2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE) | 978-1-6654-2587-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSRE52982.2021.00050

The Behavioral Diversity of Java JSON Libraries

Nicolas Harrand*, Thomas Durieux*, David Broman*, and Benoit Baudry*
*EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden
Email: {harrand, tdurieux, dbro, baudry}@kth.se

***Abstract*—JSON is an essential file and data format in domains that span scientific computing, web APIs or configuration management. Its popularity has motivated significant software development effort to build multiple libraries to process JSON data. Previous studies focus on performance comparison among these libraries and lack a software engineering perspective.**

We present the first systematic analysis and comparison of the input / output behavior of 20 JSON libraries, in a single software ecosystem: Java/Maven. We assess behavior diversity by running each library against a curated set of 473 JSON files, including both well-formed and ill-formed files. The main design differences, which influence the behavior of the libraries, relate to the choice of data structure to represent JSON objects and to the encoding of numbers. We observe a remarkable behavioral diversity with ill-formed files, or corner cases such as large numbers or duplicate data. Our unique behavioral assessment of JSON libraries paves the way for a robust processing of ill-formed files, through a multi-version architecture.

***Index Terms*—JSON, Java, Behavioral Diversity**

I. Introduction

JavaScript Object Notation, or JSON, is a ubiquitous file and data exchange format. It is used in domains that span web APIs [1], scientific computing [2], data management [3], or configuration management [4]. Despite the importance of JSON in software applications of all kinds, very few works analyze the software engineering aspects of the JSON ecosystem [5], [6] Previous research works about JSON focus on data representations through schema definitions and inference [7], [8], and efficient algorithms for processing JSON files [9], [10]. Meanwhile, the massive adoption of JSON has motivated important software development efforts, leading to the release and maintenance of many libraries to process JSON files in different languages. Our work focuses on a systematic comparison of the input/output behavior of independent JSON libraries, in Java.

Behavioral diversity can be harnessed for improving reliability [11], [12], performance [13], or testing [14], [15]. All these techniques require one foundation: a sound assessment of the existing diversity that can be exploited. For example, Koopman and DeVale quantified the diversity of failure modes among Posix implementations [16] and Srivastava and colleagues assessed the diversity of vulnerabilities among implementations of the Java Class Library [17].

Our work proposes the first assessment of the behavior diversity among Java JSON libraries. The focus on one single programming language allows for a precise behavior analysis based on the language's semantics. We systematically analyze 20 JSON libraries. This set includes a variety of implementations, from the popular `jackson` and `gson` libraries, to more uncommon ones such as `cookjson` or `sojo`. Our analysis of these libraries consists of two steps. First, we determine how each development team decides to represent the 6 JSON types defined in the RFC 8259 standard [18]. Second, we analyze the input/output behavior of the libraries. For this, we curate a novel, diverse set of 473 JSON files to be processed by each library. This set includes 206 well-formed and 267 ill-formed JSON files. We run each library with all the files, and we assess whether each library has a behavior `Conforming` to the RFC 8259 standard. We analyze the behavior of each separate library, as well as to what extent the libraries collectively behave the same for a portion of the files. The latter analysis is what we call behavior diversity: for what parts of the inputs the libraries behave the same or differently.

We observe significant variations in the representation of JSON types. Two of these design decisions have a major impact on the libraries' behavior: the choice of an ordered or unordered data structure to represent the JSON objects, and the representation of numbers. The behavioral analysis reveals that the libraries behave globally well when processing the well-formed files: 17 out of 20 libraries behave `Conforming` to the standard in more than 80% of the cases. The behavior variations when processing well-formed files relate to corner cases such as very big numbers or duplicate keys. The corpus of ill-formed JSON files reveals a significantly larger behavior diversity. A majority of the libraries exhibit a non `Conforming` behavior for more than 20% of the inputs, and the libraries behave the same only for 0.75% of the input files. Yet, when considering the whole set of libraries, 99.3% of the files are processed correctly by one library at least. This suggests the opportunity for a resilient multi-version architecture [11], [19] for JSON.

The main contributions of this work are as follows

- The first systematic analysis of the input/output behavior of 20 independent Java JSON libraries.
- An exhaustive cartography of the different representations of JSON types in Java.
- Empirical evidence of significant behavior diversity among the JSON libraries when they process ill-formed files.

II. Background about JSON

This section introduces the JSON format, as well as the JSON specifications. We hint at several factors that lay the foundations for diverse implementations.

2332-6549/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSRE52982.2021.00050

A. The JSON format

JSON is a file and data exchange format. It provides a textual representation of data that is readable by humans and machines. The JSON format proposes 6 different types: 2 composite types and 4 primitive ones. Primitive types are `Strings`, `Numbers`, `Booleans` and `null`. The composite types are `Object` types that map `String` keys to values of any type, and an `Array` type that is an ordered collection of elements of any type. This allows for arbitrary depth of nested data. Listing 1 shows an example of a JSON document. It is composed of an object with four keys, `"awardYear"` and `"prizeAmount"` are associated with numbers, `"category"` is associated with a nested object, and `"laureates"` with an array.

```
{"awardYear": 1901,
 "category":{"en":"Chemistry","no":"Kjemi","se":"Kemi"},
 "prizeAmount": 150782,
 "laureates":[{
      "knownName":{"en":"Jacobus H. van 't Hoff"}}]}
```

Listing 1: Excerpt from a JSON file returned by nobelprize.org when searching for the list of laureates.

B. JSON specifications

We introduce three root causes that support diverse behaviors for JSON libraries: the evolution of JSON specifications over time; the ambiguities and explicit flexibility of the specifications; design and implementation choices.

Four consecutive IETF Requests for Comments (RFCs) specify the JSON format: RFC 4627 [20] in 2006, RFC 7158 [21] in 2013, RFC 7159 [22] in 2014, and RFC 8259 [18] in 2017. It is important to note that the RFCs were released after JSON had been used for several years (since the early 2000s), as an attempt to standardize existing usages. In this work, we rely on the most recent specification, RFC 8259 [18], to define what constitutes a valid JSON text or not. Meanwhile, the development of several of the libraries we study started before the publication of this latest RFC. Some of them have been updated since 2017.

The different RFC versions refine the specification of the JSON format, making it less and less ambiguous. For example, the first RFC (4627) stipulates that *"octal and hex forms are not allowed"*, implicitly allowing other forms such as decimal and binary. The later RFCs refine this, explicitly stating that numbers are only *"represented in base 10"*. There is one major exception in the refinement process of JSON specifications. RFC 7158 [21] introduces an evolution that makes previously invalid JSON documents valid. It stipulates that a JSON text *"can be any JSON value, removing the constraint that it be an object or array"*. These changes in the specification illustrate how libraries developed at different times may differ in what they consider as valid inputs.

JSON RFCs explicitly leave room for library developers to choose what their parser accepts. RFC 8259 states that a *"JSON parser MUST accept all texts that conform to the JSON grammar"* but a *"JSON parser MAY accept non-JSON forms or extensions"*. On the other hand, JSON libraries that serialize objects into text *"MUST strictly conform to the JSON grammar"* to respect the specification. *"The names within an object SHOULD be unique"*.

C. Design choices for JSON libraries

In this paper, we analyze the behavior of JSON libraries, in Java. These libraries all expose functions to parse JSON files into JSON objects, e.g., to process the data or to store it in a database, as well as functions to serialize JSON objects into a JSON file, e.g., to exchange data between different services.

Library developers are free to choose how they represent the 6 JSON types in Java. The `Object` JSON type is a composite type that includes `(key,value)` pairs, which can be represented with any kind of Map or an ad-hoc data structure to represent the pairs. The `Array` JSON type is also composite and can be represented with any kind of list or array. The `String` JSON type can be represented directly with the primitive Java String type, or with ad-hoc type that encapsulates a String. JSON `Numbers` can be represented with a String or any of the number types in Java (e.g., `float`, `long`, etc.). The JSON `Boolean` type can be mapped on to an `Enum`, a `bool` or even a `String`. JSON's `Null` can be encoded directly in the Java `null` or represented with an `Enum`, or a `String`.

III. Experimental Protocol

This section presents our research questions, the set of JSON libraries we study, as well as the corpora of files we use to assess their behavior. Then, we detail the protocol to answer each question.

A. Research Questions

RQ1. To what extent do JSON libraries implement different design choices?

This question investigates how each library represents the JSON types with Java types. These choices represent a source of design diversity that can impact the behavior of the libraries.

RQ2. How does each library behave on well-formed input JSON files?

RFC 8259 specifies that *"A JSON parser MUST accept all texts that conform to the JSON grammar"* [18]. With RQ2, we observe how each library addresses this point of the specification and handles well-formed JSON files.

RQ3. How does each library behave on ill-formed input JSON files?

RFC 8259 specifies that *"A JSON parser MAY accept non-JSON forms or extensions"* [18]. In RQ3, we investigate to which extent the developers of JSON libraries take advantage of the specification's ambiguity to process ill-formed files.

RQ4. Do the 20 JSON libraries collectively behave differently on both well-formed and ill-formed JSON files?

This question compares the behavior of our JSON libraries and quantifies the diversity of software behavior within this set. Here, we hypothesize that the diversity of behaviors is larger for ill-formed files, as each implementation has to take independent decisions about how they handle these cases.

LIBRARY	# COMMITS	# STARS	VERSION	LAST ACTIVITY
cookjson	116	3	1.0.2	Sept 2017
corn	-	-	1.0.8	Feb 2014
fastjson	3793	1.4k	1.2.75	Nov 2020
flexjson	-	-	3.3	Oct 2014
genson	395	193	1.6	Dec 2019
gson	1485	18.8k	2.8.5	May 2020
jackson	7382	2.7k	2.12.0-rc2	Nov 2020
jjson	216	12	0.1.7	Jul 2016
johnzon	780	-	1.1.8	Nov 2020
json	841	3.7k	20201115	Nov 2020
json-argo	-	-	5.13	Nov 2020
json-io	1040	268	4.12.0	Oct 2013
json-lib	-	-	3.0.1	Dec 2010
json-simple	30	594	1.1.1	Jul 2014
json-util	464	48	1.10.4-java7	Oct 2016
jsonij	348	-	0.3.1	Feb 2020
jsonp	530	75	2.0.0	Nov 2018
mjson	79	67	1.4.0	May 2019
progbase	-	-	0.4.0	Nov 2019
sojo	-	-	1.0.13	Feb 2019

TABLE I: Description of the 20 Java JSON libraries under study. A '-' indicates a library that has no public code repository.

B. JSON libraries

To build our collection of independent Java libraries that implement JSON processing capabilities, we start from https://json.org. This is the official JSON website, setup by the lead author of JSON, Douglas Crockford. We visited the site on November 2020, we found a list of 22 Java libraries. Our goal is to systematically analyze all these libraries, that have been curated by a third-party, authoritative computer scientist. We ignore 3 of them (`StringTree`, `Json-taglib`, `Fossnova-json`), which we cannot build. We compare this dataset to Maven Central [23], which naturally hosts a diversity of JSON libraries [24]. We find that only `jackson` is among the 20 most popular Java JSON libraries and is not in our dataset. Consequently, we add `jackson` to the dataset for our study. Our set of libraries includes the latest version of each library, available on Maven Central on November 24th 2020. This constitutes our dataset of 20 JSON libraries.

Table I describes the libraries in our dataset. The libraries are presented in alphabetical order. When a version control system is available, we collect the number of commits. When the library is associated with a GitHub repository, we note the number of stars. Column VERSION contains the latest version available in Maven Central on November 24th 2020. Column LAST ACTIVITY gives the date of the last commit if available, or the publication date of the artifact on Maven Central otherwise.

C. JSON corpora

To assess the behavior diversity of JSON libraries, we execute all of them against a collection of 492 JSON files. We aggregate 4 JSON corpora that were previously assembled to benchmark JSON libraries. By doing so, we gather JSON files from diverse sources in order to offer a broad coverage of the JSON format.:

- The official test suite of json.org [25] that is meant to evaluate the compliance of a JSON parser to the JSON grammar. It includes 36 JSON files labelled as `pass` or `fail`.
- The Native JSON Benchmark [26] is used to evaluate the performance of native C/C++ JSON libraries as well as compliance to RFC 7159.
- The suite used for "Parsing JSON is a minefield" [27] a study of the challenges and corner cases that a developer may encounter while implementing a JSON library. It includes 318 JSON files labeled as `y` for yes, `n` for no, and `i` for syntactically correct files that RFC 8259 mentions as potentially problematic.
- The test suite of jansson [28] that includes 114 JSON files labeled as `invalid` or `valid`. We include this test suite because jansson is a popular, open source library, from a different ecosystem (C language).

SOURCE	# WELL-FORMED	# ILL-FORMED	SIZE
json.org	3	33	2.7kB
Native JSON	33	33	4.6MB
minefield	130	188	354kB
jansson	46	68	101.8kB
Corpora	206	267	5.1MB

TABLE II: Description of the corpora of JSON files used as input to assess the diversity of behavior of JSON libraries

We collect a total of 534 JSON files. We remove 42 duplicated files, as well as 19 files that can not be read by Java CharsetDecoder class with UTF-8 or UTF-16 encoding. Table II summarizes the origin of the JSON files as well as the content of our corpora (last line of the table). These files, of various sizes, include all types of JSON data, including nested data, large Strings and Numbers, or simply very large files, e.g., the Native JSON test suite includes 3 files bigger than 1MB. The distinction between `Well-formed` and `Ill-formed` files is based on the classification established by the authors of the original datasets. The well-formed JSON corpus includes 206 files that are syntactically correct, according to the JSON grammar specified in RFC 8259 [18]. The ill-formed JSON corpus includes 267 files that include some structural errors. These corpora are available online [29].

D. Protocol for RQ1

RQ1 explores how the developers of the libraries implement the standard JSON types in Java. The RFC 8259 JSON standard [18] describes 6 types for JSON: `Object`, `Array`, `String`, `Number`, `Boolean` and `Null`. By contrast, the Java standard library provides many types that can be used to represent JSON types.

To identify the design decisions implemented in each library, we manually explore their source code to find the classes that represent JSON data. In particular, we note if Java types are used directly to represent JSON values, are extended, or wrapped in a class provided by the library.

To analyze the implementation of JSON Numbers, we execute the libraries to parse JSON numeric values that correspond to extreme values of Java types (for instance, values -2147483648 and 2147483647 for 32-bit integers (`int`) and values $4.9E{-}324$, $2.2250738585072014E{-}308$ and

```
Inputs:
- jsonInput: A well-formed JSON file,
- library: A JSON library
Result: [Conforming, Silent, Error ]
1  try:
2      jsonObject ← parse(library, jsonInput)
3      if jsonObject = NULL ∧ jsonInput ≠ "null" then
4          log(Null_Object)
5          return Error
6  catch Exception:
7      if isChecked(Exception) then
8          log(Parse_Exception)
9      else
10         log(Crash)
11     return Error
12 try:
13     jsonOut ← serialize(library, jsonObject)
14     if jsonOut = jsonInput then
15         log(Equal)
16         return Conforming
17     if jsonObject ≡ parse(library, jsonOut) then
18         log(Equivalent_Object)
19         return Conforming
20     else
21         log(Non_Equivalent_Object)
22         return Silent
23 catch Exception:
24     if isChecked(Exception) then
25         log(Print_Exception)
26     else
27         log(Crash)
28     return Error
```

Algorithm 1: Test sequence to assess the behavior of JSON libraries with `Well-formed` JSON files.

$1.7976931348623157E308$ for 64-bit floating-point numbers (`double`)). We collect the Java objects that are created at runtime. This allows us to determine the diverse Java types used by the library to represent JSON Numbers. The complete list of tested values is available in the reproduction package [30].

E. Protocol for RQ2

RQ2 assesses the behavior of each of the 20 JSON libraries in our dataset on `Well-formed` JSON files. The protocol consists in executing each library, passing every file in the `Well-formed` corpus as input. We categorize the outcome of each execution as `Conforming` to the standard, `Error` or `Silent`. We consider a library as `Conforming` when it correctly parses and serializes a JSON file that is `Well-formed` according to RFC 8259 [18]. An `Error` behavior is when the library explicitly notifies an issue, e.g., with an exception, while a `Silent` behavior indicates that the library does not explicitly notify an issue.

Algorithm 1 defines the sequence of operations we execute with each library. It takes a `Well-formed` JSON file and a library as input, and returns one of the alternatives `Conforming`, `Error`, and `Silent`. The library parses a JSON file into a JSON object (step 1, line 2), then it serializes the object back into a file (step 2, line 13). If this file is strictly equal to the input file, the behavior is `Conforming`, otherwise, we parse the second file back into a JSON object (step 3, line 17). If the two objects produced after both parsing are equivalent, the behavior is `Conforming` to the specification. Any other exceptional behavior crash is an `Error`. If the objects at step 1 and step 3 are not equivalent and the library does not notify it, this is a `Silent` behavior.

```
Inputs:
- jsonInput: An ill-formed JSON file,
- library: A JSON library
Result: [Conforming, Silent, Error ]
1  try:
2      jsonObject ← parse(library, jsonInput)
3      if jsonObject = NULL then
4          log(Null_Object)
5          return Conforming
6      else
7          log(Unexpected_Object)
8          return Silent
9  catch Exception:
10     if isChecked(Exception) then
11         log(Parse_Exception)
12         return Conforming
13     else
14         log(Crash)
15         return Error
```

Algorithm 2: Test sequence to assess the behavior of JSON libraries with `Ill-formed` JSON files.

For each execution we log intermediate behavior: Equal (EQ) when the input file and the file produced at step 2 are strictly equal (not case-sensitive) (Line 14); Equivalent_Object (EV) when the Java objects retrieved at step 1 and at step 3 are equivalent; Non_Equivalent (NE) when the two objects are not equivalent (Line 21); Null_Object (NO) in step 1 (Line 4), when parsing produces a `null` object, which is not a representation of the a `null` JSON value. The execution of algorithm 1 can be interrupted by exceptions. The algorithm distinguishes between checked exceptions, that have been anticipated by the developers, from unchecked exceptions that lead to a crash. We observe 2 types of checked exceptions, in Line 8 (Parse_Exception (PA) in step 1), and in line 25 (Print_Exception (PR) in step 2). A Crash (CR) can occur in lines 10 and 27.

At step 3 on line 17, we check the equivalence between two objects according to the following rules: JSON arrays contain only equivalent elements in the same order, JSON objects include the same set of keys, and for each key, an equivalent object, strings are strictly equal, numbers are equal and of the same type, literals are equal.

F. Protocol for RQ3

RQ3 assesses the behavior of each library with `Ill-formed` JSON files. Each library tries to parse each `Ill-formed` file. This operation can result in 3 different behaviors, as described in algorithm 2. The library is `Conforming` to the standard if it recognizes the input file as `Ill-formed` and explicitly notifies so. This manifests as a Null_Object (NO) or a fail with an explicit Parse_Exception (PA). A library behaves `Silent` if it accepts to parse the `Ill-formed` file and generates an Unexpected_Object (UO), without an explicit notification (line 7). If the library crashes (CA), we classify this as an `Error` behavior.

G. Protocol for RQ4

In this research question, we investigate for which JSON files the libraries behave the same or have diverse behaviors. We make the hypothesis that the diversity of behaviors among

JSON Java libraries is greater when processing `Ill-formed` inputs rather than `Well-formed` inputs.

First, we assess the behavioral diversity pairs of libraries with the behavioral distance defined in Metric 1.

Metric 1: ***Behavioral distance***. We adapt Jaccard's distance to determine the behavioral diversity between two libraries that execute with the same set of input files. Given C, a corpus of input files, two libraries l_1 and l_2, the behavioral distance $bd_C(l_1, l_2)$ between the two libraries is the probability that the two libraries behave differently on an input file picked in C:

$$bd_C(l_1, l_2) = \frac{|\{f \in C | outcome_{l_1}(f) \neq outcome_{l_2}(f)\}|}{|C|}$$

Second, we assess the global diversity among all 20 libraries. For this, we analyze the proportion of files for which a part of the libraries behaves the same.

IV. Results

In this section, we describe and discuss the findings after performing the experiments according to the protocols and research questions described in Section III.

A. *RQ1. To what extent do JSON libraries implement different design choices?*

In this research question, we study how the 20 JSON libraries of this study map the different JSON type to Java types. Table III summarizes these design decisions. Each column in the table represents one JSON type, and their content indicates how developers have chosen to represent them. We mark `C Type` (Contains) the cases when a library defines its own type that delegates calls to the after mentioned the standard Java `Type`. For example, the `corn` library represents JSON Objects with the class `net.sf.corn.converter.json.JsTypeComplex`. This class stores the key/values pairs in a standard Java `ConcurrentHashMap`, the keys are stored as Strings. We mark `E` (Extends) the cases when a library defines a type that inherits directly from the after-mentioned Java type. For example, `cookjson` represents JSON Objects with the class `org.yuanheng.cookjson.value.CookJsonObject` that directly extends the class `HashMap` from the standard library. We mark `T` the cases when a library defines its own type to represent a JSON type and does not rely on any standard Java type. For example, the `corn` library represents JSON Null with the class `net.sf.corn.converter.json.JsTypeNull`. The last line in Table III indicates the number of different ways to represent a JSON type among our set of libraries.

JSON Objects are always represented, one way or another, with the Map interface from the Java standard library. Yet, the implementation and wrapping vary among libraries. We observe that 10 libraries use an ordered map to store JSON objects key-value pairs, while 7 use a `HashMap` that does not preserve the order of insertion. `fastjson` uses either one of these data structures, depending on an option. Both `JsonP` and `Johnzon` collections' are non-modifiable. The decision of using an ordered map directly impacts whether or not the operation of parsing and serialization leaves a JSON text unchanged syntactically.

JSON arrays are mapped to the List interface, except for `json-io` that relies on primitive `Object` array. 15 out of 20 use an `ArrayList` either directly, by extending the class, or by wrapping it in a container class. `corn` is the only library to use a `CopyOnWriteArrayList`.

JSON numbers are mapped to many different types depending on the library. For example, `sojo` relies on `Long` for integers and `Double` for real numbers. This means that the library cannot represent numbers that are larger than 2^{63}, or values more precise than 2^{-1022}, since those types from the standard library use 64 bits representation. Some libraries use primitive types, e.g., `jackson`, or their boxed version, e.g., `flexjson`. `corn` even stores a textual representation of the JSON numbers in a `String` and lets its clients decide which numeric type to use.

JSON Booleans are represented by the Java Boolean class in 10 libraries. 5 libraries define an enum to represent all three JSON literals (`TRUE`, `FALSE` and `NULL`). 8 libraries represent the JSON value `NULL` with a `null` Java object. All others represent it with a specific class, or enum. Note that a library that does use the Java `null` value to represent the JSON literal `NULL`, cannot use it as a mechanism to communicate a missing key to the library's client.

We observe that the libraries that define their own types, still heavily reuse standard types (i.e., very few cases of `T`). The most popular way of reusing standard types is through containment and delegation (54 occurrences among the 120 choices analyzed in Table III). We also observe that some libraries do extend directly types from the standard library instead. This design decision has an impact on the API that the JSON library exposes: a class that encapsulates a standard type exposes its own public interface, while a class that inherits a standard class also exposes the inherited API.

Overall, there is not a single JSON type that is universally mapped to the same Java type by all 20 libraries. The number of classes implemented in the different libraries to represent JSON types varies from 0 to 13. On one extreme, libraries such as `flexjson`, `genson`, `json-util`, `progbase` and `sojo` do not implement any specific class to model JSON types. Their parsers directly return Java objects from types provided by the standard library. Their JSON generator, also directly accepts Java objects and serializes them to JSON text. On the other end of the spectrum, libraries such as `cookjson`, `johnzon` and `jsonp` or `json-simple` implement specific classes for JSON Objects and Arrays, but represent JSON Strings, Numbers and Booleans as Java boxed type.

The last line of Table III emphasizes this wide diversity of design choices. The 20 libraries exhibit up to 13 different choices to represent Objects and 12 different choices to represent Numbers. Even the choice of String representations, which can be trivial with the standard `java.lang.String`, is subject to different choices. This diversity of design choices

Library	Object • Key	Array	String	Number	Boolean	Null
cookjson	*E* HashMap • String	*E* ArrayList	*C* String	*C* BigDecimal, Long, Double, Integer, byte[]	Enum	Enum
corn	*C* ConcurrentHashMap • String	*C* CopyOn-WriteArrayList	*C* String	*C* String	*C* String	*T*
fastjson	C HashMap/LinkedHashMap • String	C ArrayList	String	Integer, Long, BigInteger, BigDecimal	Boolean	null
flexjson	HashMap • String	ArrayList	String	Long, Double	Boolean	null
genson	HashMap • String	ArrayList	String	Long, Double	Boolean	null
gson	*C* LinkedTreeMap • String	*C* ArrayList	*C* Object	*C* Object	*C* Object	*T*
jackson	*C* LinkedHashMap • String	*C* ArrayList	*C* String	*C* int, long, double, float, short, BigDecimal, BigInteger	*C* boolean	*T*
jjson	*C* HashMap • String	*C* ArrayList	*C* StringBuffer	*C* String	*C* boolean	T
johnzon	C UnmodifiableMap *E* AbstractMap • String	*C* List *E* AbstractList	*C* String	*C* BigDecimal, double, long	*C* Enum	*C* Enum
json	*C* HashMap • String	*C* ArrayList	String	Integer, BigDecimal	Boolean	*T*
json-argo	*C* LinkedHashMap • JsonStringNode	*C* List	*C* String	*C* String	*C* Enum	*C* Enum
json-io	*E* LinkedHashMap • Object	Object[]	String	Long, Double	Boolean	null
json-lib	*C* ListOrderedMap • String	*C* ArrayList	String	Double, Integer	Boolean	*T*
json-util	LinkedHashMap • String	ArrayList	String	Long, Double	Boolean	null
jsonij	*C* LinkedHashMap • String	*C* ArrayList	*C* String	*C* double, long, Number	Enum	Enum
jsonp	*C* UnmodifiableMap *E* AbstractMap • String	*C* List	*C* String	*C* int, long, BigDecimal	*C* Enum	*C* Enum
json-simple	*E* HashMap • Object	*E* ArrayList	String	Long, Double	Boolean	null
mjson	*C* HashMap • String	*C* ArrayList	*C* String	*C* Number	*C* boolean	*T*
progbase	HashMap • String	ArrayList	String	Double, Integer	Boolean	null
sojo	LinkedHashMap • String	ArrayList	String	Long, Double	Boolean	null
Alternatives	13	7	4	12	6	4

TABLE III: A cartography of the different representations of JSON types in Java JSON libraries. We mark with a *C* class that contains another type and delegates calls to it. We mark with *E* a type that extends another. We mark with a *T* the cases where a library defines a new type that does not delegate calls to any type of the standard library.

does impact the behavior of the libraries. In particular, the choice of whether to use an ordered collection or not, as well as the choice of types used to represent JSON numbers directly affects how the library behaves.

Answer to RQ1. The diversity of design decisions among 20 libraries is remarkable, with up to 13 different ways of representing JSON Objects and 12 ways of representing JSON. We note that the choice of an ordered map for objects and the representation of numbers are two key choices that impact the behavior of the libraries, providing diverse trade-offs between performance and usability.

B. *RQ2. How does each library behave on well-formed input JSON files?*

In this research question, we investigate how the different JSON libraries behave when processing `Well-formed` JSON files. We apply the protocol described in Section III-E.

Table IV provides an overview of the outcomes on the `Well-formed` corpus. The first column gives the name of a library. The second column provides the number of files for which a library behavior is `Conforming`: number of `EQUAL` (EQ) outcomes, `EQUIVALENT_OBJECT` (EV) outcomes and the total number of `Conforming` behavior (Tot). The third column shows the number of times a library behaves as `Silent`, i.e. NON_EQUIVALENT (NE) in the case of the well-formed files. The fourth column gives the number of `Error` cases: `NULL_OBJECT`(NO), `PARSE_EXCEPTION` (PA), `PRINT_EXCEPTION` (PR), `CRASH` (CR) and total

Library	`Conforming`			`Silent`	`Error`				
	EQ	EV	Tot	NE	NO	PA	PR	CR	Tot
cookjson	97	91	188 (91.3%)	0 (0%)	-	18	-	-	18 (8.7%)
corn	154	51	205 (99.5%)	0 (0%)	-	1	-	-	1 (0.5%)
fastjson	96	93	189 (91.7%)	4 (1.9%)	-	9	4	-	13 (6.3%)
flex-json	89	101	190 (92.2%)	0 (0%)	-	11	5	-	16 (7.8%)
genson	87	104	191 (92.7%)	7 (3.4%)	8	-	-	-	8 (3.9%)
gson	129	77	206 (100%)	0 (0%)	-	-	-	-	0 (0%)
jackson	98	94	192 (93.2%)	5 (2.4%)	-	9	-	-	9 (4.4%)
jjson	125	66	191 (92.7%)	3 (1.5%)	9	-	-	2	11 (5.3%)
johnzon	92	96	188 (91.3%)	0 (0%)	-	18	-	-	18 (8.7%)
json	112	86	198 (96.1%)	3 (1.5%)	-	5	-	-	5 (2.4%)
json-argo	135	62	197 (95.6%)	0 (0%)	-	9	-	-	9 (4.4%)
json-io	88	80	168 (81.6%)	18 (8.7%)	-	15	5	-	20 (9.7%)
json-lib	95	92	187 (90.8%)	0 (0%)	-	18	-	1	19 (9.2%)
json-simple	88	98	186 (90.3%)	0 (0%)	-	15	5	-	20 (9.7%)
jsonij	96	64	160 (77.7%)	0 (0%)	-	41	5	-	46 (22.3%)
jsonp	106	90	196 (95.1%)	0 (0%)	-	10	-	-	10 (4.9%)
jsonutil	96	55	151 (73.3%)	14 (6.8%)	-	34	7	-	41 (19.9%)
mjson	91	99	190 (92.2%)	0 (0%)	-	11	5	-	16 (7.8%)
progbase	76	90	166 (80.6%)	23 (11.2%)	9	-	2	6	17 (8.3%)
sojo	86	76	162 (78.6%)	0 (0%)	-	42	2	-	44 (21.4%)
Population	184	152	206 (100%)	52 (25.2%)	9	74	17	9	89 (43.2%)

TABLE IV: Observed behavior when running the JSON libraries under study with 206 `Well-formed` files.

number of `Error` (Tot). The last line of Table IV (Population) aggregates results over the whole set of libraries: each column is the number of files in the corpus for which at least one library produces a given outcome. These aggregate observations indicate how the set of libraries behaves as a whole, with respect to well-formed files.

For example, the first row shows that the behavior of `cookjson` is `Conforming` (Tot) for 188 files (91.3%): 97 `EQUAL` (EQ) and 91 `EQUIVALENT_OBJECT` (EV). It also produces an `Error` behavior for 18 files (8.7%), where it triggers a `PARSE_EXCEPTION` (PA).

The libraries are `Conforming` for at least 73.3% of the 206 `Well-formed` JSON files and up to 100% in the case of `gson`. The libraries exhibit an `Error` for a number of `Well-formed` files ranging from 0% for `gson`, up to 22.6% for `jjson`. We observe that 89 files (43.2%) trigger an `Error` for at least one library. The share of files leading to `Silent` outcomes ranges from 0% for 12 libraries, and up to 11.2% for `progbase`. There are 59 files that trigger a `Silent` outcome for at least one library. These results illustrate that all libraries' behaviors are `Conforming` for the vast majority of the `Well-formed` files. Meanwhile, most libraries exhibit `Silent` or `Error` behavior on a few input files. In the following, we discuss the causes of non-`Conforming` behavior.

```
1  [-0]
2  [1E22]
3  [1e+2]
4  ["\u2064"]
5  {"a":null}
```

Listing 2: Syntactically correct JSON. Each line is extracted from a file in the well-formed corpus.

All libraries produce a large amount of `EQUIVALENT` outcomes, rather than only `EQUAL` ones. We find two key reasons for this. First, the standard grammar specification allows several distinct representations for the same value. The examples in Listing 2 illustrate this phenomenon. `0` and `-0` are two different character sequences that represent the same value; `1E22`, `1e22` and `1e+22` represent the same number. The developers of a JSON library need to decide in which way they serialize such values. However, whatever decision they take, it implies that some JSON input cannot be serialized back in the same character string. Second, `Equivalent` outcomes are related to the choice of an ordered or unordered map to represent JSON objects. Both design decisions comply with the standard specification. Yet, the significant diversity of choices we observed in RQ1 manifests into a diversity of behaviors, as shown among the `Conforming` results of Table IV.

Library developers sometimes deliberately decide to not strictly follow the specification. For example, the library `fastjson` does not serialize Map entries for which the value is null, if the user does not select the option `SerializerFeature`. The example in line 5 of Listing 2 is parsed correctly by `fastjson` and is serialized into an empty object, which we classify as a `Silent` behavior. This behavior is intended by the developers of the library.

```
1  [9223372036854775808]
2  [0.4e006699999999999999999999999999999999999999999999
       9999999999999999999999999999999999999999999999999999
       99999999999999999969999999006]
3  {"a":"b","a":"b"}
```

Listing 3: Syntactically correct JSON that the RFC warns against. Each line is extracted from a file of the Well-formed corpus.

The `Well-formed` corpus, is built according to the RFC 8259 [18], which allows for a JSON document to be a lonely value, i.e., a value that is neither a JSON object nor a JSON array (for instance a number). RFC 8259 indicates: *"Note that certain previous specifications of JSON constrained a JSON text to be an object or an array"* [18]. Indeed, `cookjson` does trigger a `PARSE_EXCEPTION` for the 8 files that contain lonely values in the corpus.

The `Well-formed` corpus contains only JSON files that are syntactically correct. Still, RFC 8259 mentions well-formed cases that may lead to interoperability issues among different parsers. The two paragraphs below discuss the specific cases of numbers and duplicated keys, which lead to `Error` and `Silent` outcomes.

"[RFC 8259] *allows implementations to set limits on the range and precision of numbers accepted"* [18]. Listing 3 illustrates such cases. For example, line 1 shows an array that contains an integer too big to be expressed with 64 bits. In Java, the `BigInteger` class can represent this value, and all libraries produce an `EQUAL` outcome for this case. Line 2 contains a value that cannot be represented by a 64 bit floating-point number, hence only 6 libraries do not throw any error. For example, `gson` represents the value with `LazyParsedNumber`. Note that this allows `gson` to correctly re-serialize the value, but a client of the library would still need to process a String representing the value on its own. Libraries that represent numbers with the Java types `double` and `long`, as seen in RQ1, are bound to produce either `Silent` or `Error` outcomes for values that cannot be represented with these types.

RFC 8259 indicates: *"When the names within an object are not unique, the behavior of software that receives such an object is unpredictable"* [18]. Line 3 of Listing 3 illustrates a case with a duplicate key and a duplicate value. When both a key and its value are duplicated, all libraries except `org.json`, `flex-json` and `jsonij` generate an `EQUIVALENT` outcome (the duplicated key disappears in the re-serialized version).

Answer to RQ2. `gson` is the singular case that is `Conforming` for 100% of the `Well-formed` JSON files. All other libraries behave globally well, including 16 libraries that correctly process more than 80% of the files and which behave `Error` with different inputs.

C. *RQ3. How does each library behave on ill-formed input JSON files?*

In this research question, we investigate how the different JSON libraries behave when processing `Ill-formed` JSON files. We apply the protocol described in Section III-F.

In Table V, each row summarizes the behavior of a library that processes the corpus of `Ill-formed` files. For these input files, the behavior of a library is `Conforming` when it explicitly notifies the incorrect syntax of the input, by returning a `PARSE_EXCEPTION` or a `NULL_OBJECT`. A `Silent` behavior is when the library behaves as if the input was syntactically correct. For example, the first row

Library	Conforming PA	NE	Tot	Silent UO	Error CR
cookjson	232	-	232 (86.9%)	35 (13.1%)	0 (0%)
corn	100	-	100 (37.5%)	164 (61.4%)	3 (1.1%)
fastjson	179	2	181 (67.8%)	86 (32.2%)	0 (0%)
flex-json	122	-	122 (45.7%)	142 (53.2%)	3 (1.1%)
genson	160	20	180 (67.4%)	84 (31.5%)	3 (1.1%)
gson	131	-	131 (49.1%)	136 (50.9%)	0 (0%)
jackson	232	-	232 (86.9%)	32 (12%)	3 (1.1%)
jjson	12	24	36 (13.5%)	173 (64.8%)	58 (21.7%)
johnzon	235	-	235 (88%)	29 (10.9%)	3 (1.1%)
json	108	-	108 (40.4%)	156 (58.4%)	3 (1.1%)
json-argo	247	-	247 (92.5%)	20 (7.5%)	0 (0%)
json-io	203	2	205 (76.8%)	59 (22.1%)	3 (1.1%)
json-lib	156	-	156 (58.4%)	111 (41.6%)	0 (0%)
json-simple	202	-	202 (75.7%)	65 (24.3%)	0 (0%)
jsonij	238	-	238 (89.1%)	26 (9.7%)	3 (1.1%)
jsonp	231	-	231 (86.5%)	33 (12.4%)	3 (1.1%)
jsonutil	168	2	170 (63.7%)	94 (35.2%)	3 (1.1%)
mjson	191	-	191 (71.5%)	73 (27.3%)	3 (1.1%)
progbase	-	253	253 (94.8%)	11 (4.1%)	3 (1.1%)
sojo	192	-	192 (71.9%)	72 (27%)	3 (1.1%)
Population	265	253	265 (99.3%)	220 (82.4%)	58 (21.7%)

TABLE V: Observed behavior when running the JSON libraries under study with 267 `Ill-formed` files.

shows that `cookjson` is `Conforming` for 232 out of 267 `Ill-formed` files. This library is `Silent` for 35 files. The last line of Table V, Population, indicates, in each column, the number of files in the corpus for which at least one library produces a given outcome.

The behavior of JSON libraries that process the `Ill-formed` corpus is less clear-cut than the `Well-formed`. The share of `Conforming` behaviors ranges from 7.9% (21 files out of 267) for `jjson` to 94.8% (253 files) for `progbase`. Meanwhile, the share of `Silent` outcome ranges from 4.1% to 61.4%, and the share of `Error` ranges from 0% for 7 libraries up to 28.8% for `jjson`.

Some libraries attempt to build a data structure in a "best effort mode". This yields a `Silent` behavior, which does not obviously convey the sense that the input data is ill-formed. RFC 8259 mentions that a *"JSON parser MAY accept non-JSON forms or extensions"* [18]. Table V shows that all libraries are `Silent` for some files, and 5 libraries exhibit this behavior for more than 50% of the ill-form files. The last line of the table also shows that the 82.4% files trigger a `Silent` behavior for at least one library. This is evidence that for most files of the `Ill-formed` corpus, there is at least one library that behaves as if the file was syntactically correct. On the other hand, for 99.3% `Ill-formed` files, at least one library correctly detects it as such.

```
1  [1,]
2  {"Numbers cannot be hex": 0x14}
3  ["Illegal backslash escape: \x15"]
4  {a:"b"}
```

Listing 4: Ill-formed JSON Strings. Each line is extracted from a file of the Ill-formed corpus.

Listing 4 provides examples from the `Ill-formed` corpus. Line 1 is an array with a trailing comma. This input is interpreted as [1], i.e., an array with a single value, ignoring the extra comma, by 11 libraries. This is a `Silent` behavior, since the behavior does not explicitly acknowledge the error in the input. The behavior of the other 9 libraries is `Conforming` since 8 of them throw an exception and `progbase` returns null. The developers of `json-simple` implemented a test specifying how to handle this case [31], showing that the behavior is intentional.

Lines 2, 3, and 4 of Listing 4 illustrate other ill-formed examples that are alternatively handled in an `Conforming` or `Silent` way. Line 2 is parsed silently by 7 libraries, which interpret the value $0x14$ as 20. The other libraries throw an exception. 11 libraries throw an exception when processing Line 3. The other implementations escape a character that should not be, and return an array containing a String. The example at line 4 shows an object containing a non-ambiguous key without quotes. 8 libraries still accept this input. The `org.json` test suite indicates that such a case should be handled [32]. More examples in the `Ill-formed` corpus trigger different behaviors, such as the acceptance of comments or the flexibility with respect to the representation of numbers.

Answer to RQ3. The behavior of 13 libraries is `Conforming` for less than 80% of the `Ill-formed` files. The libraries implement a `Silent` behavior for a large portion of the files, i.e., they decide to tolerate `Ill-formed` inputs, without any explicit notification. Yet, for 99.3% of these inputs, there is at least one library `Conforming`, indicating that a multi-version JSON system can increase the likelihood of a `Conforming` behavior.

D. *RQ4. Do the 20 JSON libraries collectively behave differently on both well-formed and ill-formed JSON files?*

In this research question, we study the behavioral diversity across all JSON libraries. First, we look at behavioral diversity between pairs of our 20 JSON libraries. Metric 1 (Section III-G) adapts Jaccard's distance to capture the probability that a pair of libraries behaves the same for a corpus of input files. We investigate whether there are significant differences in the average pairwise distance between libraries on the `Well-formed` and the `Ill-formed` corpora.

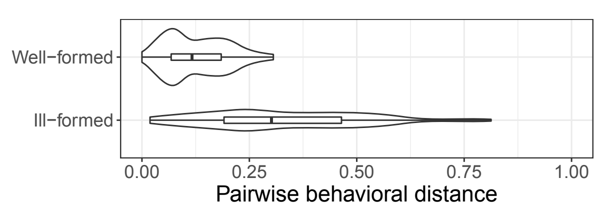

Fig. 1: Distribution of pairwise behavioral distances (cf. Metric 1) among libraries per corpus.

Figure 1 shows the distributions of the pairwise behavioral distances between libraries, for the corpus of `Well-formed` and `Ill-formed` files. On the `Well-formed` corpus, the pairwise distances range from 0 (`cookjson` and `johnzon` behave exactly the same) to 0.31 (`flex-json`

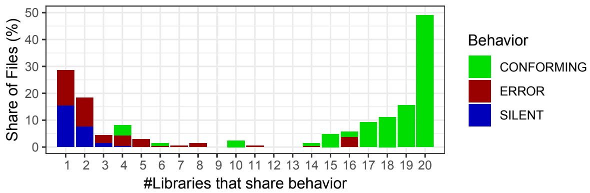

Fig. 2: Distribution of the number of libraries that behave the same with the `Well-formed` corpus

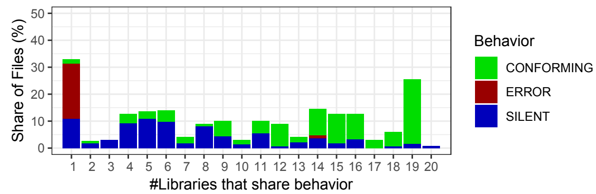

Fig. 3: Distribution of the number of libraries that behave the same with the `Ill-formed` corpus

and `json-util` behave the same for 63 out of 206 files). The median of the distribution is 0.12. For `Ill-formed` files, the pairwise distances range from 0.02 (`jackson` and `johnzon` behave differently on 5/267 files) to 0.81 (`jjson` and `progbase` behave differently on 217/267 files). The median of the distribution is 0.30. Distances and variations are significantly larger on the `Ill-formed` corpus. The average distance between two libraries on the `Well-formed` corpus is 0.13, meaning that on average, two libraries yield a different outcome on 13% of files. The average distance between two libraries on the `Ill-formed` corpus is 0.33. A Welch Two Sample t-test indicates that this difference is significant with a p-value < 0.001. The key observation on Figure 1 is that there is a larger behavioral diversity among JSON libraries when they process the `Ill-formed` corpus.

To consolidate our observations about behavioral diversity, we broaden the analysis to the whole set of 20 libraries, instead of comparing pairs. Figure 2 and Figure 3 show how many libraries exhibit the same behavior when processing an input JSON file.

Each bar in Figure 2 corresponds to the number of libraries that behave the same for a given share of the `Well-formed` files. A bar can have up to 3 subparts depending on the behavior that the libraries share (`Conforming`, `Silent` or `Error`). The x-axis ranges from 1 to 20: values from 1 to 19 correspond to the size of the subsets of libraries that have the same behavior for a file, and 20 shows the share of files that are handled in the same way by all libraries. The y-axis gives the share of files that are handled in the same way by a set of libraries of a given size. The rightmost bar indicates that all 20 libraries have a `Conforming` behavior for 49% of the `Well-formed` files (101 out of 206). This means two things: the libraries behave the same for almost half of the inputs; the consensus is all on `Conforming` behavior. The rest of the files are handled differently by subsets of the libraries. For example *negative-zero.json* (line 1, Listing 2) triggers a `Conforming` behavior for 19 libraries and an `Error` for 1, so it contributes to the bars 19 and 1. The bar on the left indicates that for 15.5% of the files, there is a singular library with a `Silent` behavior, and for 13.1% there is one library with an `Error` behavior. Overall, we observe that 84.9% of the `Well-formed` files trigger the same `Conforming` behavior for at least 17 libraries. The non-`Conforming` behaviors are distributed among small subsets of libraries.

Figure 3 shows the distribution of the number of libraries that behave the same for the `Ill-formed` corpus. The rightmost bar indicates that only 0.75% of the files (2 out of 267) are processed the same way by all 20 libraries and trigger a `Silent` behavior. 19 libraries have the same `Conforming` behavior with 24% of the files (64 out of 267). A key difference with Figure 2 is the spread of the bars: here they are not concentrated on the extreme values but distributed in many sets of all sizes. This indicates a wider diversity of behavior. Looking at `Conforming` behavior, we can compare the 84.9% of files that yield a `Conforming` behavior for at least 17 libraries in the `Well-formed`, while only 35.3% of the files yield such a consensus with the `Ill-formed` corpus. Another interesting difference is the presence of `Conforming` behavior on the left of the plot, indicating that some files are processed correctly only by one or two libraries.

There is more behavioral diversity among libraries that process ill-formed JSON files than when they process `Well-formed` files. This is consistent with the guideline from RFC 8259 [18] which states explicitly that a *"JSON parser MAY accept non-JSON forms or extensions."*. We observe indeed a greater probability of differing outcomes between two libraries on the `Ill-formed` corpus.

Answer to RQ4. All libraries behave exactly the same for 49% of the `Well-formed` files and 0.75% of the `Ill-formed` files. We observe a much wider diversity of behavior among JSON libraries when they process `Ill-formed` files. These results show that a reliable JSON processing can only be achieved through the combination of multiple libraries.

V. Threats to Validity

Internal Validity. The main internal threat to validity lies in the construction of the JSON corpora. These corpora need to cover a wide range of diverse JSON inputs, and the classification between well-formed and ill-formed needs to be as accurate as possible. To limit the risks regarding the diversity of inputs, we gather JSON files from 4 different sources, including the official json.org test suite and a test suite specifically designed to explore the corner cases of the format. To mitigate the ambiguities between ill-formed and well-formed, we manually investigated the JSON files for which a majority of libraries did not yield `Conforming` results.

External Validity. Our study is limited to 20 Java libraries.

Hence, our results might not generalize to other libraries or languages. It is important to note that the root of the observed diversity in the behavior of libraries partly comes from the JSON specification itself, as well as from the difference between JSON types and Java types. We believe that as long as both of these properties hold, it is likely to observe a similar diversity in other ecosystems.

Construct validity. In order to limit threats to construct validity, we use diverse perspectives and we do not rely on one single metric to draw the conclusion of a greater diversity of behavior among libraries on `Ill-formed` inputs. We rely on the range of `Conforming` behavior presented in RQ2 and in RQ3, while for R4 we use a notion of distance and the distribution of the number of libraries that behave the same.

VI. Related Work

Analyzing behavior diversity. Several works study software diversity among multiple software projects providing similar or the same functionalities. Koopman and colleagues [16] propose a comparison of 13 POSIX implementations' behavior. They feed these implementations with a corpus of abnormal inputs parameters and observe the outcome. They observe that when not considering the implementation of the C library, only 3.8% of failures are common to all 13 implementations. Gashi and colleagues [33] examine 4 SQL server implementations and the bugs that affect them. They find that no bug affects all four implementations, and emphasize the opportunity of building a fault tolerant system based on this diversity. Harrand and colleagues [34] study the diversity of 8 Java bytecode decompilers. They observe all decompilers do not fail with the same input files. They propose a meta decompiler that combines the results of different decompilers to build a more reliable one. Carzaniga [35] and Gabel [36] check the input/output behavior redundancy of code snippets with random testing. Our work contributes to this body of knowledge about natural software diversity with novel observations about JSON libraries.

Exploiting software diversity. Software diversity [37] has been exploited in various works for dependability, reliability, testing and security [38]. Muralidharan et al [39] leverage code variants to adapt performance in the context of GPU code. Basios [40] and Shacham and colleagues [13] exploit the diversity of data structure implementations to tailor the selection according to the application that uses a data structure. Sondhi and colleagues [14] leverage similarities between library implementations to reuse test cases from one to test another. Boussaa and colleagues [15] study family of code generators that target different languages from the same sources. They rely on metamorphic testing to automatically detect inconsistencies in these code generators. Srivastava and colleagues [17] compare the multiple implementations of Java libraries to find bugs in the enforcement of security policies. Xu leverages the diversity of computing platforms, focusing on eight factors in OS design and implementation, to build an efficient system to detect malicious documents [41]. Our analysis of JSON libraries behavior sets the foundations for future work that uses multiple implementations for a more resilient management of JSON data.

Analyzing JSON. The small number of existing studies that compare serialization libraries, including JSON, focus on performance. To our knowledge, there is no previous work that compares functional behavior. Maeda [42] compares the performance of twelve Java serialization libraries (XML, JSON and binary). Those libraries exhibit significant performance differences, while all staying in a reasonable range. Similarly, Vanura et al [43] evaluate 49 serialization libraries in diverse languages and propose a benchmark aiming at measuring their performance. In his blog post, *Parsing JSON is a minefield*, Seriot [44] proposes a collection of JSON files to test how JSON parsers handle corner cases of the JSON format. He strongly emphasizes that the specification leaves ambiguity. We integrate Seriot's collection in our corpora. Our work differs from these previous works as we focus on assessing and comparing the input/output behavior of Java JSON libraries.

VII. Conclusion

The JSON format has become increasingly popular in the past 20 years. The popularity of JSON has fueled the development and maintenance of multiple libraries that all provide services to process JSON files. While the format is thoroughly specified in RFC 8259 [18], the specification leaves significant room for choice when implementing a specific library to process JSON. We propose the first systematic analysis of 20 Java JSON libraries. We observe that libraries make significantly different choices of data structures to represent JSON types. Executing the libraries on 473 JSON files, we observe that the diversity of design choices is reflected in the input/output behavior of these libraries. Most of the libraries have a behavior `Conforming` to the standard for `Well-formed` files, including `gson` that processes 100% of the files without errors. Meanwhile, the processing of ill-formed files exhibits a significant diversity of behavior. Only 0.75% of the ill-formed files are recognized as such by all libraries and all libraries exhibit non `Conforming` behavior on some ill-formed files. Yet, when considering the collective behavior of the JSON libraries, up to 99.3% of the files are recognized as ill-formed by one library at least.

The essential role of JSON in distributed systems calls for reliable and secure processing solutions. Our findings open exciting possibilities in terms of software resilience, as developers who build variants of their applications with diverse JSON library implementations could benefit from this natural diversity to mitigate the risks of bugs due to the mishandling of ill-formed JSON data.

VIII. Acknowledgement

This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and by the Trust-Full project funded by the Swedish Foundation for Strategic Research.

REFERENCES

[1] W. Tan, Y. Fan, A. Ghoneim, M. A. Hossain, and S. Dustdar, "From the service-oriented architecture to the web api economy," *IEEE Internet Computing*, vol. 20, no. 4, pp. 64–68, 2016.

[2] K. J. Millman, F. Pérez, V. Stodden, F. Leisch, and R. D. Peng, "Developing open-source scientific practice," *Implementing Reproducible Research*, vol. 149, 2014.

[3] Z. H. Liu, B. C. Hammerschmidt, D. Mcmahon, H. J. Chang, Y. Lu, J. Spiegel, A. C. Sosa, S. Suresh, G. Arora, and V. Arora, "Native JSON datatype support: Maturing SQL and nosql convergence in oracle database," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3059–3071, 2020.

[4] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 351–361, 2016.

[5] G. Barbaglia, S. Murzilli, and S. Cudini, "Definition of REST web services with JSON schema," *Softw. Pract. Exp.*, vol. 47, no. 6, pp. 907–920, 2017.

[6] A. Habib, A. Shinnar, M. Hirzel, and M. Pradel, "Finding data compatibility bugs with JSON subschema checking," in *Proc. of ISSTA*, p. 620–632, 2021.

[7] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON schema," in *Proceedings of the 25th International Conference on World Wide Web*, pp. 263–273, 2016.

[8] P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč, "JSON: data model, query languages and schema specification," in *Proc. of the Symposium on principles of database systems*, pp. 123–135, 2017.

[9] G. Langdale and D. Lemire, "Parsing gigabytes of JSON per second," *VLDB Journal*, vol. 28, no. 6, pp. 941–960, 2019.

[10] M. A. Baazizi, D. Colazzo, G. Ghelli, and C. Sartiani, "Parametric schema inference for massive JSON datasets," *The VLDB Journal*, vol. 28, no. 4, pp. 497–521, 2019.

[11] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, pp. 1491–1501, 1985.

[12] P. T. Popov, V. Stankovic, and L. Strigini, "An empirical study of the effectiveness of "forcing" diversity based on a large population of diverse programs," in *Proc. of ISSRE'12*, pp. 41–50, 2012.

[13] O. Shacham, M. Vechev, and E. Yahav, "Chameleon: adaptive selection of collections," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 408–418, 2009.

[14] D. Sondhi, M. Jobanputra, D. Rani, S. Purandare, S. Sharma, and R. Purandare, "Mining similar methods for test adaptation," *IEEE Transactions on Software Engineering*, 2021.

[15] M. Boussaa, O. Barais, G. Sunyé, and B. Baudry, "Leveraging metamorphic testing to automatically detect inconsistencies in code generator families," *Software Testing, Verification and Reliability*, vol. 30, no. 1, p. e1721, 2020.

[16] P. Koopman and J. DeVale, "Comparing the robustness of posix operating systems," in *Proc. of the Symposium on Fault-Tolerant Computing*, pp. 30–37, IEEE, 1999.

[17] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov, "A security policy oracle: Detecting security holes using multiple api implementations," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 343–354, 2011.

[18] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 8259, RFC Editor, December 2017.

[19] M. Xu, K. Lu, T. Kim, and W. Lee, "Bunshin: Compositing security mechanisms through diversification," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 271–283, USENIX Association, July 2017.

[20] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627, RFC Editor, July 2006.

[21] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7158, RFC Editor, March 2013.

[22] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159, RFC Editor, March 2014.

[23] F. Rodriguez, "Most popular JSON library on Maven Central." https://mvnrepository.com/open-source/json-libraries, 2021. [Online; accessed 1-January-2021].

[24] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The emergence of software diversity in maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 333–343, IEEE, 2019.

[25] D. Crockford, "json.org official test suite." http://www.json.org/JSON_checker/, 2021. [Online; accessed 1-January-2021].

[26] M. Yip, "Native JSON Benchmark." https://github.com/miloyip/nativejson-benchmark, 2021. [Online; accessed 1-January-2021].

[27] N. Seriot, "JSON Parsing Test Suite." https://github.com/nst/JSONTestSuite, 2021. [Online; accessed 1-January-2021].

[28] akheron, "Jansson." https://github.com/akheron/jansson, 2021. [Online; accessed 1-January-2021].

[29] Nicolas Harrand, Thomas Durieux, David Broman, and Benoit Baudry, "JSON corpora." https://github.com/nharrand/json-diversity/tree/main/data/bench, 2021. [Online; accessed 1-January-2021].

[30] Nicolas Harrand, Thomas Durieux, David Broman, and Benoit Baudry, "List of tested values." https://github.com/nharrand/json-diversity/tree/main/test-bench/src/main/resources/types, 2021. [Online; accessed 1-January-2021].

[31] "json-simple permissiveness test." https://github.com/fangyidong/json-simple/blob/a8b94b79/src/test/java/org/json/simple/Test.java/#L49, 2021. [Online; accessed 1-January-2021].

[32] "JSON-java permissiveness test." https://github.com/stleary/JSON-java/blob/6b779f8ca7ed6669d8342a39e3c3d1dc44fd8f8f/src/test/java/org/json/junit/JSONObjectTest.java#L200, 2021. [Online; accessed 1-January-2021].

[33] I. Gashi, P. Popov, and L. Strigini, "Fault diversity among off-the-shelf sql database servers," in *International Conference on Dependable Systems and Networks, 2004*, pp. 389–398, IEEE, 2004.

[34] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java decompiler diversity and its application to meta-decompilation," *Journal of Systems and Software*, vol. 168, p. 110645, 2020.

[35] A. Carzaniga, A. Mattavelli, and M. Pezzè, "Measuring software redundancy," in *Proc. of the International Conference on Software Engineering*, pp. 156–166, IEEE, 2015.

[36] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 147–156, 2010.

[37] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pp. 67–72, IEEE, 1997.

[38] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Survey*, vol. 48, no. 1, pp. 16:1–16:26, 2015.

[39] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai, "Architecture-adaptive code variant tuning," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 325–338, 2016.

[40] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr, "Darwinian data structure selection," in *Proc. of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 118–128, 2018.

[41] M. Xu and T. Kim, "Platpal: Detecting malicious documents with platform diversity," in *Proc. of USENIX Security*, pp. 271–287, USENIX Association, 2017.

[42] K. Maeda, "Performance evaluation of object serialization libraries in xml, json and binary formats," in *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pp. 177–182, 2012.

[43] J. Vanura and P. Kriz, "Perfomance evaluation of java, javascript and php serialization libraries for xml, json and binary formats," in *Services Computing – SCC 2018* (J. E. Ferreira, G. Spanoudakis, Y. Ma, and L.-J. Zhang, eds.), (Cham), pp. 166–175, Springer International Publishing, 2018.

[44] N. Seriot, "Parsing JSON is a Minefield." http://seriot.ch/parsing_json.php, 2018. [Online; accessed 1-September-2020].