ACVP Extendable Output Function (XOF) JSON Specification

Christopher Celi
Information Technology Laboratory
Computer Security Division

January 01, 2019

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

Abstract

This document defines the JSON schema for testing Extendable Output Function 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 Extendable Output Function 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 Extendable Output Function implementations using ACVP.

2. Supported Algorithms

The following XOFs may be advertised by this ACVP compliant crypto module:

- cSHAKE-128
- cSHAKE-256
- parallelHash-128
- parallelHash-256
- tupleHash-128
- tupleHash-256
- KMAC-128
- KMAC-256

Other XOFs may be advertised by the ACVP elsewhere.

3. Test Types and Test Coverage

This section describes the design of the tests used to validate Extendable Output Function implementations.

3.1. Test Types

This section describes the design of the tests used to validate implementations of XOFs. There are three types of tests for these algorithms: Algorithm Functional Tests (AFT), Monte Carlo Tests (MCT) and MAC Verification Tests (MVT). Each has a specific value to be used in the testType field. The testType field definitions are:

- "AFT"—Algorithm Functional Tests. These tests can be processed by the client using a normal 'encrypt' or 'decrypt' operation. AFTs cause the implementation under test to exercise normal operations on a single block, multiple blocks, or (where applicable) partial blocks. In some cases random data is used, in others, static, predetermined tests are provided.
- "MCT"—Monte Carlo Tests. These tests exercise the implementation under test under strenuous 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. Not every algorithm and mode combination has an MCT. See Section 3.2 for implementation details.
- "MVT"—MAC Verification Tests. XXX

3.2. Monte Carlo tests for XOFs

3.2.1. cSHAKE Monte Carlo Test

```
INPUT: The initial Msg is the length of the digest size

MCT(Msg, MaxOutLen, MinOutLen, OutLenIncrement)
{
  Range = (MaxOutLen – MinOutLen + 1);
  OutputLen = MaxOutLen;
  FunctionName = "";
  Customization = "";

  Output[0] = Msg;
  for (j = 0; j < 100; j++)
  {
    for (i = 1; i < 1001; i++)
    {
      InnerMsg = Left(Output[i-1] || ZeroBits(128), 128);
      Output[i] = CSHAKE(InnerMsg, OutputLen, FunctionName, Customization);
      Rightmost_Output_bits = Right(Output[i], 16);
```

```
      OutputLen = MinOutLen + (floor((Rightmost_Output_bits % Range) /
 OutLenIncrement) * OutLenIncrement);
      Customization = BitsToString(InnerMsg || Rightmost_Output_bits);
    }

    OutputJ[j] = Output[1000];
  }

  return OutputJ;
}
```

Figure 1

3.2.2. ParallelHash Monte Carlo Test

```
INPUT: The initial Msg is the length of the digest size

MCT(Msg, MaxOutLen, MinOutLen, OutLenIncrement, MaxBlockSize, MinBlockSize)
{
  Range = (MaxOutLen – MinOutLen + 1);
  OutputLen = MaxOutLen;
  BlockRange = (MaxBlockSize – MinBlockSize + 1);
  BlockSize = MinBlockSize;
  Customization = "";

  Output[0] = Msg;
  for (j = 0; j < 100; j++)
  {
    for (i = 1; i < 1001; i++)
    {
      InnerMsg = Left(Output[i-1] || ZeroBits(128), 128);
      Output[i] = ParallelHash(InnerMsg, OutputLen, BlockSize, FunctionName,
 Customization);
      Rightmost_Output_bits = Right(Output[i], 16);
      OutputLen = MinOutLen + (floor((Rightmost_Output_bits % Range) /
 OutLenIncrement) * OutLenIncrement);
      BlockSize = MinBlockSize + Right(Rightmost_Output_bits, 8) % BlockRange;
      Customization = BitsToString(InnerMsg || Rightmost_Output_bits);
    }

    OutputJ[j] = Output[1000];
  }

  return OutputJ;
}
```

Figure 2

3.2.3. TupleHash Monte Carlo Test

```
INPUT: The initial Single-Tuple of a random length between 0 and 65536 bits.

MCT(Tuple, MaxOutLen, MinOutLen, OutLenIncrement)
{
  Range = (MaxOutLen – MinOutLen + 1);
  OutputLen = MaxOutLen;
  Customization = "";

  T[0][0] = Tuple;

  for (j = 0; j < 100; j++)
  {
    for (i = 1; i < 1001; i++)
    {
      workingBits = Left(T[i-1][0] || ZeroBits(288), 288);
      tupleSize = Left(workingBits, 3) % 4 + 1; // never more than 4 tuples to
 a round
      for (k = 0; k < tupleSize; k++)
      {
        T[i][k] = Substring of workingBits from (k * 288 / tupleSize) to ((k
+1) * 288 / tupleSize - 1);
      }
      Output[i] = TupleHash(T[i], OutputLen, Customization);
      Rightmost_Output_bits = Right(Output[i], 16);
      OutputLen = MinOutLen + (floor((Rightmost_Output_bits % Range) /
 OutLenIncrement) * OutLenIncrement);
      Customization = BitsToString(T[i][0] || Rightmost_Output_bits);
    }

    OutputJ[j] = Output[1000];
  }

  return OutputJ;
}
```

Figure 3

3.2.4. BitsToString Function

```
BitsToString(bits)
{
  string = "";
  foreach byte in bits
  {
      string = string + ASCII((byte % 26) + 65);
  }
```

```
}
```

Figure 4

3.3. Test Coverage

The tests described in this document have the intention of ensuring an implementation is conformant to [SP 800-185].

3.3.1. XOF Requirements Covered

In TBD.

3.3.2. XOF Requirements Not Covered

Some requirements in the outlined specification are not easily tested. Often they are not ideal for black-box testing such as the ACVP. In TBD.

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 XOF 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 5

4.2. XOF Algorithm Capabilities Registration

This section describes the constructs for advertising support of XOFs to the ACVP server. ACVP **REQUIRES** cryptographic modules to register their capabilities in a registration. This allows the cryptographic module to advertise support for specific algorithms, notifying the ACVP server which algorithms need test vectors generated for the validation process.

The XOF capabilities **MUST** be advertised as JSON objects within the 'algorithms' value of the ACVP registration message. The 'algorithms' value **MUST** be an array, where each array element is an individual JSON object defined in this section. The 'algorithms' value **MUST** be part of the 'capability_exchange' element of the ACVP JSON registration message.

Each XOF algorithm capability advertised **SHALL** be a self-contained JSON object.

Each algorithm capability advertised is a self-contained JSON object. The following JSON values are used for XOF algorithm capabilities:

Table 2 — XOF Algorithm Capabilities JSON Values

JSON Value	Description	JSON type
algorithm	The algorithm and mode to be validated.	string
revision	The algorithm testing revision to use.	string
xof	Implementation has the ability to act as an XOF or a non-XOF algorithm	array of boolean
hexCustomization	An optional feature to the implementation. When true, "hex" customization strings are supported, otherwise they aren't. ASCII strings **SHALL** be tested regardless of the value within the `hexCustomization` property.	boolean
msgLen	Input length for the XOF	domain
outputLen	Output length for the XOF	domain
keyLen	Supported key lengths	domain
macLen	Supported MAC lengths	domain
blockSize	blockSize (in bits) to be used with ParallelHash	domain

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

Table 3 — XOF Capabilities Applicability Grid

algorithm	xof	hexCustomization	msgLen	outputLen	keyLen	macLen	blockSize
cSHAKE-128			{Min: 0, Max: 65536, Increment: any}	{Min: 16, Max: 65536, Increment: any}			
cSHAKE-256			{Min: 0, Max: 65536, Increment: any}	{Min: 16, Max: 65536, Increment: any}			
KMAC-128	[true, false]	true, false	{Min: 0, Max: 65536, Increment: any}	{Min: 0, Max: 65536, Increment: any}	{Min: 128, Max: 524288, Increment: 8}	{Min: 32, Max: 65536, Increment: 8}	
KMAC-256	[true, false]	true, false	{Min: 0, Max: 65536, Increment: any}	{Min: 0, Max: 65536, Increment: any}	{Min: 128, Max: 524288, Increment: 8}	{Min: 32, Max: 65536, Increment: 8}	
ParallelHash-128	[true, false]	true, false	{Min: 0, Max: 65536, Increment: any}	{Min: 16, Max: 65536, Increment: any}			{Min: 1, Max: 128, Increment: 1}
ParallelHash-256	[true, false]	true, false	{Min: 0, Max: 65536, Increment: any}	{Min: 16, Max: 65536, Increment: any}			{Min: 1, Max: 128, Increment: 1}
TupleHash-128	[true, false]	true, false	{Min: 0, Max: 65536, Increment: any}	{Min: 16, Max: 65536, Increment: any}			
TupleHash-128	[true, false]	true, false	{Min: 0, Max: 65536,	{Min: 16, Max: 65536,			

algorithm	xof	hexCustomization	msgLen	outputLen	keyLen	macLen	blockSize
			Increment: any}	Increment: any}			
NOTE – For cSHAKE, ParallelHash, and TupleHash, 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 for these algorithms (see Section 3.2).							

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 Extendable Output Function 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 6

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 MVT as defined in Section 3	string
xof	Whether or not the group uses the arbitrary output (XOF) version of the algorithm	boolean
hexCustomization	Whether or not the group uses customization strings in hex (true) or ASCII (false)	boolean
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 for cSHAKE, KMAC and ParallelHash	integer
len	Length of each tuple for TupleHash	array of integer
outLen	Length of the digest	integer
functionName	The function name used in the XOF	string
customization	The ASCII customization string used (between 0 and 161 ASCII characters in length)	string
customizationHex	The hex customization string used (between 0 and 322 hex characters in length)	hex
msg	Value of the message or seed. Messages are represented as	hex

JSON Value	Description	JSON type
	little-endian hex for all SHA3 variations	
keyLen	Length of the key used in KMAC	integer
key	The key used in KMAC	hex
macLen	Length of the MAC	integer
mac	The MAC used in KMAC	hex
blockSize	The blockSize used in ParallelHash	integer
tuple	The tuple of messages used in TupleHash	array of hex

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
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
mac	The IUT's MAC response to an AFT for KMAC	hex
testPassed	The IUT's reponse to an MVT for KMAC	boolean
md	The IUT's digest response to an AFT	hex
outLen	The output length of the digest	integer
resultsArray	Array of JSON objects that represent each iteration of an MCT. Each iteration will contain the md and outLen	array of objects containing the md and outLen
NOTE – The tcId **MUST** be included in every test case object sent between the client and the server.		

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	2019-01-01	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.

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.

John M. Kelsey, Shu-jen H. Chang, Ray A. Perlner (December 2016) *SHA-3 Derived Functions—cSHAKE, KMAC, TupleHash, and ParallelHash* (Gaithersburg, MD), December 2016. SP 800-185. https://doi.org/10.6028/NIST.SP.800-185.

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.

Appendix E — Example Capabilities JSON Objects

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


```
{
  "algorithm": "CSHAKE-128",
  "revision": "1.0",
  "hexCustomization": false,
  "outputLen": [
    {
      "min": 256,
      "max": 4096,
      "increment": 1
    }
  ],
  "msgLen": [
    {
      "min": 0,
      "max": 65536,
      "increment": 1
    }
  ]
}
```


Figure E-1

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


```
{
  "algorithm": "KMAC-128",
  "revision": "1.0",
  "xof": [true, false],
  "hexCustomization": false,
  "msgLen": [
    {
      "min": 0,
      "max": 65536,
      "increment": 1
    }
  ],
  "keyLen": [
    {
      "min": 256,
      "max": 4096,
      "increment": 1
    }
  ],
```

```
  "macLen": [
    {
      "min": 256,
      "max": 4096,
      "increment": 1
    }
  ]
}
```

Figure E-2

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

```
{
  "algorithm": "ParallelHash-128",
  "revision": "1.0",
  "xof": [true, false],
  "hexCustomization": false,
  "blockSize": [
    {
      "min": 1,
      "max": 16,
      "increment": 1
    }
  ],
  "outputLen": [
    {
      "min": 256,
      "max": 4096,
      "increment": 1
    }
  ],
  "msgLen": [
    {
      "min": 0,
      "max": 65536,
      "increment": 1
    }
  ]
}
```

Figure E-3

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

```
{
  "algorithm": "TupleHash-128",
  "revision": "1.0",
```

```
  "xof": [true, false],
  "hexCustomization": false,
  "outputLen": [
    {
      "min": 256,
      "max": 4096,
      "increment": 1
    }
  ],
  "msgLen": [
    {
      "min": 0,
      "max": 65536,
      "increment": 1
    }
  ]
}
```


Figure E-4

Appendix F — Example Test Vectors JSON Objects

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


```
[
{ "acvVersion": <acvp-version> },
{
  "vsId": 0,
  "algorithm": "CSHAKE-128",
  "revision": "1.0",
  "testGroups": [
    {
      "tgId": 1,
      "testType": "AFT",
      "hexCustomization": false,
      "tests": [
        {
          "tcId": 1,
          "msg": "",
          "len": 0,
          "functionName": "",
          "customization": "",
          "outLen": 256
        },
        {
          "tcId": 2,
          "msg": "",
          "len": 0,
          "functionName": "",
          "customization": "[",
          "outLen": 323
        }
      ]
    },
    {
      "tgId": 2,
      "testType": "MCT",
      "hexCustomization": false,
      "tests": [
        {
          "tcId": 251,
          "msg": "5FB4BAE618DABE000B9FDAB178388671",
          "len": 128,
          "functionName": "",
          "customization": ""
```

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


Figure F-1

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


```
[
{ "acvVersion": <acvp-version> },
{
  "vsId": 0,
  "algorithm": "KMAC-128",
  "revision": "1.0",
  "testGroups": [
    {
      "tgId": 1,
      "testType": "AFT",
      "xof": false,
      "hexCustomization": false,
      "tests": [
        {
          "tcId": 1,
          "key":
 "57F9E51E6EE790EA224F33B09184980EC53D4ADC437269BC64CAD4E0BF43FC72",
          "keyLen": 256,
          "msg": "",
          "msgLen": 0,
          "macLen": 256,
          "customization": ""
        },
        {
          "tcId": 2,
          "key":
 "BBEA88A07BD90177E199E488D8725CF926F4702A3703E53CF8E4EF19C10B8A6F80",
          "keyLen": 257,
          "msg": "C0",
          "msgLen": 4,
          "macLen": 264,
          "customization": "i"
        }
      ]
    },
```

```
    {
   "tgId": 3,
      "testType": "MVT",
      "xof": false,
      "hexCustomization": false,
      "tests": [
        {
          "tcId": 501,
          "key":
"4389AD97264009279AD996F6BCFE30BBCF73644DBEFA109A60B3B9E3E3B29520",
          "keyLen": 256,
          "msg": "572C482D8B06A9F1493B1DB1D82621D5",
          "msgLen": 128,
          "mac":
"DF47909B75ADB5DC4B508B8C6CEFB9D2CA28F8C36BC5677CB0FCC06C7F5021...",
          "macLen": 4089,
          "customization": ""
        },
        {
          "tcId": 502,
          "key":
"71E9CAE4EA9FE46DA380B387A4F4C6A0E343B1117812E7252FDC73DB8BDC9437",
          "keyLen": 256,
          "msg": "7CA0261C96E9FEE41B2A855FC2765D2A",
          "msgLen": 128,
          "mac":
"CF0A761E9AB2D7A5CB8B6CD437541AB1F1F74FAA28F6D7896631EF9B79E93...",
          "macLen": 831,
          "customization": "."
        }
      ]
    }
  ]
}
]
```


Figure F-2

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


```
[
{ "acvVersion": <acvp-version> },
{
  "vsId": 0,
  "algorithm": "ParallelHash-128",
  "revision": "1.0",
```

```
  "testGroups": [
    {
      "tgId": 1,
      "testType": "AFT",
      "function": "ParallelHash",
      "xof": true,
      "hexCustomization": false,
      "tests": [
        {
          "tcId": 1,
          "msg": "",
          "len": 0,
          "blockSize": 64,
          "customization": "",
          "outLen": 256
        },
        {
          "tcId": 2,
          "msg": "8B30",
          "len": 12,
          "blockSize": 64,
          "customization": "O",
          "outLen": 289
        }
      ]
    },
    {
      "tgId": 3,
      "testType": "MCT",
      "function": "ParallelHash",
      "xof": true,
      "hexCustomization": false,
      "tests": [
        {
          "tcId": 501,
          "msg": "5ABA124055F84766A91603B7D1B57243",
          "len": 128
        }
      ]
    }
  ]
}
]
```

Figure F-3

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


```
[
{ "acvVersion": <acvp-version> },
{
  "vsId": 0,
  "algorithm": "TupleHash-128",
  "revision": "1.0",
  "testGroups": [
    {
      "tgId": 1,
      "testType": "AFT",
      "xof": true,
      "tests": [
        {
          "tcId": 1,
          "tuple": [],
          "len": [],
          "customization": "",
          "outLen": 256
        },
        {
          "tcId": 2,
          "tuple": [
            ""
          ],
          "len": [
            0
          ],
          "customization": "",
          "outLen": 256
        }
      ]
    },
    {
      "tgId": 3,
      "testType": "MCT",
      "xof": true,
      "tests": [
        {
          "tcId": 381,
          "tuple": [

  "B1D95CA98C5AB973C5BB25B1880A67EC1AA78582DBC7877EFDAC53EF31516E0ED0E125A5"
          ],
```

```
          "len": [
            288
          ]
        }
      ]
    }
  ]
}
]
```


Figure F-4

Appendix G — Example Test Results JSON Objects

The following is an example JSON object for cSHAKE test results sent from the crypto module to the ACVP server. The JSON objects for ParallelHash and TupleHash match this schema.


```
[
{ "acvVersion": <acvp-version> },
{
  "vsId": 0,
  "algorithm": "CSHAKE-128",
  "revision": "1.0",
  "testGroups": [
    {
      "tgId": 1,
      "tests": [
        {
          "tcId": 1,
          "md":
 "7F9C2BA4E88F827D616045507605853ED73B8093F6EFBC88EB1A6EACFA66EF26",
          "outLen": 256
        },
        {
          "tcId": 2,
          "md":
 "4DF7FFE48F76B1083A35A28D8580B15E9910BBC7C1E55B4986B7C257A1F62E36317180B322D0BFAFC0",
          "outLen": 323
        },
      ]
    },
    {
      "tgId": 2,
      "tests": [
        {
          "tcId": 251,
          "resultsArray": [
            {
              "md": "59A04B1AF85FA05A1B830B04257A382119CCE8815C29C02EFCEA0A..
.",
              "outLen": 2864
            },
            {
              "md": "B9C5B6D1CF00B17F39B5D8688F187BF974E567FA42E89221C230EF..
.",
              "outLen": 2176
            },
            {
```

```
              "md": "FEFAB0000CC69905FF217BA2E8CABB45CE9AE46AC9E8AECAC7BEA5..
.",
              "outLen": 1128
            }
          ]
        }
      ]
    }
  ]
}
]
```


Figure G-1

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


```
[
{ "acvVersion": <acvp-version> },
{
  "vsId": 0,
  "algorithm": "KMAC-128",
  "revision": "1.0",
  "testGroups": [
    {
   "tgId": 1,
      "tests": [
        {
          "tcId": 1,
          "mac":
 "5D3138562EBFFB47C88261CDDD988D077A3010EBE48AD01B75DFE5547F96963A"
        },
        {
          "tcId": 2,
          "mac":
 "FFC6F9C7D02D6D9F55434CE9301E5F6E0374EB64D11D2DCB596BEC894EB22E0787"
        }
      ]
    },
    {
      "tgId": 4,
      "tests": [
        {
          "tcId": 516,
          "testPassed": true
        },
        {
```

```
          "tcId": 517,
          "testPassed": false
        }
      ]
    }
  ]
}
]
```


Figure G-2