ACVP Deterministic Random Bit Generator (DRBG) Algorithm JSON Specification

Apostol Vassilev
Information Technology Laboratory
Computer Security Division

June 05, 2019

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

Abstract

This document defines the JSON schema for testing SP800-90A DRBG 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 SP800-90A DRBG 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 SP800-90A DRBG implementations using ACVP.

2. Supported DRBGs

The following deterministic random bit generators **MAY** be advertised by the ACVP compliant cryptographic module:

- hashDRBG-SHA-1
- hashDRBG-SHA2-224
- hashDRBG-SHA2-256
- hashDRBG-SHA2-384
- hashDRBG-SHA2-512
- hashDRBG-SHA2-512/224
- hashDRBG-SHA2-512/256
- hmacDRBG-SHA-1
- hmacDRBG-SHA2-224
- hmacDRBG-SHA2-256
- hmacDRBG-SHA2-384
- hmacDRBG-SHA2-512
- hmacDRBG-SHA2-512/224
- hmacDRBG-SHA2-512/256
- ctrDRBG-AES-128
- ctrDRBG-AES-192
- ctrDRBG-AES-256
- ctrDRBG-TDES

2.1. Counter DRBG Triple-DES

The “ctrDRBG-TDES” mode shall only be used with the three-key option of the Triple-DES algorithm.

2.2. Supported values per DRBG option

DRBG minimum/maximum values for several options such as minimum entropy and nonce, vary depending on the DRBG capabilities registered. The following table depicts those values

Table 1 — Supported DRBG Values

DRBG Algorithm	Mode	Derivation Function	Security Strength	Min Entropy	Max Entropy	Max PersoString	Max Addl String	Min Nonce
Counter	AES128	TRUE	128	128	65536	65536	65536	64

DRBG Algorithm	Mode	Derivation Function	Security Strength	Min Entropy	Max Entropy	Max PersoString	Max Addl String	Min Nonce
Counter	AES192	TRUE	192	192	65536	65536	65536	96
Counter	AES256	TRUE	256	256	65536	65536	65536	128
Counter	TDES	TRUE	112	112	65536	65536	65536	56
Counter	AES128	FALSE	128	256	256	256	256	0
Counter	AES192	FALSE	192	320	320	320	320	0
Counter	AES256	FALSE	256	384	384	384	384	0
Counter	TDES	FALSE	112	232	232	232	232	0
Hash	SHA1	N/A	80	80	65536	65536	65536	40
Hash	SHA2-224	N/A	112	112	65536	65536	65536	56
Hash	SHA2-256	N/A	128	128	65536	65536	65536	64
Hash	SHA2-384	N/A	192	192	65536	65536	65536	96
Hash	SHA2-512	N/A	256	256	65536	65536	65536	128
Hash	SHA2-512/224	N/A	112	112	65536	65536	65536	56
Hash	SHA2-512/256	N/A	128	128	65536	65536	65536	64
Hmac	SHA1	N/A	128	128	65536	65536	65536	64
Hmac	SHA2-224	N/A	192	192	65536	65536	65536	96
Hmac	SHA2-256	N/A	256	256	65536	65536	65536	128
Hmac	SHA2-384	N/A	256	256	65536	65536	65536	128
Hmac	SHA2-512	N/A	256	256	65536	65536	65536	128
Hmac	SHA2-512/224	N/A	192	192	65536	65536	65536	96
Hmac	SHA2-512/256	N/A	256	256	65536	65536	65536	128

3. Test Types and Test Coverage

The ACVP server performs a set of tests on the IUT's DRBG in order to assess the correctness and robustness of the implementation. A typical ACVP validation session would require multiple tests to be performed for every supported permutation of DRBG capabilities. This section describes the design of the tests used to validate implementations of the DRBG algorithms. There is a single test type for DRBG testing:

- "AFT" — Algorithm Function Test. In the AFT test mode, the IUT **MUST** be capable of injecting the values provided by the ACVP server, into their IUT's implementation. The IUT is **REQUIRED** to run the DRBG function calls, depending on registration options, as defined in Table 7

3.1. Test Coverage

The tests described in this document have the intention of ensuring an implementation is conformant to [SP 800-90A].

3.1.1. Requirements Covered

- SP 800-90A — 7.1 Entropy Input. The IUT is **REQUIRED** to inject the ACVP server's provided entropy for testing.
- SP 800-90A — 7.2 Other Inputs. The IUT is **REQUIRED** to inject the ACVP server's provided other input information for testing.
- SP 800-90A — 7.3 Internal State. Indirect testing of the IUT's DRBG internal state **SHALL** be inferred through multiple calls to the DRBG "generate" function. Multiple calls **SHALL** ensure the internal state is successfully mutated for each "generate" invocation.
- SP 800-90A — 7.4 The DRBG Mechanism Functions. "Instantiate", "Generate", and "Reseed" DRBG functions **SHALL** be tested within the ACVP server's provided tests.
- SP 800-90A — 8 DRBG Mechanism Concepts and General Requirements. The ACVP server **SHALL** validate "Instantiate", "Generate", and "Reseed" are properly implemented. Reseeding is partially tested through ACVP test vectors through an explicit reseed operation.
- SP 800-90A — 9 DRBG Mechanism Functions. "Instantiate", "Generate", and "Reseed" **SHALL** be evaluated as a part of ACVP generated tests.
- SP 800-90A — 10 DRBG Algorithm Specifications. "Instantiate", "Generate", and "Reseed" DRBG functions in scope **SHALL** be tested as per the specifications in this section.

3.1.2. Requirements Not Covered

- SP 800-90A — 7.1 Entropy Input. The ACVP Server **SHALL** provide all instances of randomness to utilize from the IUT's perspective. Implementation of the IUT's RBG **SHALL NOT** be in scope of testing.

- SP 800-90A — 7.2 Other Inputs. The ACVP server **SHALL** provide all instances of randomness throughout the IUT's testing of the DRBG.
- SP 800-90A — 7.3 Internal State. Though direct testing of the IUT's internal state **SHALL NOT** be not performed, the act of testing multiple "generate" outputs from the IUT DRBG helps to ensure a successful IUT implementation.
- SP 800-90A — 7.4 The DRBG Mechanism Functions. "Uninstantiate", "Health Test" **SHALL NOT** be in scope of testing.
- SP 800-90A — 8 DRBG Mechanism Concepts and General Requirements. The ACVP server **SHALL NOT** directly validate internal DRBG state. Additionally, DRBG boundaries are out of scope of ACVP testing. Seed construction is performed by the ACVP server, the IUT is **REQUIRED** to utilize the ACVP provided seed (via entropy, inputs, etc) to perform validation testing. Reseeds operations that are performed automatically due to a large number of generate operations, **SHALL NOT** be in scope of ACVP testing.
- SP 800-90A — 9 DRBG Mechanism Functions. Error conditions (CATASTROPHIC_ERROR_FLAG or otherwise), reseeding due to end of seed life, uninstantiation, and health checks are out of scope of ACVP tests.
- SP 800-90A — 10 DRBG Algorithm Specifications. All previously mentioned DRBG functions that are not in scope — Uninstantiate, Health check, automatic reseed, error conditions — **SHALL NOT** be tested as part of testing.
- SP 800-90A — 11 Assurances. Health check and error handling testing **SHALL NOT** be performed within the scope of ACVP testing.

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 DRBG 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 2 — 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. Property Registration

The SP800-90A DRBG mode capabilities are advertised as JSON objects within the 'capabilities_exchange' property.

4.3. Default values

ACVP has default values for many of the input parameters for testing the DRBG algorithms.For example, the Entropy Input, Nonce, Personalization String, and Addtional Input parameters have default values. The specific details and restrictions on each of these input lengths is specified in Section 2.2, Table 3 and the notes following it. To indicate a preference for using a default value for any of these parameters, the value zero (0) should be set. If the implementation does not support one of these defaults, the corresponding supported bit length values shall be set explicitly.

4.4. Registration Example

A registration **SHALL** use these properties within each object within the 'capabilities' array

Table 3 — Registration Capability Properties

JSON Value	Description	JSON type	Valid Values
mode	The algorithm mode to be validated	string	See Section 2
derFuncEnabled	Derivation function option. See Table 3 notes below	boolean	true/false
entropyInputLen	See Table 3 notes below	domain	Min: maximum security strength, Max: XXX
nonceLen	See Table 3 notes below	domain	Min: half the maximum security strength, Max: XXX. Set to 0 if not supported
persoStringLen	Personalization string length. See Table 3 notes below	domain	Min: maximum security strength, Max: XXX. Set to 0 if not supported
additionalInputLen	See Table 3 notes below	domain	Min: maximum security strength, Max: XXX. Set to 0 if not supported
returnedBitsLen	See Table 3 notes below	integer	

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

Table 4 — DRBG Algorithm Capabilities

JSON Value	Description	JSON type	Valid Values
algorithm	Name of the algorithm to be validated	string	See Section 2
revision	ACVP Test version	string	"1.0"
prereqVals	Prerequisites of the algorithm	object	See Section 4.1
predResistanceEnabled	An implementation that can be used with prediction resistance. See Table 3 notes below	array of boolean containing one or two distinct values	[true], [true, false], or [false]
reseedImplemented	Reseeding of the DRBG shall be performed in accordance with the specification for the given DRBG mechanism. See Table 3 notes below	boolean	true or false

NOTE 1 – 2 If an implementation utilizes a nonce in the construction of a seed during instantiation, the length of the nonce shall be at least half the maximum security strength supported. See Tables 2 and 3 in [SP 800-90A] for help on choosing appropriate parameter values for the tested DRBG implementation.

NOTE 2 – 3 If an implementation can only be used without prediction resistance, the array 'predResistanceEnabled' shall only contain a single 'false' element. Implementations that either have prediction resistance always ON or always OFF, the array 'predResistanceEnabled' shall contain two distinct elements, 'true' and 'false'. Implementations containing multiple equal array elements for 'predResistanceEnabled' will be rejected.

NOTE 3 – 4 For 'ctrDRBG' implementations, the 'derFuncEnabled' property must be included.

NOTE 4 – 5 All DRBGs are tested at their maximum supported security strength so this is the minimum bit length of the entropy input that ACVP will accept. The maximum supported security strength is also the default value for this input. Longer entropy inputs are permitted, with the following exception: for 'ctrDRBG' with 'derFuncEnabled' set to 'false', the 'entropyInputLen' must equal the seed length. See Tables 2 and 3 in [SP 800-90A] for help on choosing appropriate parameter values for the DRBG being tested.

NOTE 5 – 6 'ctrDRBG' with 'derFuncEnabled' set to 'false' does not use a nonce; the nonce values, if supplied, will be ignored for this case. The default nonce bit length is one-half the maximum security strength supported by the mechanism/option. See Tables 2 and 3 in [SP 800-90A] for help on choosing appropriate parameter values for the tested DRBG implementation.

NOTE 6 – 7 ACVP allows bit length values for 'persoString' ranging from the maximum supported security strength except in the case of 'derFuncEnabled' set to 'false', where the second personalization string length must

be less than or equal to the seed length. If the implementation only supports one personalization string length, then set only that value as the range min and max and set the step to 0. If the implementation does not use a 'persoString', set all range parameters (min, max, step) to 0. If the implementation can work with and without 'persoString', set the min to 0, set the max to at least the maximum supported strength and set the step equal to at least the maximum supported strength to avoid testing lengths less than that. See Tables 2 and 3 in [SP 800-90A] for help on choosing appropriate parameter values for the tested DRBG implementation.

NOTE 7 – 8 The 'addtionalInput' configuration and restrictions are the same as those for the 'persoString'.

An example registration within an algorithm capability exchange looks like this

```
{
  "algorithm": "ctrDRBG",
  "revision": "1.0",
  "predResistanceEnabled": [
    true,
    false
  ],
  "reseedImplemented": true,
  "capabilities": [
    {
      "mode": "TDES",
      "derFuncEnabled": true,
      "entropyInputLen": [
        232
      ],
      "nonceLen": [
        232
      ],
      "persoStringLen": [
        232
      ],
      "additionalInputLen": [
        232
      ],
      "returnedBitsLen": 512
    },
    {
      "mode": "AES-128",
      "derFuncEnabled": false,
      "entropyInputLen": [
        256
      ],
      "nonceLen": [
        256
      ],
```

```
      "persoStringLen": [
        256
      ],
      "additionalInputLen": [
        256
      ],
      "returnedBitsLen": 512
    }
  ]
}
```

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 SP800-90A DRBG 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 5 — 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

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 SP800-90A DRBG JSON elements of the Test Group JSON object

Table 6 — Test Group JSON Object

JSON Value	Description	JSON type
tgId	Test group identifier	integer
mode	The mode of the DRBG, see Section 2	string
derFunc	Use derivation function or not	boolean
predResistance	Use prediction resistance	boolean
reSeed	Use reseeding	boolean
entropyInputLen	Entropy length	integer
nonceLen	Nonce length; set to 0 if not used/ supported. See also notes after Table 3 above	integer
persoStringLen	Personalization string length; set to 0 if not used/supported. See also notes after Table 3 above	integer
additonalInputLen	Additional input length; set to 0 if not used/supported. See also notes after Table 3 above	integer
returnedBitsLen	returned bits length	integer
tests	Array of individual test cases	array

The ‘tgId’, ‘testType’ and ‘tests’ objects **MUST** appear in every test group element communicated from the server to the client as a part of a prompt. Other properties are dependent on which ‘testType’ (see Section 3) the group is addressing.

NOTE – 11 According to SP 800-90A [SP 800-90A], a DRBG implementation has two separate controls for determining the correct test procedure for handling addtional entropy and other data in providing prediction resistance assurances. Depending on the capabilities advertised by the predResistanceEnabled and reseedImplemented flags ACVP generates test data according to the following test scenarios:

Table 7 — Generated Test Data per Scenario

Prediction Resistance Assurance Options	Test Procedure
“predResistanceEnabled” : true; “reseedImplemented”: true	
	Instantiate DRBG
	Generate but don’t output
	Generate output
	Uninstantiate
“predResistanceEnabled” : false; “reseedImplemented” : true	
	Instantiate DRBG
	Reseed
	Generate but don’t output
	Generate output
	Uninstantiate

Prediction Resistance Assurance Options	Test Procedure
“predResistanceEnabled” : true/false; “reseedImplemented”: false	
	Instantiate DRBG
	Generate but don’t output
	Generate output
	Uninstantiate

5.2. Test Cases

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 SP800-90A DRBG test vector.

Table 8 — Test Case JSON Object

JSON Value	Description	JSON type
tcId	Test case idenfitier	integer
entropyInput	Entropy value	hex
nonce	Value of the nonce	hex
persoString	value of the personlization string	hex
otherInput	array of additonal input/entropy input value pairs for testing. See Table 9	array

Each test group contains an array of one or more tests. Each test object contains an otherInput object, which is an array of objects, each with the intendedUse property indicating if the particular test data is to be used for reSeed or generate — see Table 7. Each test vector is a JSON object that represents a single test case to be processed by the ACVP client. The following table describes the JSON elements for each DRBG predcition resistance test vector.

Table 9 — DRBG Predictive Resistance JSON Elements

JSON Value	Description	JSON type
additionalInput	value of the additional input string to use in predition resistance tests	hex
entropyInput	value of the entropy input to use in prediction resistance tests	hex
intendedUse	“reSeed”, “generate”	string

Here is an abbreviated yet fully constructed example of the prompt

```
{
  "vsId": 1,
  "algorithm": "ctrDRBG",
  "revision": "1.0",
  "testGroups": [
```

```
{
  "tgId": 1,
  "testType": "AFT",
  "derFunc": true,
  "reSeed": true,
  "predResistance": true,
  "entropyInputLen": 256,
  "nonceLen": 256,
  "persoStringLen": 256,
  "additionalInputLen": 256,
  "returnedBitsLen": 512,
  "mode": "AES-128",
  "tests": [
    {
      "tcId": 1,
      "entropyInput": "E9EDA8BF1E6155BDF11AD74E2702004C20B39...",
      "nonce": "D77D611F0665CBFD7E00D5E5118629F5F40996B764F0...",
      "persoString": "E9ADEA726418EF002C03DC2196296D4B273AB6...",
      "otherInput": [
        {
          "intendedUse": "generate",
          "additionalInput": "52F47C6A1B12C202D309D062C3EE09...",
          "entropyInput": "C30F4C916B90A79B5764DC6FA950B3F34..."
        },
        {
          "intendedUse": "generate",
          "additionalInput": "4FFDE712D249A99006F46D7070D5CA...",
          "entropyInput": "8A4724F1514C480DE1604C5D870CFA464..."
        }
      ]
    },
    {
      "tcId": 2,
      "entropyInput": "DC1B4E9B1782A9E701CB2A74EDBDF483462E9...",
      "nonce": "A5B7D117BDE77D46A65DBD0EBA085C4376C7B72F164E...",
      "persoString": "D7FC54DD4E759C8D3FCE61463ED40BE130D2B4...",
      "otherInput": [
        {
          "intendedUse": "generate",
          "additionalInput": "28EFD5114D06D1A065863C50BDD2DE...",
          "entropyInput": "F5C9DF132A4C066C5D0AFEE79FBC7EB07..."
        },
        {
          "intendedUse": "generate",
          "additionalInput": "D1AACBAE8BA37208161CBA9042BB92...",
          "entropyInput": "5C6D2F80696D1691FCA40B0C3444CE927..."
```

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

Figure 4

6. 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 10 — Vector Set Response JSON Object

JSON Property	Description	JSON Type
acvVersion	The version of the protocol	string
vsId	The vector set identifier	integer
testGroups	The test group data	array

An example of this is the following

```
{
 "acvVersion": "version",
 "vsId": 1,
 "testGroups": [ ... ]
}
```

Figure 5

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 11 — Vector Set Group Response JSON Object

JSON Property	Description	JSON Type
tgId	The test group identifier	integer
tests	The test case data	array

An example of this is the following

```
{
 "tgId": 1,
 "tests": [ ... ]
}
```

Figure 6

The following table describes the JSON object that represents a test case response for a SP800-90A DRBG.

Table 12 — Test Case Results JSON Object

JSON Property	Description	JSON Type
tcId	The test case identifier	integer

JSON Property	Description	JSON Type
returnedBits	The outputted bits from the DRBG	hex

Here is an abbreviated example of the response

```
{
  "vsId": 1,
  "algorithm": "ctrDRBG",
  "revision": "1.0",
  "testGroups": [
    {
      "tgId": 1,
      "tests": [
        {
          "tcId": 1,
          "returnedBits": "99F0D5740DCAA1ECC4E5329B38B..."
        },
        {
          "tcId": 2,
          "returnedBits": "62A458CA72C19316A1ECBC3211B..."
        }
      ]
    }
  ]
}
```

Figure 7

7. Security Considerations

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

8. Example DRBG Capabilities JSON Object

The following is a example JSON object advertising support for ctrDRBG with TDES and all key sizes of AES.

{ "algorithm": "ctrDRBG", "revision": "1.0", "prereqVals": [{ "algorithm": "AES", "valValue": "1234" }, { "algorithm": "TDES", "valValue": "5678" }], "predResistanceEnabled": [true, false], "reseedImplemented": true, "capabilities": [{ "mode": "AES-128", "derFuncEnabled": true, "entropyInputLen": [256], "nonceLen": [256], "persoStringLen": [256], "additionalInputLen": [256], "returnedBitsLen": 512 }, { "mode": "AES-192", "derFuncEnabled": true, "entropyInputLen": [320], "nonceLen": [320], "persoStringLen": [320], "additionalInputLen": [320], "returnedBitsLen": 512 }, { "mode": "AES-256", "derFuncEnabled": true, "entropyInputLen": [384], "nonceLen": [384], "persoStringLen": [384], "additionalInputLen": [384], "returnedBitsLen": 512 }, { "mode": "TDES", "derFuncEnabled": true, "entropyInputLen": [232], "nonceLen": [232], "persoStringLen": [232], "additionalInputLen": [232], "returnedBitsLen": 512 }, { "mode": "AES-128", "derFuncEnabled": false, "entropyInputLen": [256], "nonceLen": [256], "persoStringLen": [256], "additionalInputLen": [256], "returnedBitsLen": 512 }, { "mode": "AES-192", "derFuncEnabled": false, "entropyInputLen": [320], "nonceLen": [320], "persoStringLen": [320], "additionalInputLen": [320], "returnedBitsLen": 512 }, { "mode": "AES-256", "derFuncEnabled": false, "entropyInputLen": [384], "nonceLen": [384], "persoStringLen": [384], "additionalInputLen": [384], "returnedBitsLen": 512 }, { "mode": "TDES", "derFuncEnabled": false, "entropyInputLen": [232], "nonceLen": [232], "persoStringLen": [232], "additionalInputLen": [232], "returnedBitsLen": 512 }] }

Figure 8

The following is a example JSON object advertising support for hashDRBG with various SHA sizes. Note that in this example the implementation works with or without additional input and personalization data.

{ "algorithm": "hashDRBG", "revision": "1.0", "prereqVals": [{ "algorithm": "AES", "valValue": "1234" }, { "algorithm": "SHA", "valValue": "5678" }], "predResistanceEnabled": [true, false], "reseedImplemented": true, "capabilities": [{ "mode": "SHA-1", "derFuncEnabled": false, "entropyInputLen": [160], "nonceLen": [160], "persoStringLen": [160], "additionalInputLen": [160], "returnedBitsLen": 640 }, { "mode": "SHA2-224", "derFuncEnabled": false, "entropyInputLen": [224], "nonceLen": [224], "persoStringLen": [224], "additionalInputLen": [224], "returnedBitsLen": 896 }, { "mode": "SHA2-256", "derFuncEnabled": false, "entropyInputLen": [256], "nonceLen": [256], "persoStringLen": [256], "additionalInputLen": [256], "returnedBitsLen": 1024 }, { "mode": "SHA2-384", "derFuncEnabled": false, "entropyInputLen": [384], "nonceLen": [384], "persoStringLen": [384], "additionalInputLen": [384], "returnedBitsLen": 1536 }, { "mode": "SHA2-512", "derFuncEnabled": false, "entropyInputLen": [512], "nonceLen": [512], "persoStringLen": [512], "additionalInputLen": [512], "returnedBitsLen": 2048 }, { "mode": "SHA2-512/224", "derFuncEnabled": false, "entropyInputLen": [224], "nonceLen": [224], "persoStringLen": [224], "additionalInputLen": [224], "returnedBitsLen": 896 }, { "mode": "SHA2-512/256",

"derFuncEnabled": false, "entropyInputLen": [256], "nonceLen": [256], "persoStringLen": [256], "additionalInputLen": [256], "returnedBitsLen": 1024 }] }

Figure 9

9. Example Test Vectors JSON Object

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

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1133, "algorithm": "ctrDRBG", "mode": "TDES", "revision": "1.0", "testGroups": [{ "tgId": 1, "derFunc": true, "predResistance": true, "reSeed": true, "entropyInputLen": 112, "nonceLen": 56, "persoStringLen": 112, "additionalInputLen": 112, "returnedBitsLen": 256, "tests": [{ "tcId": 1815, "entropyInput":"78aac2cb444594e2...", "nonce":"41ef9c67ffe438", "persoString":"b8e84de200a9239a0...", "otherInput" : [{ "intendedUse" : "generate", "additionalInput":"f1e8edf0...", "entropyInput": "6cd4096638..."}, { "intendedUse" : "generate", "additionalInput" : "f535773...", "entropyInput" : "a0cdf5c1c6..."}] }, { "tcId": 1816, "entropyInput" : "b8ab88b9c5fda8...", "nonce": "f1bcc6ff60dd37", "persoString" : "018c1f9d22f3c7f...", "otherInput" : [{"intendedUse" : "generate", "additionalInput" : "356a6e9...", "entropyInput" : "bed693401b..."}, {"intendedUse" : "generate", "additionalInput" : "4321b3a...", "entropyInput" : "a632ef16f2..."}] }] }] }]

Figure 10

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

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1146, "algorithm": "hmacDRBG", "mode": "AES-256", "revision": "1.0", "testGroups": [{ "tgId": 1, "predResistance": true, "reSeed": false, "entropyInputLen": 256, "nonceLen": 128, "persoStringLen": 256, "additionalInputLen": 256, "returnedBitsLen": 1024, "tests": [{ "tcId": 2111, "entropyInput": "ee3392c5f3de6f3...", "nonce": "b991a820fac75fd02642ad...", "persoString": "30f3a50b0e2309da...", "otherInput" : [{"intendedUse" : "generate", "additionalInput":"4ea46abe...", "entropyInput": "e4413a2e40..."}, {"intendedUse" : "generate", "additionalInput" : "61b7204...", "entropyInput" : "968ea185d1..."}] }, { "tcId": 2112, "entropyInput" : "a0ace75784b972...", "nonce": "b671308068fc7909a360c7...", "persoString" : "338d5f2bd93262d...", "otherInput" : [{"intendedUse" : "generate", "additionalInput" : "7acd8bf...", "entropyInput" : "47b26bbe93..."}, {"intendedUse" : "generate", "additionalInput" : "d4b24c7...", "entropyInput" : "acb63f3b59..."}] }] }] }]

Figure 11

The following is a example JSON object for hashDRBG test vectors sent from the ACVP server to the crypto module. In this example the implementation is tested without additional input and personalization data.

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1156, "algorithm": "hashDRBG", "mode": "SHA2-256", "revision": "1.0", "testGroups": [{ "tgId": 1, "predResistance": true, "reSeed": false, "entropyInputLen": 256, "nonceLen": 128, "persoStringLen": 0, "additionalInputLen": 0, "returnedBitsLen": 1024, "tests": [{ "tcId": 2151, "entropyInput": "ae0a3acd541d0d5...", "nonce": "786f03ad697332d74fad7a...", "persoString": "", "otherInput" : [{"intendedUse" : "generate", "additionalInput": "", "entropyInput": "4852aed7c..."}, {"intendedUse" : "generate", "additionalInput" : "", "entropyInput" : "8b8a35a1..."}] }, { "tcId": 2152, "entropyInput" : "26d8c9a9b982cd...", "nonce": "36dff124f908a95a022edf...", "persoString" : "", "otherInput" :

[{"intendedUse" : "generate", "additionalInput" : "", "entropyInput" : "648bbdc4d4..."}, {"intendedUse" : "generate", "additionalInput" : "", "entropyInput" : "fff51d05b1..."}] }] }] }]

Figure 12

The following is a example JSON object for hashDRBG test vectors sent from the ACVP server to the crypto module. In this example the implementation is tested with “predResistance”: false, “reSeed”: true options.

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1157, "algorithm": "hashDRBG", "mode": "SHA2-256", "revision": "1.0", "testGroups": [{ "tgId": 1, "predResistance": false, "reSeed": true, "entropyInputLen": 256, "nonceLen": 128, "persoStringLen": 256, "additionalInputLen": 256, "returnedBitsLen": 1024, "tests": [{ "tcId": 3151, "entropyInput": "860d051cedbb935...", "nonce": "5813070f9774d21e644d64...", "persoString": "545ba29faf1bb1bf...", "otherInput" : [{"intendedUse" : "reSeed", "additionalInput": "95b08...", "entropyInput": "2e92955b1..."}, {"intendedUse" : "generate", "additionalInput" : "ddfa...", "entropyInput" : ""}, {"intendedUse" : "generate", "additionalInput" : "edb88...", "entropyInput" : ""}] }, { "tcId": 3152, "entropyInput" : "371d2944c9ace6...", "nonce": "4bb34ab1e882d97687c3f8...", "persoString" : "c5b03354a9fad34...", "otherInput" : [{"intendedUse" : "reSeed", "additionalInput" : "6e3fa8e...", "entropyInput" : "afd7e6b0b4..."}, {"intendedUse" : "generate", "additionalInput" : "deb8ed5...", "entropyInput" : ""}, {"intendedUse" : "generate", "additionalInput" : "a554bb9...", "entropyInput" : ""}] }] }] }]

Figure 13

The following is a example JSON object for hashDRBG test vectors sent from the ACVP server to the crypto module. In this example the implementation is tested with “predResistance”: false, “reSeed”: false options.

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1167, "algorithm": "hashDRBG", "mode": "SHA2-256", "revision": "1.0", "testGroups": [{ "tgId": 1, "predResistance": false, "reSeed": false, "entropyInputLen": 256, "nonceLen": 128, "persoStringLen": 256, "additionalInputLen": 256, "returnedBitsLen": 1024, "tests": [{ "tcId": 4151, "entropyInput": "090db63c22de171...", "nonce": "6f7c6bec9825079cabd947...", "persoString": "c2f1a59806197792...", "otherInput" : [{"intendedUse" : "generate", "additionalInput": "3fc72d...", "entropyInput": ""}, {"intendedUse" : "generate", "additionalInput" : "968a3...", "entropyInput" : ""}] }, { "tcId": 4152, "entropyInput" : "bd0e2dbba872bb...", "nonce": "a97dfbaea505a3e36210a8...", "persoString" : "7d0de87d097551f...", "otherInput" : [{"intendedUse" : "generate", "additionalInput" : "fe1adf1...", "entropyInput" : ""}, {"intendedUse" : "generate", "additionalInput" : "1df719a...", "entropyInput" : ""}] }] }] }]

Figure 14

10. Example Test Results JSON Object

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

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1133, "testGroups": [{ "tgId": 1, "tests": [{ "tcId": 1815, "returnedBits ": "4565e85447af71..." }, { "tcId": 1816, "returnedBits": "b67acc3b2231ec5..." }] }] }]

Figure 15

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

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1146, "testGroups": [{ "tgId": 1, "tests": [{ "tcId": 2111, "returnedBits": "e42130fd1d920a2bc..." }, { "tcId": 2112, "returnedBits": "495b2a0de6b5fc454..." }] }] }]

Figure 16

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

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1156, "testGroups": [{ "tgId": 1, "tests": [{ "tcId": 2151, "returnedBits ": "1af967534c670271..." }, { "tcId": 2152, "returnedBits": "8a74a8c31ea4e6e62..." }] }] }]

Figure 17

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

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1157, "testGroups": [{ "tgId": 1, "tests": [{ "tcId": 3151, "returnedBits ": "0eadc82746890ee0..." }, { "tcId": 3152, "returnedBits": "6452be2ee730d7245..." }] }] }]

Figure 18

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

[{ "acvVersion": <acvp-version> }, { "vectorSetId": 1167, "testGroups": [{ "tgId": 1, "tests": [{ "tcId": 4151, "returnedBits ": "5dbfd26651bc7159..." }, { "tcId": 4152, "returnedBits": "ff3cce0b5585172b1..." }] }] }]

Figure 19

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	2019-06-05	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.

Elaine B. Barker, John M. Kelsey (January 2012) *Recommendation for Random Number Generation Using Deterministic Random Bit Generators* (Gaithersburg, MD), January 2012. SP 800-90A. https://doi.org/10.6028/NIST.SP.800-90A.

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.