	2018-11-01	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.

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.

National Institute of Standards and Technology (August 2015) *SHA-3 Standard— Permutation-Based Hash and Extendable-Output Functions* (Gaithersburg, MD), August 2015. FIPS 202. https://doi.org/10.6028/NIST.FIPS.202.

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.

Bassham III L (2014) *The Secure Hash Algorithm Validation System (SHAVS)* (National Institute of Standards and Technology, Gaithersburg, MD), 2014.

Extending NIST's CAVP Testing of Cryptographic Hash Function Implementations. LDT.