Stream: Independent Submission
RFC: 8785
Category: Informational
Published: June 2020
ISSN: 2070-1721
Authors: A. Rundgren *Independent*, B. Jordan *Broadcom*, S. Erdtman *Spotify AB*

RFC 8785
JSON Canonicalization Scheme (JCS)

Abstract

Cryptographic operations like hashing and signing need the data to be expressed in an invariant format so that the operations are reliably repeatable. One way to address this is to create a canonical representation of the data. Canonicalization also permits data to be exchanged in its original form on the "wire" while cryptographic operations performed on the canonicalized counterpart of the data in the producer and consumer endpoints generate consistent results.

This document describes the JSON Canonicalization Scheme (JCS). This specification defines how to create a canonical representation of JSON data by building on the strict serialization methods for JSON primitives defined by ECMAScript, constraining JSON data to the Internet JSON (I-JSON) subset, and by using deterministic property sorting.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This is a contribution to the RFC Series, independently of any other RFC stream. The RFC Editor has chosen to publish this document at its discretion and makes no statement about its value for implementation or deployment. Documents approved for publication by the RFC Editor are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8785.

Copyright Notice

Table of Contents

1. Introduction

This document describes the JSON Canonicalization Scheme (JCS). This specification defines how to create a canonical representation of JSON [RFC8259] data by building on the strict serialization methods for JSON primitives defined by ECMAScript [ECMA-262], constraining JSON data to the I-JSON [RFC7493] subset, and by using deterministic property sorting. The output from JCS is a "hashable" representation of JSON data that can be used by cryptographic methods. The subsequent paragraphs outline the primary design considerations.

Cryptographic operations like hashing and signing need the data to be expressed in an invariant format so that the operations are reliably repeatable. One way to accomplish this is to convert the data into a format that has a simple and fixed representation, like base64url [RFC4648]. This is how JSON Web Signature (JWS) [RFC7515] addressed this issue. Another solution is to create a canonical version of the data, similar to what was done for the XML signature [XMLDSIG] standard.

The primary advantage with a canonicalizing scheme is that data can be kept in its original form. This is the core rationale behind JCS. Put another way, using canonicalization enables a JSON object to remain a JSON object even after being signed. This can simplify system design, documentation, and logging.

To avoid "reinventing the wheel", JCS relies on the serialization of JSON primitives (strings, numbers, and literals), as defined by ECMAScript (aka JavaScript) [ECMA-262] beginning with version 6.

Seasoned XML developers may recall difficulties getting XML signatures to validate. This was usually due to different interpretations of the quite intricate XML canonicalization rules as well as of the equally complex Web Services security standards. The reasons why JCS should not suffer from similar issues are:

- JSON does not have a namespace concept and default values.
- Data is constrained to the I-JSON [RFC7493] subset. This eliminates the need for specific parsers for dealing with canonicalization.
- JCS-compatible serialization of JSON primitives is currently supported by most web browsers as well as by Node.js [NODEJS].
- The full JCS specification is currently supported by multiple open-source implementations (see Appendix G). See also Appendix F for implementation guidelines.

JCS is compatible with some existing systems relying on JSON canonicalization such as JSON Web Key (JWK) Thumbprint [RFC7638] and Keybase [KEYBASE].

For potential uses outside of cryptography, see [JSONCOMP].

The intended audiences of this document are JSON tool vendors as well as designers of JSON-based cryptographic solutions. The reader is assumed to be knowledgeable in ECMAScript, including the "JSON" object.

2. Terminology

Note that this document is not on the IETF standards track. However, a conformant implementation is supposed to adhere to the specified behavior for security and interoperability reasons. This text uses BCP 14 to describe that necessary behavior.

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 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Detailed Operation

This section describes the details related to creating a canonical JSON representation and how they are addressed by JCS.

Appendix F describes the **RECOMMENDED** way of adding JCS support to existing JSON tools.

3.1. Creation of Input Data

Data to be canonically serialized is usually created by:

- Parsing previously generated JSON data.
- Programmatically creating data.

Irrespective of the method used, the data to be serialized **MUST** be adapted for I-JSON [RFC7493] formatting, which implies the following:

- JSON objects **MUST NOT** exhibit duplicate property names.
- JSON string data **MUST** be expressible as Unicode [UNICODE].
- JSON number data **MUST** be expressible as IEEE 754 [IEEE754] double-precision values. For applications needing higher precision or longer integers than offered by IEEE 754 double precision, it is **RECOMMENDED** to represent such numbers as JSON strings; see Appendix D for details on how this can be performed in an interoperable and extensible way.

An additional constraint is that parsed JSON string data **MUST NOT** be altered during subsequent serializations. For more information, see Appendix E.

Note: Although the Unicode standard offers the possibility of rearranging certain character sequences, referred to as "Unicode Normalization" [UCNORM], JCS-compliant string processing does not take this into consideration. That is, all components involved in a scheme depending on JCS **MUST** preserve Unicode string data "as is".

3.2. Generation of Canonical JSON Data

The following subsections describe the steps required to create a canonical JSON representation of the data elaborated on in the previous section.

Appendix A shows sample code for an ECMAScript-based canonicalizer, matching the JCS specification.

3.2.1. Whitespace

Whitespace between JSON tokens **MUST NOT** be emitted.

3.2.2. Serialization of Primitive Data Types

Assume the following JSON object is parsed:

```
{
  "numbers": [333333333.33333329, 1E30, 4.50,
              2e-3, 0.000000000000000000000000001],
  "string": "\u20ac$\u000F\u000aA'\u0042\u0022\u005c\\\"\/",
  "literals": [null, true, false]
}
```

If the parsed data is subsequently serialized using a serializer compliant with ECMAScript's "JSON.stringify()", the result would (with a line wrap added for display purposes only) be rather divergent with respect to the original data:

```
{"numbers":[333333333.3333333,1e+30,4.5,0.002,1e-27],"string":
"€$\u000f\nA'B\"\\\\\"/","literals":[null,true,false]}
```

The reason for the difference between the parsed data and its serialized counterpart is due to a wide tolerance on input data (as defined by JSON [RFC8259]), while output data (as defined by ECMAScript) has a fixed representation. As can be seen in the example, numbers are subject to rounding as well.

The following subsections describe the serialization of primitive JSON data types according to JCS. This part is identical to that of ECMAScript. In the (unlikely) event that a future version of ECMAScript would invalidate any of the following serialization methods, it will be up to the developer community to either stick to this specification or create a new specification.

3.2.2.1. Serialization of Literals

In accordance with JSON [RFC8259], the literals "null", "true", and "false" **MUST** be serialized as null, true, and false, respectively.

3.2.2.2. Serialization of Strings

For JSON string data (which includes JSON object property names as well), each Unicode code point **MUST** be serialized as described below (see Section 24.3.2.2 of [ECMA-262]):

- If the Unicode value falls within the traditional ASCII control character range (U+0000 through U+001F), it **MUST** be serialized using lowercase hexadecimal Unicode notation (\uhhhh) unless it is in the set of predefined JSON control characters U+0008, U+0009, U+000A, U+000C, or U+000D, which **MUST** be serialized as \b, \t, \n, \f, and \r, respectively.
- If the Unicode value is outside of the ASCII control character range, it **MUST** be serialized "as is" unless it is equivalent to U+005C (\) or U+0022 ("), which **MUST** be serialized as \\ and \", respectively.

Finally, the resulting sequence of Unicode code points **MUST** be enclosed in double quotes (").

Note: Since invalid Unicode data like "lone surrogates" (e.g., U+DEAD) may lead to interoperability issues including broken signatures, occurrences of such data **MUST** cause a compliant JCS implementation to terminate with an appropriate error.

3.2.2.3. Serialization of Numbers

ECMAScript builds on the IEEE 754 [IEEE754] double-precision standard for representing JSON number data. Such data **MUST** be serialized according to Section 7.1.12.1 of [ECMA-262], including the "Note 2" enhancement.

Due to the relative complexity of this part, the algorithm itself is not included in this document. For implementers of JCS-compliant number serialization, Google's implementation in V8 [V8] may serve as a reference. Another compatible number serialization reference implementation is Ryu [RYU], which is used by the JCS open-source Java implementation mentioned in Appendix G. Appendix B holds a set of IEEE 754 sample values and their corresponding JSON serialization.

Note: Since Not a Number (NaN) and Infinity are not permitted in JSON, occurrences of NaN or Infinity **MUST** cause a compliant JCS implementation to terminate with an appropriate error.

3.2.3. Sorting of Object Properties

Although the previous step normalized the representation of primitive JSON data types, the result would not yet qualify as "canonical" since JSON object properties are not in lexicographic (alphabetical) order.

Applied to the sample in Section 3.2.2, a properly canonicalized version should (with a line wrap added for display purposes only) read as:

```
{"literals":[null,true,false],"numbers":[333333333.3333333,
1e+30,4.5,0.002,1e-27],"string":"€$\u000f\nA'B\"\\\\\"/"}
```

The rules for lexicographic sorting of JSON object properties according to JCS are as follows:

- JSON object properties **MUST** be sorted recursively, which means that JSON child Objects **MUST** have their properties sorted as well.
- JSON array data **MUST** also be scanned for the presence of JSON objects (if an object is found, then its properties **MUST** be sorted), but array element order **MUST NOT** be changed.

When a JSON object is about to have its properties sorted, the following measures **MUST** be adhered to:

- The sorting process is applied to property name strings in their "raw" (unescaped) form. That is, a newline character is treated as U+000A.
- Property name strings to be sorted are formatted as arrays of UTF-16 [UNICODE] code units. The sorting is based on pure value comparisons, where code units are treated as unsigned integers, independent of locale settings.
- Property name strings either have different values at some index that is a valid index for both strings, or their lengths are different, or both. If they have different values at one or more index positions, let k be the smallest such index; then, the string whose value at position k has the smaller value, as determined by using the "<" operator, lexicographically precedes the other string. If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string.

  In plain English, this means that property names are sorted in ascending order like the following:

  ```
          ""
          "a"
          "aa"
          "ab"
  ```

The rationale for basing the sorting algorithm on UTF-16 code units is that it maps directly to the string type in ECMAScript (featured in web browsers and Node.js), Java, and .NET. In addition, JSON only supports escape sequences expressed as UTF-16 code units, making knowledge and handling of such data a necessity anyway. Systems using another internal representation of string data will need to convert JSON property name strings into arrays of UTF-16 code units before sorting. The conversion from UTF-8 or UTF-32 to UTF-16 is defined by the Unicode [UNICODE] standard.

The following JSON test data can be used for verifying the correctness of the sorting scheme in a JCS implementation:


```
{
  "\u20ac": "Euro Sign",
  "\r": "Carriage Return",
  "\ufb33": "Hebrew Letter Dalet With Dagesh",
  "1": "One",
  "\ud83d\ude00": "Emoji: Grinning Face",
  "\u0080": "Control",
  "\u00f6": "Latin Small Letter O With Diaeresis"
}
```


Expected argument order after sorting property strings:

```
"Carriage
        Return"
"One"
"Control"
"Latin Small Letter O With Diaeresis"
"Euro Sign"
"Emoji: Grinning Face"
"Hebrew Letter Dalet With Dagesh"
```

Note: For the purpose of obtaining a deterministic property order, sorting of data encoded in UTF-8 or UTF-32 would also work, but the outcome for JSON data like above would differ and thus be incompatible with this specification. However, in practice, property names are rarely defined outside of 7-bit ASCII, making it possible to sort string data in UTF-8 or UTF-32 format without conversion to UTF-16 and still be compatible with JCS. Whether or not this is a viable option depends on the environment JCS is used in.

3.2.4. UTF-8 Generation

Finally, in order to create a platform-independent representation, the result of the preceding step **MUST** be encoded in UTF-8.

Applied to the sample in Section 3.2.3, this should yield the following bytes, here shown in hexadecimal notation:


```
7b 22 6c 69
        74 65 72 61 6c 73 22 3a 5b 6e 75 6c 6c 2c 74 72
75 65 2c 66 61 6c 73 65 5d 2c 22 6e 75 6d 62 65 72 73 22 3a
5b 33 33 33 33 33 33 33 33 33 2e 33 33 33 33 33 33 33 2c 31
65 2b 33 30 2c 34 2e 35 2c 30 2e 30 30 32 2c 31 65 2d 32 37
5d 2c 22 73 74 72 69 6e 67 22 3a 22 e2 82 ac 24 5c 75 30 30
30 66 5c 6e 41 27 42 5c 22 5c 5c 5c 5c 5c 22 2f 22 7d
```


This data is intended to be usable as input to cryptographic methods.

4. IANA Considerations

This document has no IANA actions.

5. Security Considerations

It is crucial to perform sanity checks on input data to avoid overflowing buffers and similar things that could affect the integrity of the system.

When JCS is applied to signature schemes like the one described in Appendix F, applications **MUST** perform the following operations before acting upon received data:

1. Parse the JSON data and verify that it adheres to I-JSON.
2. Verify the data for correctness according to the conventions defined by the ecosystem where it is to be used. This also includes locating the property holding the signature data.
3. Verify the signature.

If any of these steps fail, the operation in progress **MUST** be aborted.

6. References

6.1. Normative References

[ECMA-262] ECMA International, "ECMAScript 2019 Language Specification", Standard ECMA-262 10th Edition, June 2019, <https://www.ecma-international.org/ecma-262/10.0/index.html>.

[IEEE754] IEEE, "IEEE Standard for Floating-Point Arithmetic", IEEE 754-2019, DOI 10.1109/IEEESTD.2019.8766229, <https://ieeexplore.ieee.org/document/8766229>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[UCNORM] The Unicode Consortium, "Unicode Normalization Forms", <https://www.unicode.org/reports/tr15/>.

[UNICODE] The Unicode Consortium, "The Unicode Standard", <https://www.unicode.org/versions/latest/>.

6.2. Informative References

[JSONCOMP] Rundgren, A., ""Comparable" JSON (JSONCOMP)", Work in Progress, Internet-Draft, draft-rundgren-comparable-json-04, 13 February 2019, <https://tools.ietf.org/html/draft-rundgren-comparable-json-04>.

[KEYBASE] Keybase, "Canonical Packings for JSON and Msgpack", <https://keybase.io/docs/api/1.0/canonical_packings>.

[NODEJS] OpenJS Foundation, "Node.js", <https://nodejs.org>.

[OPENAPI] OpenAPI Initiative, "The OpenAPI Specification: a broadly adopted industry standard for describing modern APIs", <https://www.openapis.org/>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[RYU] "Ryu floating point number serializing algorithm", commit 27d3c55, May 2020, <https://github.com/ulfjack/ryu>.

[V8] Google LLC, "What is V8?", <https://v8.dev/>.

[XMLDSIG] W3C, "XML Signature Syntax and Processing Version 1.1", W3C Recommendation, April 2013, <https://www.w3.org/TR/xmldsig-core1/>.

Appendix A. ECMAScript Sample Canonicalizer

Below is an example of a JCS canonicalizer for usage with ECMAScript-based systems:

```
////////////////////////////////////////////////////////////
// Since the primary purpose of this code is highlighting //
// the core of the JCS algorithm, error handling and      //
// UTF-8 generation were not implemented.                 //
////////////////////////////////////////////////////////////
var canonicalize = function(object) {

    var buffer = '';
    serialize(object);
    return buffer;

    function serialize(object) {
        if (object === null || typeof object !== 'object' ||
            object.toJSON != null) {
            ///////////////////////////////////////////////
            // Primitive type or toJSON, use "JSON"      //
            ///////////////////////////////////////////////
            buffer += JSON.stringify(object);

        } else if (Array.isArray(object)) {
            ///////////////////////////////////////////////
            // Array - Maintain element order            //
            ///////////////////////////////////////////////
            buffer += '[';
            let next = false;
            object.forEach((element) => {
                if (next) {
                    buffer += ',';
                }
                next = true;
                /////////////////////////////////////////
                // Array element - Recursive expansion //
                /////////////////////////////////////////
                serialize(element);
            });
            buffer += ']';

        } else {
            ///////////////////////////////////////////////
            // Object - Sort properties before serializing //
            ///////////////////////////////////////////////
            buffer += '{';
            let next = false;
            Object.keys(object).sort().forEach((property) => {
                if (next) {
                    buffer += ',';
                }
                next = true;
                ///////////////////////////////////////////
                // Property names are strings, use "JSON"  //
                ///////////////////////////////////////////
                buffer += JSON.stringify(property);
                buffer += ':';
                //////////////////////////////////////////
                // Property value - Recursive expansion //
                //////////////////////////////////////////
                serialize(object[property]);
```

```
                });
                buffer += '}';
            }
        }
    };
```

Appendix B. Number Serialization Samples

The following table holds a set of ECMAScript-compatible number serialization samples, including some edge cases. The column "IEEE 754" refers to the internal ECMAScript representation of the "Number" data type, which is based on the IEEE 754 [IEEE754] standard using 64-bit (double-precision) values, here expressed in hexadecimal.

IEEE 754	JSON Representation	Comment
0000000000000000	0	Zero
8000000000000000	0	Minus zero
0000000000000001	5e-324	Min pos number
8000000000000001	-5e-324	Min neg number
7fefffffffffffff	1.7976931348623157e+308	Max pos number
ffefffffffffffff	-1.7976931348623157e+308	Max neg number
4340000000000000	9007199254740992	Max pos int (1)
c340000000000000	-9007199254740992	Max neg int (1)
4430000000000000	295147905179352830000	~2**68 (2)
7fffffffffffffff		NaN (3)
7ff0000000000000		Infinity (3)
44b52d02c7e14af5	9.999999999999997e+22	
44b52d02c7e14af6	1e+23	
44b52d02c7e14af7	1.0000000000000001e+23	
444b1ae4d6e2ef4e	999999999999999700000	
444b1ae4d6e2ef4f	999999999999999900000	
444b1ae4d6e2ef50	1e+21	

IEEE 754	JSON Representation	Comment
3eb0c6f7a0b5ed8c	9.999999999999997e-7	
3eb0c6f7a0b5ed8d	0.000001	
41b3de4355555553	333333333.3333332	
41b3de4355555554	333333333.33333325	
41b3de4355555555	333333333.3333333	
41b3de4355555556	333333333.3333334	
41b3de4355555557	333333333.33333343	
becbf647612f3696	-0.0000033333333333333333	
43143ff3c1cb0959	1424953923781206.2	Round to even (4)

Table 1: ECMAScript-Compatible JSON Number Serialization Samples

Notes:

(1) For maximum compliance with the ECMAScript "JSON" object, values that are to be interpreted as true integers **SHOULD** be in the range -9007199254740991 to 9007199254740991. However, how numbers are used in applications does not affect the JCS algorithm.

(2) Although a set of specific integers like 2**68 could be regarded as having extended precision, the JCS/ECMAScript number serialization algorithm does not take this into consideration.

(3) Values out of range are not permitted in JSON. See Section 3.2.2.3.

(4) This number is exactly 1424953923781206.25 but will, after the "Note 2" rule mentioned in Section 3.2.2.3, be truncated and rounded to the closest even value.

For a more exhaustive validation of a JCS number serializer, you may test against a file (currently) available in the development portal (see Appendix I) containing a large set of sample values. Another option is running V8 [V8] as a live reference together with a program generating a substantial amount of random IEEE 754 values.

Appendix C. Canonicalized JSON as "Wire Format"

Since the result from the canonicalization process (see Section 3.2.4) is fully valid JSON, it can also be used as "Wire Format". However, this is just an option since cryptographic schemes based on JCS, in most cases, would not depend on that externally supplied JSON data already being canonicalized.

In fact, the ECMAScript standard way of serializing objects using "JSON.stringify()" produces a more "logical" format, where properties are kept in the order they were created or received. The example below shows an address record that could benefit from ECMAScript standard serialization:

```
{
  "name": "John Doe",
  "address": "2000 Sunset Boulevard",
  "city": "Los Angeles",
  "zip": "90001",
  "state": "CA"
}
```

Using canonicalization, the properties above would be output in the order "address", "city", "name", "state", and "zip", which adds fuzziness to the data from a human (developer or technical support) perspective. Canonicalization also converts JSON data into a single line of text, which may be less than ideal for debugging and logging.

Appendix D. Dealing with Big Numbers

There are several issues associated with the JSON number type, here illustrated by the following sample object:

```
{
  "giantNumber": 1.4e+9999,
  "payMeThis": 26000.33,
  "int64Max": 9223372036854775807
}
```

Although the sample above conforms to JSON [RFC8259], applications would normally use different native data types for storing "giantNumber" and "int64Max". In addition, monetary data like "payMeThis" would presumably not rely on floating-point data types due to rounding issues with respect to decimal arithmetic.

The established way of handling this kind of "overloading" of the JSON number type (at least in an extensible manner) is through mapping mechanisms, instructing parsers what to do with different properties based on their name. However, this greatly limits the value of using the JSON number type outside of its original, somewhat constrained JavaScript context. The ECMAScript "JSON" object does not support mappings to the JSON number type either.

Due to the above, numbers that do not have a natural place in the current JSON ecosystem **MUST** be wrapped using the JSON string type. This is close to a de facto standard for open systems. This is also applicable for other data types that do not have direct support in JSON, like "DateTime" objects as described in Appendix E.

Aided by a system using the JSON string type, be it programmatic like

```
var obj = JSON.parse('{"giantNumber": "1.4e+9999"}');
var biggie = new BigNumber(obj.giantNumber);
```

or declarative schemes like OpenAPI [OPENAPI], JCS imposes no limits on applications, including when using ECMAScript.

Appendix E. String Subtype Handling

Due to the limited set of data types featured in JSON, the JSON string type is commonly used for holding subtypes. This can, depending on JSON parsing method, lead to interoperability problems, which **MUST** be dealt with by JCS-compliant applications targeting a wider audience.

Assume you want to parse a JSON object where the schema designer assigned the property "big" for holding a "BigInt" subtype and "time" for holding a "DateTime" subtype, while "val" is supposed to be a JSON number compliant with JCS. The following example shows such an object:

```
{
  "time": "2019-01-28T07:45:10Z",
  "big": "055",
  "val": 3.5
}
```

Parsing of this object can be accomplished by the following ECMAScript statement:

```
var object = JSON.parse(JSON_object_featured_as_a_string);
```

After parsing, the actual data can be extracted, which for subtypes, also involves a conversion step using the result of the parsing process (an ECMAScript object) as input:

```
... = new Date(object.time); // Date object
... = BigInt(object.big);    // Big integer
... = object.val;            // JSON/JS number
```

Note that the "BigInt" data type is currently only natively supported by V8 [V8].

Canonicalization of "object" using the sample code in Appendix A would return the following string:

```
{"big":"055","time":"2019-01-28T07:45:10Z","val":3.5}
```

Although this is (with respect to JCS) technically correct, there is another way of parsing JSON data, which also can be used with ECMAScript as shown below:

```
// "BigInt" requires the following code to become JSON serializable
BigInt.prototype.toJSON = function() {
    return this.toString();
};

// JSON parsing using a "stream"-based method
var object = JSON.parse(JSON_object_featured_as_a_string,
    (k,v) => k == 'time' ? new Date(v) : k == 'big' ? BigInt(v) : v
);
```

If you now apply the canonicalizer in Appendix A to "object", the following string would be generated:

```
{"big":"55","time":"2019-01-28T07:45:10.000Z","val":3.5}
```

In this case, the string arguments for "big" and "time" have changed with respect to the original, presumably making an application depending on JCS fail.

The reason for the deviation is that in stream- and schema-based JSON parsers, the original string argument is typically replaced on the fly by the native subtype that, when serialized, may exhibit a different and platform-dependent pattern.

That is, stream- and schema-based parsing **MUST** treat subtypes as "pure" (immutable) JSON string types and perform the actual conversion to the designated native type in a subsequent step. In modern programming platforms like Go, Java, and C#, this can be achieved with moderate efforts by combining annotations, getters, and setters. Below is an example in C#/Json.NET showing a part of a class that is serializable as a JSON object:

```
// The "pure" string solution uses a local
// string variable for JSON serialization while
// exposing another type to the application
[JsonProperty("amount")]
private string _amount;

[JsonIgnore]
public decimal Amount {
    get { return decimal.Parse(_amount); }
    set { _amount = value.ToString(); }
}
```

In an application, "Amount" can be accessed as any other property while it is actually represented by a quoted string in JSON contexts.

Note: The example above also addresses the constraints on numeric data implied by I-JSON (the C# "decimal" data type has quite different characteristics compared to IEEE 754 double precision).

E.1. Subtypes in Arrays

Since the JSON array construct permits mixing arbitrary JSON data types, custom parsing and serialization code may be required to cope with subtypes anyway.

Appendix F. Implementation Guidelines

The optimal solution is integrating support for JCS directly in JSON serializers (parsers need no changes). That is, canonicalization would just be an additional "mode" for a JSON serializer. However, this is currently not the case. Fortunately, JCS support can be introduced through externally supplied canonicalizer software acting as a post processor to existing JSON serializers. This arrangement also relieves the JCS implementer from having to deal with how underlying data is to be represented in JSON.

The post processor concept enables signature creation schemes like the following:

1. Create the data to be signed.
2. Serialize the data using existing JSON tools.
3. Let the external canonicalizer process the serialized data and return canonicalized result data.
4. Sign the canonicalized data.
5. Add the resulting signature value to the original JSON data through a designated signature property.
6. Serialize the completed (now signed) JSON object using existing JSON tools.

A compatible signature verification scheme would then be as follows:

1. Parse the signed JSON data using existing JSON tools.
2. Read and save the signature value from the designated signature property.
3. Remove the signature property from the parsed JSON object.
4. Serialize the remaining JSON data using existing JSON tools.
5. Let the external canonicalizer process the serialized data and return canonicalized result data.
6. Verify that the canonicalized data matches the saved signature value using the algorithm and key used for creating the signature.

A canonicalizer like above is effectively only a "filter", potentially usable with a multitude of quite different cryptographic schemes.

Using a JSON serializer with integrated JCS support, the serialization performed before the canonicalization step could be eliminated for both processes.

Appendix G. Open-Source Implementations

The following open-source implementations have been verified to be compatible with JCS:

- JavaScript: <https://www.npmjs.com/package/canonicalize>
- Java: <https://github.com/erdtman/java-json-canonicalization>
- Go: <https://github.com/cyberphone/json-canonicalization/tree/master/go>
- .NET/C#: <https://github.com/cyberphone/json-canonicalization/tree/master/dotnet>
- Python: <https://github.com/cyberphone/json-canonicalization/tree/master/python3>

Appendix H. Other JSON Canonicalization Efforts

There are (and have been) other efforts creating "Canonical JSON". Below is a list of URLs to some of them:

- <https://tools.ietf.org/html/draft-staykov-hu-json-canonical-form-00>
- <https://gibson042.github.io/canonicaljson-spec/>
- <http://wiki.laptop.org/go/Canonical_JSON>

The listed efforts all build on text-level JSON-to-JSON transformations. The primary feature of text-level canonicalization is that it can be made neutral to the flavor of JSON used. However, such schemes also imply major changes to the JSON parsing process, which is a likely hurdle for adoption. Albeit at the expense of certain JSON and application constraints, JCS was designed to be compatible with existing JSON tools.

Appendix I. Development Portal

The JCS specification is currently developed at: <https://github.com/cyberphone/ietf-json-canon>.

JCS source code and extensive test data is available at: <https://github.com/cyberphone/json-canonicalization>.

Acknowledgements

Building on ECMAScript number serialization was originally proposed by James Manger. This ultimately led to the adoption of the entire ECMAScript serialization scheme for JSON primitives.

Other people who have contributed with valuable input to this specification include Scott Ananian, Tim Bray, Ben Campbell, Adrian Farell, Richard Gibson, Bron Gondwana, John-Mark Gurney, Mike Jones, John Levine, Mark Miller, Matthew Miller, Mark Nottingham, Mike Samuel, Jim Schaad, Robert Tupelo-Schneck, and Michal Wadas.

For carrying out real-world concept verification, the software and support for number serialization provided by Ulf Adams, Tanner Gooding, and Remy Oudompheng was very helpful.

Authors' Addresses

Anders Rundgren
Independent
Montpellier
France
Email: anders.rundgren.net@gmail.com
URI: https://www.linkedin.com/in/andersrundgren/

Bret Jordan
Broadcom
1320 Ridder Park Drive
San Jose, CA 95131
United States of America
Email: bret.jordan@broadcom.com

Samuel Erdtman
Spotify AB
Birger Jarlsgatan 61, 4tr
SE-113 56 Stockholm
Sweden
Email: erdtman@spotify.com