Cross-Language Differential Testing of JSON Parsers

Jonas Möller[1,2], Felix Weißberg[1], Lukas Pirch[1], Thorsten Eisenhofer[1], Konrad Rieck[1,2,3]

[1]Technische Universität Berlin
[2]Berlin Institute for the Foundations of Learning and Data (BIFOLD)
[3]Technische Universität Wien

ABSTRACT

JSON is a widely used format for representing data on the Internet. Unfortunately, the format is imprecisely specified, which poses the risk of confusion and ambiguity when processing sensitive data. While previous work has focused on manual analysis of parsers, an automatic analysis of the interplay of multiple parsers resulting from this imprecision has received little attention so far. In this paper, we address this problem and propose a framework for differential testing of JSON parsers tailored towards discovering semantic discrepancies. To spot these differences automatically, we overcome two challenges: First, we introduce a consensus-based normalization of JSON that enables us to analyze data semantics in absence of a precise specification. Second, we propose a novel mechanism for tracking test coverage across runtime environments, so that confusions between parsers written in C, C++, Rust, Java, and Python can be detected simultaneously. In a comparative analysis of 22 JSON parsers, we uncover various semantic discrepancies, ranging from minor inconsistencies in the representation of numbers and strings to severe confusions in the handling of object keys and values. We illustrate the security impact of these discrepancies in different case studies, echoing recent efforts to enforce a stricter specification for JSON in security applications.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Cross-language, Differential testing, JSON specification

ACM Reference Format:
Jonas Möller, Felix Weißberg, Lukas Pirch, Thorsten Eisenhofer, and Konrad Rieck. 2024. Cross-Language Differential Testing of JSON Parsers. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3634737.3657003

1 INTRODUCTION

JavaScript Object Notation, or JSON for short, has become a mainstay for the serialization and transmission of information. While the format has been originally developed for web content only, it has since become one of the most frequently used data exchange formats alongside XML. Due to this widespread use, many programming languages have been extended to include JSON parsers in their standard libraries, such as Java and Python. Similarly, the JSON format has been adopted as a general data representation in modern security protocols, like JWT [21] and OAuth2 [18]. As a result of this development, a plethora of implementations for JSON has emerged, covering different runtime environments and focusing on varying aspects, such as performance, portability, usability or memory requirements.

Unfortunately, the JSON specification lacks precision and is riddled with ambiguity, providing only vague guidance for parser development [25, 32]. Although the format appears lightweight and simple at first glance, implementing parsers that behave identically under all possible inputs is a hard problem. Hence, when passing data across multiple parsers, such as in a distributed service, we cannot ensure that they interpret semantics in the same way. For example, the parsers may disagree on the precision of numbers, the encoding of strings, or even the keys contained in an object. Such confusions are not just a minor annoyance, but pose a serious security problem once JSON is employed in security-critical applications, as discussed by Seriot [32] and Miller [25]. A notable example of such an issue is CVE-2017-12635 [6] which allows remote code execution due to the different interpretation of the same input data by multiple JSON parsers.

Previous security research has tackled this problem by designing methods for localizing defects in individual parser implementations and libraries. While this work has helped reduce the attack surface of JSON, methods for uncovering confusions and discrepancies *between* parsers have not be explored so far. In this paper, we bridge this gap by introducing Crossy, an automated framework designed to identify semantic differences in JSON parsers. In contrast to previous research, our framework allows us to analyze multiple parsers simultaneously and identify discrepancies in their implementations that would not surface in their individual analysis.

Technically, our framework is based on the concept of *differential testing* [24], in which implementations of one specification are exposed to identical inputs and monitored for differences. In particular, we build on coverage-guided differential testing. However, analyzing JSON comes with unique challenges in this setting: First, without a precise specification, it is hard to reason about the semantics implemented by different parsers. To tackle this challenge, we propose a consensus-based normalization, where the interpretation of JSON is determined by the "common sense" of multiple parsers. Second, we take the wide variety of JSON implementations into account and develop a mechanism to track coverage across runtime environments. This allows us to identify discrepancies between parsers written in different programming languages.

Based on our framework Crossy, we conduct a comparative analysis of 22 popular JSON parsers, covering five programming

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07.
https://doi.org/10.1145/3634737.3657003

languages (C, C++, Rust, Java, and Python) and three runtime environments (native, Java interpreter, Python interpreter). We observe that discrepancies are far from rare and that our framework is constantly uncovering new cases where parsers disagree on the interpretation of inputs. By systematizing these findings, we can identify classes of confusions ranging from minor misinterpretations of numbers and strings to erroneous handling of object keys and values. The latter cases in particular pose serious security problems, as we show in corresponding case studies.

Our analysis provides an expanded view on JSON security. While developing a robust parser is a challenge in itself, our investigation shows that the interplay of multiple parsers creates further attack surface that can only be uncovered through differential testing and that is hard to mitigate without a rigid specification. Consequently, we advocate for more stringent guidelines for the JSON format, in line with recent initiatives supporting standards, such as I-JSON [11] and related schemes [31].

In summary, we make the following contributions in this work:

(1) **Differential testing for JSON**. We propose the first framework for differential testing of JSON, suitable for identifying semantic discrepancies across parsers automatically.
(2) **Normalized and cross-language analysis.** Our framework compensates for the lack of precise specification and jointly searches for differences across runtime environments.
(3) **Large-scale analysis of parsers.** We analyze 22 popular JSON parsers and discover inconsistencies in their interpretation of data, ranging from subtle differences to severe security problems.

Roadmap. We introduce our framework Crossy in Section 2 and discuss its implementation in Section 3. Our comparative analysis of JSON parsers is presented in Section 4. We consider limitations and related work in Section 5 and Section 6, respectively. Finally, Section 7 concludes the paper.

2 THE CROSSY FRAMEWORK

The JSON standard defines a simple and intuitive grammar that specifies the form of valid JSON objects. On closer inspection, however, the specification contains many minor ambiguities and degrees of freedom. Even if a JSON parser meticulously adheres to the standard, interoperability problems between parsers can hardly be ruled out. To illustrate this issue, let us investigate a simple JSON object and its potential semantic interpretations.

```
[ 12345678901234567 ] ⟶ { [ 12345678901234567 ]
                        { [ 1.234567890123457e16 ]
                        { [ 4294967296 ]
```

The object contains a large integer number. Depending on the implementation of the parser, this number may be interpreted as given, converted into a floating point number with loss of precision or even truncated to a 32-bit integer value. It is already evident from this simple example that semantic discrepancies between parsers pose a notable risk in security-critical applications.

The goal of our analysis is thus to systematically uncover these ambiguities and determine the "common ground" on which current JSON parsers agree and where they diverge. To this end, we introduce Crossy, a framework that simultaneously analyzes parsers and automatically detects differences in their understanding of identical data. A natural concept for realizing this analysis is *differential testing* [24]. Given two parser implementations p and $\hat{p}$ for a specification, the testing aims to find an input x where the implementations disagree, that is, $p(x) \neq \hat{p}(x)$. In our setting, p and $\hat{p}$ are JSON parsers and we are interested in finding a *valid* JSON object x that results in diverging interpretations.

As the basis for our framework, we build on the differential testing approach by Petsios et al. [29]. The approach generates inputs using a guided fuzzer that maximizes the execution diversity between parsers. For this purpose, the fuzzer tracks the code coverage of inputs across parsers and aims to uncover new combinations of code regions. However, two challenges hinder the application of this approach that need to be overcome for finding differences in JSON parsers:

C1 *Cross-language testing.* JSON parsers are written in several programming languages, such as C, Python, and Java. Uncovering differences between these parsers requires tracking code coverage across runtime environments, which is not possible with existing approaches.

C2 *JSON normalization.* The lack of a precise specification renders it difficult to compare data representations and locate disagreements. To account for this, we normalize JSON using a consensus-based method that distills the "common sense" of all parsers under test.

C3 *Difference analysis.* Not all differences inherently reflect security issues. We therefore introduce a taxonomy derived from the JSON grammar that supports the manual analysis of discrepancies and can be gradually extended to categorize findings of Crossy.

Technically, our framework Crossy consists of two major stages. The fuzzing stage as shown in Figure 1 operates in a loop consisting of four steps: the input generation ❶, a filtering of uninteresting inputs ❷, a consensus-based difference detection ❸ and a logging mechanism for later analysis ❹. It is complemented by the analysis stage shown in Figure 2, which first gathers the outputs of different parsers ❺, from which normalized parse trees are generated ❻. Functional differences are then categorized by the difference location in the respective parse tree ❼.

In the following sections, we introduce the main components of this framework, that is, the cross-language differential testing and the consensus-based normalization. For a general introduction to differential testing we refer to the works of McKeeman [24] and Petsios et al. [29].

2.1 Cross-Language Differential Testing

Due to the increasing prevalence of JSON in practice, corresponding parsers are now available for various programming languages. Hence, semantic discrepancies cannot only occur between parsers but also across different runtime environments of these languages. For example, a server may process a JSON object in the Python interpreter and send it to a web browser, which analyzes it with

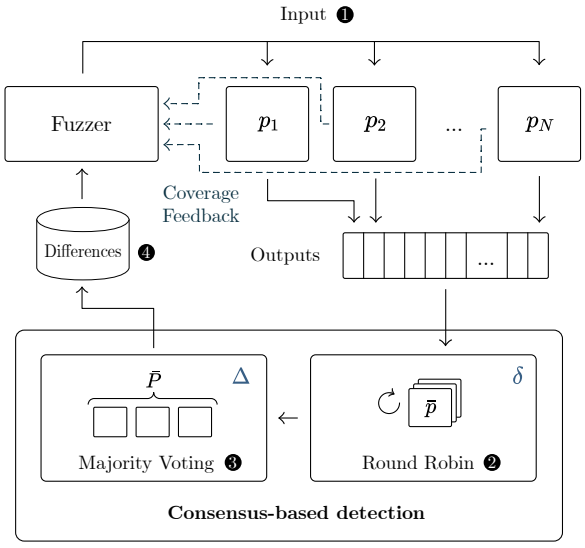

Figure 1: The fuzzing stage. The fuzzer generates inputs and provides them to a set of targets. The outputs are passed to a consensus-based detector to find differences.

a C++ parser. This cross-language transfer is not limited to network applications and can also take place on the same host, for example, when two processes with different runtime environments communicate with each other. This crossing of environments poses a challenge for differential testing, as code coverage in the parsers is language-dependent and thus not directly accessible.

While a simple solution like binary instrumentation [e.g., 2, 30] can be applied to all runtime environments, for interpreted languages we would merely track the states of their virtual machine. The interesting aspects of the execution—the control flow on programming language level—would still remain opaque. Similarly, there exist approaches for tracking coverage on the language level, such as fuzzing frameworks like *Atheris* for Python [1] and Jazzer for Java [3]. Yet, these approaches only monitor a single environment and cannot be generalized to cross-language targets.

As a remedy, we introduce a new method for cross-language tracking of code coverage in Crossy. The core idea is to monitor the visiting of control-flow edges across all runtime environments in a shared memory map. For interpreted languages, the runtime environments are instrumented at the language level so that they do not report coverage of the underlying virtual machines. As a consequence, the guided fuzzer in Crossy can operate a single map of coverage, guiding the construction of new inputs depending on all parsers simultaneously.

2.2 Semantic Normalization

The second challenge in realizing differential testing for JSON is the definition of semantic discrepancies. While for security-critical applications, such as cryptographic libraries, often a diverging exit code already indicates a problem, for JSON we require a more detailed analysis to spot semantic disagreement.

Ideally, we would like to compare the internal representation that each parser extracts from a provided input. However, because each parser uses different data structures for holding this internal state, a direct comparison is technically not feasible. Instead, we take advantage of the fact that most JSON parsers provide functions to serialize their internal representation back into the JSON format. That is, during testing each parser receives a JSON object as input and returns its interpretation as serialization of its internal state. By comparing this output, we can test parsing libraries at a more detailed level than just examining the exit codes.

Formally, we thus model a parser p as a function that maps one JSON object to another,

$$p : \mathcal{J} \mapsto \mathcal{J}, \quad p(x) \to s(r_x)$$

where $\mathcal{J}$ is the domain of valid JSON data, s a serialization function and r_x the internal representation of the parser. Note that we exclude invalid JSON from our analysis, as corresponding defects can be determined with regular testing strategies.

However, a simple serialization of the internal representation is not sufficient because of syntactic variance, such as Unicode equivalence (e.g., `"\u000a"` vs. `"\n"`). In other words, there exist multiple syntactic correct strings that can represent the same semantic information. A further hindrance in interpreting these objects is the prevalence of implementation-defined freedoms (e.g. precision and range of numbers) and ambiguities (e.g. unordered and duplicate keys). For example, the following two JSON objects are syntactically different, yet semantically equivalent:

```
{"a":1, "\u0062":2} ≡ {"b":2, "\u0061":1}
```

Since we only want to recognize semantic differences, we have to normalize serialized JSON objects so that their equivalence is visible to our framework.

Direct normalization. Ideally, there would exist a normalization function that transforms JSON objects into a canonical representation which resolves syntactic differences. As a naive approach, we could use one of the parsers itself as a *normalizer* and apply it to the outputs of all other parsers. However, by committing to only one parser, all defects of this parser would become part of the normalization and could not be detected by our framework. Moreover, a faulty parser could transform a valid input into an invalid one and vice versa. Therefore, direct normalization is a chicken-or-the-egg problem that requires an error-free parser to begin with and hence is practically infeasible.

Consensus-based detection. Instead of relying on a single parser, we thus propose to use an ensemble of parsers to create a "common sense" of JSON semantics. In this setting, errors of any single parser do not matter for the result as the respective parser gets outvoted by the remaining parsers. This is in line with the results by Harrand et al. [19] who found that resilience against JSON parsing errors is improved by a multi-version architectures [9].

More formally, let us consider a set of parsers P, each taking a string x representing valid JSON as input and returning a serialization of its internal state as output. Starting from the naive approach, we can simply select one of the parsers $\bar{p} \in P$ and use it to normalize the output of the other ones. That is, we calculate $y_j = \bar{p}(p_j(x))$ for each $p_j \in P$. If all parsers are semantically equivalent, the normalized outputs y_j are identical, otherwise a discrepancy is detected.

We describe the discovery of such a discrepancy using $\bar{p}$ by

$$\delta(\bar{p}, x) = \begin{cases} 1 & \text{if not all } y_j \text{ are identical} \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

Since committing to a single normalization parser $\bar{p}$ would jeopardize our analysis, however, we instead use a group of multiple parsers $\bar{P} \in \mathcal{P}(P)$ where $\mathcal{P}$ denotes the power set. The decision of the group of parsers for a given input is now given by a majority voting on each of the normalization parsers' decision:

$$\Delta(\bar{P}, x) = \begin{cases} 1 & \frac{1}{|\bar{P}|} \sum_{\bar{p} \in \bar{P}} \delta(\bar{p}, x) > \frac{1}{2} \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

This approach is inspired by fault-tolerant systems, where a group of potentially faulty components can make a robust decision by considering their consensus rather than individual results. Thus, given the imprecise specification of JSON, we distill the "common sense" from $\bar{P}$ instead of trusting each parser alone. While our approach cannot rule out wrong decisions in general, the probability of overlooking a semantic discrepancy decreases with the size and quality of the parsers in the $\bar{P}$.

Round-robin detection. The error tolerance of our consensus-based detection of discrepancies comes at a price: The runtime overhead during fuzzing increases linearly with the size of $\bar{P}$, so that we need to trade-off performance with correctness. Fortunately, we can leverage an observation made during our tests to significantly reduce the overhead of our detection approach.

A semantic difference between two parsers is often triggered by several inputs. For example, diverging interpretations of numbers manifest in a large number of reported discrepancies. Consequently, false negatives are less relevant in our framework as the same error is triggered repeatedly during testing. Hence, we can optimize our detection strategy: For each input, we follow the naive approach of using a single parser for normalization. This parser is selected with the round-robin method from $\bar{P}$ and thus provides only one opinion of the ensemble. If this opinion indicates a potential difference, we confirm this decision with the entire ensemble by a majority vote. Otherwise, we proceed and assume that a missed difference is detected later if another parser is selected for normalization. This simple approach provides three advantages:

(1) Each non-difference input only needs to be processed once, independent of the size of $\bar{P}$.
(2) The round-robin method prevents the system from committing to a single parser for normalization.
(3) All detected discrepancies are confirmed by the ensemble, so that errors only occur if its majority errs.

2.3 Difference Analysis

At the end of the testing run, we receive a set of valid JSON inputs that cause differing behavior in parsers according to our consensus-based detection. In the next step, we aim to further analyze the differences in terms of their root-cause. With this analysis we can determine different behavior of JSON parsers, faulty behavior by individual parsers, and classify differences by their severity. To this end, we compare the outputs of the JSON parsers on a syntactic and semantic level. We create two normalized parse trees for each pair of parsers and recursively compare each element in the trees. To avoid generating a tree with faulty elements, we employ a JSON parser for this task that agrees with the employed parser ensemble and hence interprets the objects as the majority.

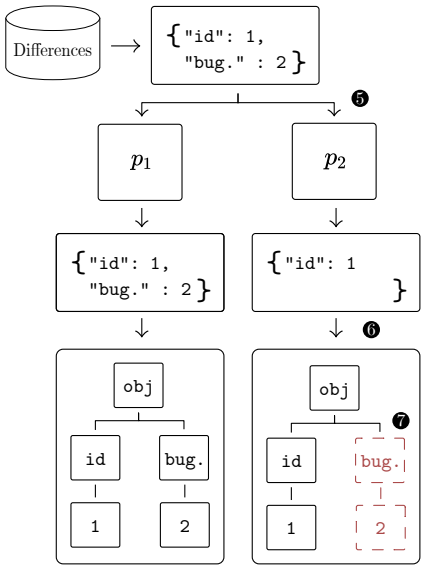

Figure 2: The analysis stage. For each input in our difference corpus, we do a pair-wise comparison by first constructing two parse trees and then traversing the nodes simultaneously.

Parse tree traversal. At its core, the JSON grammar consists of seven JSON values: `object`, `array`, `number`, `string`, `"true"`, `"false"`, and `"null"`. While the last three of these are trivial to compare, the other elements come with their own semantics which need to be incorporated for explaining discovered discrepancies. Hence, during the parse tree traversal, we first identify the element type and then employ the following comparison strategy.

(1) **Objects.** As objects consists of name/value pairs, two objects are equal if the number of name/value pairs match and the set of names/value pairs are equal. This results in a recursive comparison since we need to compare the names using string comparison and the associated values using the respective type comparisons.
(2) **Arrays.** Similarly, two arrays are equal if they contain the same number of values, the values are equal according to their respective type comparison, and the values' order is identical between the arrays.
(3) **Numbers.** In JSON, the number concept includes both integer and floating point representation. While the numbers `1` and `1.0` have the same numeric value, their types could differ in languages like C. Additionally, the JSON specification refrains from defining clear ranges and precisions for numbers and only suggests adherence to IEEE 754 binary64 [7]. To compare two numbers, we examine whether the number is an integer or a floating point number and then compare it based on their respective value.

(4) **Strings.** JSON strings consist of Unicode characters, which as of RFC 8259 [12], have to be encoded in UTF-8 outside of closed systems. To check the equality of two strings, they are compared on a code unit by code unit basis. Unicode characters in the Basic Multilingual Plane (BMP) can additionally be encoded as a six character sequence, the "\u escapement notation". For example, the Unicode character `U+0061` for an `a`, can be encoded as `"\u0061"`. Characters outside of the BMP can be encoded with a 12-character sequence using the UTF-16 surrogate pair. Lastly, some characters can be encoded using a two character encoding, e.g., `"\n"`, `"\t"`.

3 IMPLEMENTATION

We implement Crossy on top of libFuzzer [5]. Our method of collecting cross-language coverage, however, is independent of the underlying fuzzer, as it only requires mechanisms for inter-process communication (IPC) and the availability of coverage information in the target language. In this section, we provide further implementation details of Crossy and discuss important design decisions of our framework. We make our source code publicly available at https://github.com/j-moeller/crossy.

3.1 Cross-Language Coverage

In general, we can distinguish between two types of target languages: *in-process* target languages (i.e., C, C++, and Rust) and *inter-process* target languages (i.e., Python and Java). While the former languages can be compiled and incorporated directly into our main fuzzing process, the latter are based on virtual machines and require a separate process for execution. In our implementation, we communicate with theses processes via pipes for control flow and shared memory for data exchange. The coverage is collected in the main fuzzing process in a bitmap which is provided to the non-native languages via memory-mapped shared memory.

Overall, our cross-language coverage-guidance is target language agnostic: if the respective target can be fuzzed by a "regular" fuzzer at the language level and provides methods for inter-process communication, it can be fuzzed by Crossy.

In-process targets. By using in-process targets, fuzzing performance is improved as IPC and process synchronization mechanisms are avoided. Similar to Petsios et al. [29], we compile each target to shared objects and load them via `dlopen` to prevent name collisions. Upon loading an instrumented shared object, the reserved memory for coverage is registered via callbacks with the libFuzzer part of our implementation.

C/C++. For C/C++, instrumentation is added via LLVM's SanitizerCoverage using the `-fsanitize-coverage` parameter. To reduce the number of instrumented edges for larger libraries, we additionally utilize `-fsanitize-coverage-allowlist` to focus on the JSON modules.

Rust. Rust is compatible with C/C++ applications. It can be compiled with LLVM's `-fsanitize-coverage` parameter as well and also loaded as a shared object. However, the Rust compiler currently does not support targeted instrumentation via an `allowlist` which increases the number of instrumented edges.

Inter-process targets. The inter-process targets are initialized as persistent processes at the start of the fuzzing phase. These set up the communication with the main fuzzing process via pipes and memory-mapped shared memory. During fuzzing, the inter-process communication with the targets follows a simple protocol that aims at minimizing the communication overhead: Each target waits for input to become available from a pipe. After the input has been processed by the parser, the output and coverage information are written to shared memory. Finally, the process signals the end of its execution to the main process via an eight byte write to a pipe. Overall, this results in an IPC overhead of two syscalls (one read, one write) and writes to memory-mapped shared memory.

Python. While Python's C API provides means to start an in-process Python interpreter, we explicitly decided against this solution because the API only supports one interpreter per process. Although a single interpreter could potentially increase the performance, all Python libraries under test would have to share a common state which could, potentially, lead to interference between them. We therefore opted for a multi-process solution with isolated Python interpreters. For the instrumentation, we use a modified Atheris [1] which rewrites Python bytecode to trace edge coverage.

Java. We build a wrapper application using the Java Native Interface that is started from the fuzzing process. The application then initializes the IPC mechanisms to communicate with the fuzzer, launches the Java virtual machine containing the target program, and writes coverage information to shared memory after each run. The instrumentation is based on a modified version of Jazzer [3]. Because of implementation details of the Java instrumentation, we can only give an estimate of instrumented edges to the next largest power of two.

3.2 Measuring δ-Diversity

Conceptually, Crossy is an extension of the Nezha framework proposed by Petsios et al. [29] and also employs the concept of path δ-diversity on top of code coverage. Nezha realizes efficient differential testing by monitoring behavioral asymmetries between programs. Instead of concatenating the coverage information and dismissing the inherent difference between programs, Nezha defines a metric called path δ-diversity which examines the *combination* of the targets' coverage information. Since keeping track of all combinations encountered during a fuzzing run would exceed memory capabilities, Nezha defines two variants of path δ-diversity to approximate the true metric. The coarse version only keeps track of the combinations of the numbers of covered edges, while the fine metric monitors the combinations of the sets of covered edges. Since Nezha's implementation is based on an older version of libFuzzer, we re-implemented the concept for a newer version and incorporated it into our framework.

4 ANALYSIS

Equipped with the proposed Crossy framework, we continue to analyze JSON parsers across different programming languages and runtime environments. Our main objective is the identification and systematization of discrepancies among parsers.

Table 1: Overview of parsers. The set P of JSON parsers used in our experiment. The calibrated ensemble $\bar{P}_+$ is listed in bold.

Parser (Github projects)	Language	Edges	Exec/s
rustyrussell/ccan	C	622	54 732
DaveGamble/cJSON	C	958	27 379
cesanta/frozen	C	695	9866
akheron/jansson	C	1527	19 062
zserge/jsmn	C	221	64 433
json-c/json-c	C	1560	12 818
json-parser/json-parser	C	621	50 489
sheredom/json.h	C	849	87 110
vincenthz/libjson	C	256	60 931
pocoproject/poco	C	7405	8894
lloyd/yajl	C	960	56 015
boostorg/boost	C++	6142	37 684
open-source-parsers/jsoncpp	C++	4667	4095
nlohmann/json	C++	2800	18 208
mozilla/gecko-dev	C++	7324	33 536
Tencent/rapidjson	C++	815	53 896
v8/v8	C++	3764	26 683
google/gson	Java	≈ 4096	18 341
FasterXML/jackson	Java	≈ 65 536	13 466
json (stdlib)	Python	440	49 943
simplejson/simplejson	Python	788	51 762
serde-rs/json	Rust	24 254	7373

Therefore, we select 22 popular parsers, covering five programming languages (C, C++, Rust, Java, and Python) and three runtime environments (native, Java interpreter, Python interpreter). We find a notable number of differences between parsers which we group into eight bug classes in the following analysis. Our results indicate that most pairs of parsers suffer from at least one bug class and there exists only two groups of parsers that behave equivalent.

4.1 Setup

Before discussing our findings in detail, we first need to define the target set of parsers P, collect our initial corpus, and filter the representative subset $\bar{P}_+$ used for the consensus-based detection.

Parser selection. For our analysis, we select 22 popular JSON parsers, which are listed in Table 1. As basis for our selection, we use the parsers listed at json.org and filter according to the (a) programming language, (b) popularity (i.e., GitHub stars), and (c) whether the parser is still actively being developed.

For Java, we select *gson* from Google and *jackson* which is a well-known library for data serialization formats in Java. For Python, we choose the standard library's parser and *simplejson*. Note, that other popular options such as *ultrajson*, *orjson*, and *yajl-py* only provide Python bindings to parsers written in low-level languages. For Rust, we select *serde* JSON. For C, we select 11 parser which differ in their design, performance and targeted application. For example, *cJSON* is build for ease-of-use, *jsmn* is optimized for performance, and *frozen* is build for usage in embedded systems. For C++, we include parsers from popular browsers, i.e., *v8* (Chrome) and *spidermonkey* (Firefox; gecko-dev), as well as *rapidjson* from Tencent and the well-known *boost* project.

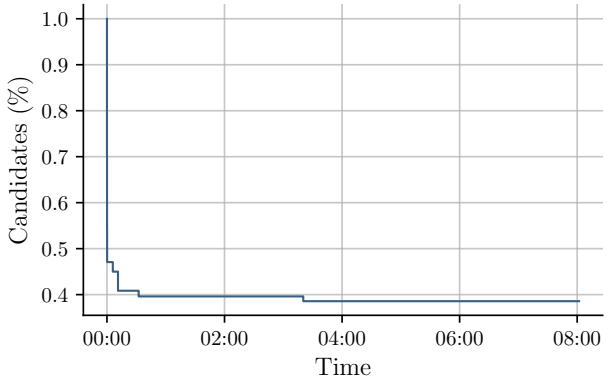

Figure 3: Consensus subgroup. Fraction of candidates that agree with the full ensemble as a function of the fuzzers run-time.

Fuzzing corpus. To bootstrap the fuzzing process, we need an initial corpus of JSON files. This corpus should include a diverse set of inputs used as a starting point for the fuzzer. Therefore, we use test cases that are specifically crafted to test the correctness of the JSON parsers. In particular, we browse through the parsers' repositories in our test-bed and collect a total of 566 distinct JSON files from eight different repositories.

Calibrated ensemble. As discussed in Section 2.2, using the entire ensemble of parsers P for the consensus-based detection would impose a significant performance burden. Therefore, we substitute this ensemble with a smaller, representative group $\bar{P}_+$ that captures the behavior of the complete set as best as possible.

In particular, we determine this calibrated ensemble by minimizing the following optimization problem

$$\bar{P}_+ = \underset{\bar{P}\in\mathcal{P}(P)}{\arg\min}\, \mathbb{E}_x\left[\Delta(P,x) - \Delta(\bar{P},x)\right], \tag{3}$$

where x denotes valid JSON inputs from a set X of JSON objects, the decision of the ensemble $\Delta(P,x)$, and the decision of a candidate group $\Delta(\bar{P},x)$. Simply put, we seek an ensemble whose majority decisions are as close as possible to the majority decisions of all considered parsers. For our analysis, we choose a group size of $|\bar{P}| = 3$ as a trade-off between expressiveness and overhead. This results in a total of $\binom{22}{3} = 1540$ possible groups.

To find a calibrated ensemble from this set, we conduct a calibration run for 8 hours with 96 fuzzing instances on a shared corpus to collect diverse set of inputs X. Subsequently, for each of these inputs, we compute the decision $\Delta(P,x)$ of the full ensemble as well as the decisions $\Delta(\bar{P},x)$ of each candidate. In Figure 3, we show the number candidate groups that agree with the full ensemble as a function of the fuzzers run-time. Around 50% of combinations are ruled out after a few minutes of fuzzing and the number of candidates saturates after around 4 hours at 38%. From the remaining set, we choose $\bar{P}_+ = \{jsmn, json.h, libjson\}$ as the group with the highest executions per second (see Table 1).

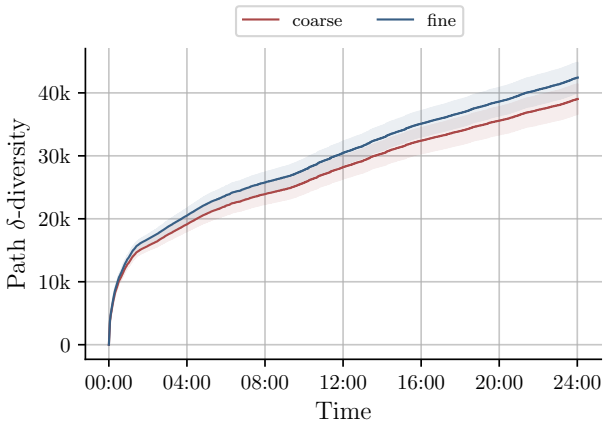

Figure 4: Path δ-diversity. The two path δ-diversity coverage metrics defined by Petsios et al. [29] during the testing run.

4.2 Bug Class Taxonomy

For our main experiment, we run CROSSY for 24 hours with the full set P of 22 parsers and $\bar{P}_+ = \{jsmn, json.h, libjson\}$ for the consensus-based normalization. We employ 96 parallel instances that work on a shared corpus.

Coverage. We observe a saturation of coverage in the individual parsers after around ten minutes. This is expected: JSON parsers are relatively simple programs and the high quality test samples in the initial corpus already cover most of the program space. However, the path δ-diversity, which is specifically designed for use in differential tests, continues to increases afterwards as shown in Figure 4. This confirms the findings by Petsios et al. [29] that path δ-diversity coverage is well suited for differential testing.

Bug classes. Over the entire fuzzing run, CROSSY collects a total set of 29 266 unique JSON inputs that lead to different behavior between parsers. Following the methodology introduced in Section 2.3, we create a normalized parse tree for each of these inputs. As the inputs caused a semantic discrepancy, there must be at least one conflicting node in the corresponding tree that can be uncovered through a traversal. Therefore, to categorize inputs into bug classes, we traverse all nodes for a given input and categorize the bug according to the identified type of node where the difference occurs.

Taxonomy of differences. Based on the identified bug classes, we construct a taxonomy describing their relationship. Recall that the JSON grammar defines seven fundamental values from which we use `object`, `array`, `number`, and `string` to construct this taxonomy. This initial *grammar-based* nodes are shown as solid in Figure 5. For brevity, we only include nodes for which we found discrepancies. As such we exclude primitive values of `null`, `true`, and `false`.

Starting from major type differences, we manually analyze the uncovered inconsistencies in the inputs and extend the grammar by adding more refined bug classes. On the first level these include semantic properties as discussed in Section 2.3. For objects, we include `name error` and `length error` which are violations of the set of object names or the length of the object respectively. These are further differentiated into errors from individual parsers. For numbers, we include the distinction between integers and floating point numbers. Depending on the type, if the values of the number differ, we assign either an `integer error` or a `float error`. If the values differ within a margin of error, we assign the more fine-grained `precision error`. Some parsers unexpectedly cast from integer to float which we denote by `type error`.

Parsing error. There are two bug classes not listed in the taxonomy. These stem from the fact that either one or both JSON parsers might transform valid into invalid JSON, from which we can not build a parse tree to analyze. We treat these cases as two additional bug classes: `Parse error` and `Parse error both`.

Interoperability matrix. Based on the bold printed nodes from Figure 5, we construct an interoperability matrix between all pairs of parsers in Figure 6. The rows and columns contain the parsers whose output we compare, and the cell represents the amount of bug classes this pair is susceptible to. The diagonal, naturally, consists of only zero entries since each parser returns the same output as itself. This representation provides a condensed view on our analysis and indicates the alarming number of differences between the considered parsers.

Incompatibility between parser. Most of the parsers exhibit at least one incompatibility with another parser. Although the bug classes do not always constitute a serious problem, the prevalence of the incompatibilities come at a surprise given the rather low complexity of the JSON standard. *json-parser* stands out from all other parsers as it exhibits the highest incompatibility with all other parsers.

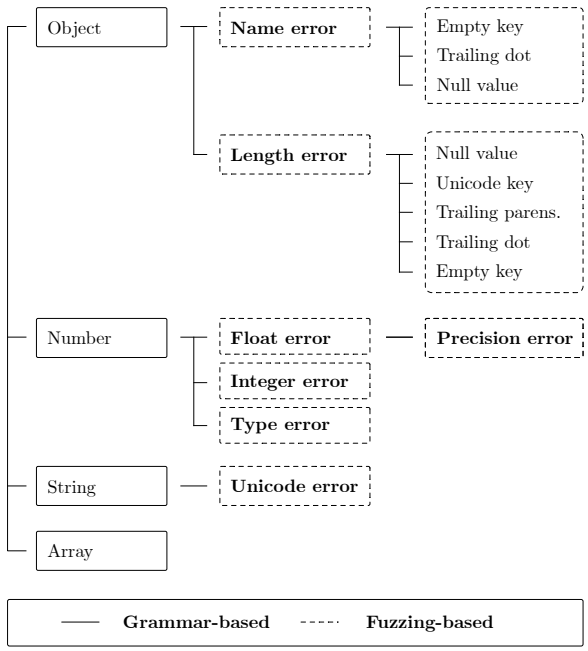

Figure 5: Bug class taxonomy. For our taxonomy, we differentiate between grammar-based nodes and more fine-grained nodes found empirically during fuzzing. We denote the nodes that we use as basis of our analysis in **bold**.

Table 2: Invalid encoding handling. We differentiate four different cases: "u" denotes the case that the parser outputs the character in "\u escapement notation". "x" denotes the case that the parser outputs the raw character. "T" denotes the parser returning a valid JSON object with a truncated string. "/" denotes an invalid JSON object resulting from a truncated string. And an underscore "_" denotes a parsing error. Error cases are highlighted in red. Critical cases are denoted in **bold**.

Input	ccan	cJSON	jansson	jsmn	json-c	json-parser	json.h	jsoncpp	gson
0x00	_	**T**	_	_	_	_	_	u	u
0x01	_	u	_	x	u	x	x	u	u
…									
0x07	_	u	_	x	u	x	x	u	u
0x08	_	u	_	x	u	u	u	u	u
0x09	_	u	_	x	u	u	_	u	u
0x0a	_	u	_	x	u	u	_	u	u
0x0b	_	u	_	x	u	x	x	u	u
0x0c	_	u	_	x	u	u	u	u	u
0x0d	_	u	_	x	u	u	_	u	u
0x0e	_	u	_	x	u	x	x	u	u
…									
0x1f	_	u	_	x	u	x	x	u	u
"\u0000"	_	**T**	_	u	u	/	/	u	u
"\u0001"	u	u	u	u	u	x	x	u	u
…									
"\u0007"	u	u	u	u	u	x	x	u	u
"\u0008"	u	u	u	u	u	u	u	u	u
"\u0009"	u	u	u	u	u	u	u	u	u
"\u000a"	u	u	u	u	u	u	u	u	u
"\u000b"	u	u	u	u	u	x	x	u	u
"\u000c"	u	u	u	u	u	u	u	u	u
"\u000d"	u	u	u	u	u	u	u	u	u
"\u000e"	u	u	u	u	u	x	x	u	u
…									
"\u001e"	u	u	u	u	u	x	x	u	u
"\u001f"	**x**	u	u	u	u	x	x	u	u

Intra-/ and inter-language. One could expect parsers from the same language to be more compatible or exhibit the same behavior (e.g., because there are common pitfalls when handling Unicode characters). However, we find that the language does not seem to influence the compatibility. Neither intra-language nor inter-language incompatibilities show any significant differences.

Groups of parsers. We find two clusters of parsers to be completely compatible with each other: The first group is an array of cross-languages parsers: (C) *jansson*, *jsmn*, *json-c*, *libjson*, *yajl* and (C++) *jsoncpp*, *nlohmann* and (Java) *jackson*. The second one contains *v8* and *spidermonkey*. Both of these stem from JavaScript engines for which compatibility is particularly important.

4.3 Case studies

We continue to discuss selected bugs that we discovered during our analysis. These semantic inconsistencies serve as examples of how parsers can arrive at different interpretations and how such inconsistencies can impact security.

Omission of null values. The unusual yet documented default behavior of the parser *gson* is to remove name/value pairs in JSON objects where the value is null, that is, the input `{"key": null}` is serialized back to `{}`. This behavior is also shared by the Java parser *fastjson*, which is deprecated at the time of writing so it is not considered in our analysis [19].

$$\texttt{\{"key": null\}} \longrightarrow \texttt{\{\}}$$

Although removing a member whose value is null seems like a logical step, it might lead to unexpected behavior if subsequent steps rely on the presence of the member. In a security context, for example, the absence of a specific key in an object may trigger the loading of a default, potentially insecure value.

Unexpected type coercion. The JSON specification defines the "number" concept without a clear differentiation between integers and floating point numbers. For example, given an integer number without decimal point, most implementations return the exact same integer representation. The *gson* parser, however, always returns a floating point number (i.e., a number containing a decimal point). Also, semantic inconsistencies may arise when numeric values are crossing the boundaries of internal types for representation. If numbers exceed such a range, some parsers use type coercion to transform the integer into a floating point approximation. The parsers *frozen*, *jsmn*, *json.h*, *libjson*, *yajl*, *jackson*, *py-json*, and *simplejson* return the original number without modification whereas *cJSON*, *jansson*, and *poco* do not return any output for large numbers. Interestingly, for very large numbers ($> 10^{100}$) *gson* no longer returns a float, but a string representation of the whole number. These differences become security-critical when large numbers are used for authentication, for example, as session identifiers or timestamps, and therefore minor changes can invalidate a running session and trigger re-authentication.

Parsing of control characters. The JSON specification requires control characters (`U+0000` through `U+001F`) to be in "\u escapement notation" in strings. If control characters are included as bytes, parsers should reject the input as it does not conform to the JSON grammar. However, there are several parsers that accept inputs containing control characters (*cJSON*, *jsmn*, *json-c*, *json-parser*, *json.h*, *jsoncpp*, *gson*), effectively ignoring this aspect of the specification. The concrete parsing behavior is shown in the upper half of Table 2. This is an unexpected result: the parsers not only suffer from the imprecise specification, but even build further ambiguities into their implementations, thus weakening their interoperability. Obviously, inconsistent representation of strings can have serious consequences for security, for example, when they are used to match login names and permissions.

Invalid control characters serialization. Related to this, a serialization function should not output control character bytes as this results in invalid JSON. This is especially problematic when control

	C											C++							Java		Python		Rust
	ccan	cjson	frozen	jansson	jsmn	json-c	json-parser	jsonh	libjson	poco	yajl	boost	nlohmann	jsoncpp	rapidjson	spidermonkey	v8	gson	jackson	py-json	simplejson	serde
ccan	0	4	6	5	5	5	7	6	5	5	5	5	5	5	5	4	4	5	5	5	6	5
cjson	4	0	6	4	4	4	6	5	4	4	4	4	4	4	4	4	4	4	4	5	5	4
frozen	6	6	0	2	2	2	6	2	2	3	2	4	2	2	4	4	4	4	2	3	3	5
jansson	5	4	2	0	0	0	5	1	0	1	0	2	0	0	2	2	2	2	0	1	1	3
jsmn	5	4	2	0	0	0	5	1	0	1	0	2	0	0	2	2	2	2	0	1	1	3
json-c	5	4	2	0	0	0	5	1	0	1	0	2	0	0	2	2	2	2	0	1	1	3
json-parser	7	6	6	5	5	5	0	4	5	5	5	5	5	5	5	6	6	6	5	6	6	5
jsonh	6	5	2	1	1	1	4	0	1	2	1	3	1	1	3	3	3	3	1	2	2	4
libjson	5	4	2	0	0	0	5	1	0	1	0	2	0	0	2	2	2	2	0	1	1	3
poco	5	4	3	1	1	1	5	2	1	0	1	3	1	1	3	2	2	2	1	2	2	3
yajl	5	4	2	0	0	0	5	1	0	1	0	2	0	0	2	2	2	2	0	1	1	3
boost	5	4	4	2	2	2	5	3	2	3	2	0	2	2	2	4	4	4	2	3	3	3
nlohmann	5	4	2	0	0	0	5	1	0	1	0	2	0	0	2	2	2	2	0	1	1	3
jsoncpp	5	4	2	0	0	0	5	1	0	1	0	2	0	0	2	2	2	2	0	1	1	3
rapidjson	5	4	4	2	2	2	5	3	2	3	2	2	2	2	0	4	4	4	2	3	3	3
spidermonkey	4	4	4	2	2	2	6	3	2	2	2	4	2	2	4	0	0	2	2	3	3	4
v8	4	4	4	2	2	2	6	3	2	2	2	4	2	2	4	0	0	2	2	3	3	4
gson	5	4	4	2	2	2	6	3	2	2	2	4	2	2	4	2	2	0	2	3	3	4
jackson	5	4	2	0	0	0	5	1	0	1	0	2	0	0	2	2	2	2	0	1	1	3
py-json	5	5	3	1	1	1	6	2	1	2	1	3	1	1	3	3	3	3	1	0	1	4
simplejson	6	5	3	1	1	1	6	2	1	2	1	3	1	1	3	3	3	3	1	1	0	4
serde	5	4	5	3	3	3	5	4	3	3	3	3	3	3	3	4	4	4	3	4	4	0

Figure 6: Interoperability matrix. Each cell displays the number of unique output difference classes between the parser pairs.

characters in "\u escapement notation" are serialized to bytes, because this transforms a former valid JSON string into invalid JSON. The faulty serialization behavior is marked with an "x" in Table 2. Interestingly, we found one case where this error manifests even for a single parser: *ccan* becomes self-incompatible as `"\u001f"` is serialized to `0x1f` which *ccan* itself can not parse again. Affected parsers: *ccan*, *jsmn*, *json-parser*, *json.h*.

`["\u001f"]` → `[ 0x1f ]`

By transforming valid into invalid JSON, the parser breaks a core integrity assumption about the data. If subsequent systems rely on the validity of the output of the parsers, this might lead to unexpected behavior or crash the application.

Invalid handling of UTF-16 surrogate pairs. Unicode characters not in the Basic Multilingual Plane are encoded using the 12-character escape sequence. We found *json-parser* to incorrectly handle these codes, e.g., the string `"\uDBFF\uDFFF"` is serialized back to `0xf3 0xbf 0xbf 0xbf` instead of `0xf4 0x8f 0xbf 0xbf`.

Invalid handling of `U+0000`. In languages containing null terminated strings, the null character indicates the end of a string. This leads to problems when parsing JSON. If a valid JSON string contains the null character in "\u escapement notation", *ccan* and *jansson* do not parse the string. For *cJSON* the documented behavior is to return a string that is truncated after the null character. The parsers *json-parser* and *json.h* truncate the entire JSON object after the null character which leads to an invalid output.

`["Visible\u0000Hidden"]` → `["Visible"]`

Truncating the string after the null character in "\u escapement notation" breaks the integrity of the data. For example, in a setting where the parser is used in a component that validates incoming data, malicious data can be appended behind the null character to bypass validation.

Invalid handling of names. The *frozen* parser exhibits faulty behavior that allows an attacker to manipulate the JSON object which has been assigned the CVE ID "CVE-2023-48891" [8]. This undermines the integrity of the JSON processing as data might be removed, the JSON structure might be manipulated, or the resulting string might be made invalid. The behavior occurs for name/value pairs in JSON objects where the name is empty or ends in a period ("."). The resulting output depends on the corresponding value's type of the affected member:

(1) If the value is a primitive type (i.e., number, string, boolean, null), the entry is removed from the object. For example the input `{"key.": 10}` is transformed to `{}`. If subsequent parsers depend on the presence of the member this might lead to unexpected behavior.

`{"key.": 10}` → `{}`

(2) If the value is an array, the values of the array are added to the parent JSON object. Because this can never result

in a valid member, the resulting output is always invalid JSON, i.e., valid JSON can be turned into invalid JSON. As an example, the input `{"key.": [1,2,3]}` will be serialized to the invalid JSON output `{1,2,3}`. This behavior makes *frozen* self-incompatible as it can not be applied to its own output.

`{"key.": [1,2,3]}` → `{1,2,3}`

(3) The last configuration is the most critical. If the corresponding value is an object, the object is integrated into its parent object, e.g., the input `{"key.": {"admin": true}}` results in the output `{"admin": true}`. If an attacker has control over the name in the member, it is possible to modify the structure of the resulting JSON string. If the attacker can additionally control the nested JSON object, it is possible to rewrite the parent JSON object.

`{"key.": {"admin": true}}` → `{"admin": true}`

Similarly, for entries in JSON objects where the key ends in "]", the values get integrated into the parent JSON object. This is equivalent to the behavior where the name ends in a period and the value is an array. As an example, the input `{"key]": [1,2,3]}` gets serialized to `{[1,2,3]}`.

`{"key]": [1,2,3]}` → `{[1,2,3]}`

5 LIMITATIONS

Our approach to differential testing of JSON parsers is based on different assumptions and naturally entails limitations, which we discuss below.

Independent execution. Since our implementation of Crossy is build on libFuzzer, we inherit the assumption that the executions of targets are independent of each other, that is, an input always triggers the same functionality and produces the same result. For JSON parsers in our experiments, we believe this to be a reasonable assumption, as these consume data in one go and do not rely on a streaming state that could disrupt subsequent inputs. To track cross-language coverage, we further make the assumption that targets do not fork to execute subprocesses or call into lower-level languages.

Incorrect normalization. Our method for normalizing JSON is based on the consensus of an ensemble of parsers. Although the realized combination of majority and round-robin voting greatly reduces the probability of error, we cannot rule out that the normalization works correctly in all possible cases. In particular, if the majority of the ensemble errs, discrepancies can be overlooked (false negatives) or incorrectly reported (false positives). Without a precise specification, such errors cannot be avoided. However, by increasing the size of the ensemble and calibrating it, as done in our analysis, we can ensure that such errors are at least rare.

Vulnerabilities and discrepancies. Our framework Crossy enables uncovering semantic differences in the interpretation of JSON data. Apparently, some of these inconsistencies are only minor misunderstandings and do not point to obvious security vulnerabilities. Nevertheless, it is also difficult to rule out that they have no impact on security. Given the widespread use of JSON in web applications, embedded systems and other security-related services, it is challenging to generally prove that a semantic difference cannot be the basis for a vulnerability. As we discuss in Section 4, for many of the found defects, there exist scenarios where the divergent interpretation leads to exploitable confusion. Consequently, we argue that any form of differences needs to be avoided when employing JSON across multiple parsers.

Table 3: Differential testing approaches. Language agnostic: Is used for multiple languages ✓, could be used for multiple languages (✓), is restricted to a single language ✗.

Project	Targets	Generation	Coverage-guided	Language-agnostic	Comparison
Our work	JSON Parser	M	✓	✓	O
CSmith [35]	C compiler	G	✗	(✓)	V
T-Reqs [20]	HTTP-Parser	G,GM	✗	✓	O
Harrand et al. [19]	JSON Parser	C	✗	(✓)	E+M
JIT-Picker [10]	JS Engine	M	✓	✗	V
classfuzz [16]	JVM	M	✓	✗	E
classming [17]	JVM	M	✓	✗	E
RustSmith [33]	Rust Compiler	G	✗	(✓)	O
Frankencerts [13]	SSL/TLS Parser	GM	✗	✓	E
Mucert [15]	SSL/TLS Parser	M	✓	✗	E

C: *Corpus*, G: *Grammar-based generation*, M: *Mutations*, GM: *Grammar-based mutations*
E: *Exit codes*, M: *Manual analysis*, O: *Program output*, V: *Internal variable states*

6 RELATED WORK

Differential testing was first introduced 1998 by McKeeman [24] for software testing. Since then, it has been applied to various domains such as testing C compilers [24, 26, 27, 35], JavaScript engines [4, 10, 28], the Java virtual machine [16, 17, 34], or SSL/TLS parsers [13–15].

In this work, we use differential testing to automatically test JSON parsers across multiple languages. We summarize related approaches in Table 3. CSmith [35] and T-Reqs [20] generate inputs from a random seed for C compiler and HTTP parser testing, respectively. Brubaker et al. [13] present an approach that randomly mutates existing X.509 certificates to create "frankencerts". Another line of work borrow methods from the field of fuzz testing. Most notably, this includes coverage-guidance that dynamically instrument targets to improve the input generation. For example, Chen and Su [15] present X.509 certificates called "mucerts" that have been generated using mutations sampled by coverage-guided Markov chain Monte Carlo. Bernhard et al. [10] propose JIT-Picker that uses coverage-guided fuzzing to mutate an intermediate representation to differentially test the interpreter and Just-In-Time compilers of a JavaScript engine.

Petsios et al. [29] exploited the behavioral asymmetries between targets under test. Instead of disregarding the origin of different coverage sources, the interplay of the targets is respected and used as additional information in the differential testing process. Petsios et al. evaluate their on SSL/TLS libraries, PDF viewers, and ELF and XZ parsers. For Crossy we incorporate the concept of their proposed path data diversity metric.

With regards to JSON testing, Seriot [32] has created manual tests based on the JSON specification and compared various JSON parsers for the specification conformity. The primary focus of this work was on whether the parsers accepted or rejected specific test cases. In contrast, our work focuses on whether the semantics of the inputs can be correctly deserialized and serialized. Miller [25] analyze the interplay of JSON parsers based on exemplary case studies. Lastly, Harrand et al. [19] compared Java based JSON parsers by evaluating them on a fixed testbed of valid and invalid JSON input samples. Their work includes a manual in-depth program analysis as they inspect the employed data structures of the JSON parsers.

Another area related to Crossy is concerned with the cross-language testing of applications. While our work implements cross-language differential testing across *multiple* programs of the same functionality, Li et al. [22] use cross-language fuzzing on *one* application consisting of multiple programming languages. More similar to our work is the collaborative fuzzing approach by Li et al. [23] that jointly fuzzes the Python runtime and Python applications with coverage support to find bugs in the Python runtime.

7 CONCLUSION

Data ambiguity is a notorious root cause of security problems in the transfer and processing of data. This is especially true for general-purpose formats, such as JSON, which are deployed in several security-critical systems and protocols. While prior work has been mainly concerned with manual testing of JSON parsers, we demonstrate that differential testing is a crucial strategy for unveiling semantic differences in their interplay. Our analysis reveals a wide range of discrepancies between common parsers and maps out the attack surface of employing JSON parsers in conjunction.

We have notified the developers of the JSON parsers with our discovered discrepancies and reported the corresponding vulnerability to Mitre CVE. Aside from these individual defects, our analysis shows that the effects of an imprecise specification can hardly be resolved at the implementation level. We therefore advocate the enforcement of a stricter specification of JSON whenever the format is used in networks and distributed systems. By doing so, we echo existing efforts in the security community to establish more rigid standards, such as the I-JSON [11] format.

ACKNOWLEDGMENTS

This work was funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – (390781972), the European Research Council (ERC) under the consolidator grant MALFOY (101043410), and the German Federal Ministry of Education and Research under the grant BIFOLD24B.

REFERENCES

[1] Atheris: A coverage-guided, native python fuzzer. https://github.com/google/atheris. Accessed: 2023-12-08.
[2] Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://github.com/frida/frida. Accessed: 2023-12-08.
[3] Jazzer: Fuzz Testing for the JVM. https://github.com/CodeIntelligenceTesting/jazzer. Accessed: 2023-12-08.
[4] funfuzz. https://github.com/MozillaSecurity/funfuzz. Accessed: 2023-12-08.
[5] libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2023-12-08.
[6] CVE-2017-12635. Entry on MITRE, 2017.
[7] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.
[8] CVE-2023-48891. Entry on MITRE, 2023.
[9] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985. doi: 10.1109/TSE.1985.231893.
[10] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz. Jit-picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–364, 2022.
[11] T. Bray. The I-JSON Message Format. RFC 7493, 2015.
[12] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, Dec. 2017. URL https://www.rfc-editor.org/info/rfc8259.
[13] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129. IEEE, 2014.
[14] C. Chen, P. Ren, Z. Duan, C. Tian, X. Lu, and B. Yu. Sbdt: Search-based differential testing of certificate parsers in ssl/tls implementations. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 967–979, 2023.
[15] Y. Chen and Z. Su. Guided differential testing of certificate validation in ssl/tls implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 793–804, 2015.
[16] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of jvm implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99, 2016.
[17] Y. Chen, T. Su, and Z. Su. Deep differential testing of jvm implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1257–1268. IEEE, 2019.
[18] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, 2012.
[19] N. Harrand, T. Durieux, D. Broman, and B. Baudry. The behavioral diversity of java json libraries. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 412–422. IEEE, 2021.
[20] B. Jabiyev, S. Sprecher, K. Onarlioglu, and E. Kirda. T-reqs: Http request smuggling with differential fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1805–1820, 2021.
[21] M. B. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519, 2015.
[22] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai. Polyfuzz: Holistic greybox fuzzing of multi-language systems. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
[23] W. Li, H. Yang, X. Luo, L. Cheng, and H. Cai. Pyrtfuzz: Detecting bugs in python runtimes via two-level collaborative fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1645–1659, 2023.
[24] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10 (1):100–107, 1998.
[25] J. Miller. An exploration of json interoperability vulnerabilities. https://bishopfox.com/blog/json-interoperability-vulnerabilities. Accessed: 2023-10-05.
[26] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of c compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*, pages 48–53, 2012.
[27] E. Nagai, A. Hashimoto, and N. Ishiura. Reinforcing random testing of arithmetic optimization of c compilers by scaling up size and number of expressions. *IPSJ Transactions on System LSI Design Methodology*, 7:91–100, 2014.
[28] J. Park, S. An, D. Youn, G. Kim, and S. Ryu. Jest: N+ 1-version differential testing of both javascript engines and specification. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 13–24. IEEE, 2021.
[29] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.
[30] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, pages 22–es, 2004.
[31] A. Rundgren, B. Jordan, and S. Erdtman. JSON Canonicalization Scheme (JCS). RFC 8785, June 2020.
[32] N. Seriot. Parsing json is a minefield. https://seriot.ch/projects/parsing_json.html. Accessed: 2023-10-05.
[33] M. Sharma, P. Yu, and A. F. Donaldson. Rustsmith: Random differential compiler testing for rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1483–1486, 2023.
[34] M. Wu, Y. Ouyang, M. Lu, J. Chen, Y. Zhao, H. Cui, Y. Guo, and Y. Zhang. Sjfuzz: Seed & mutator scheduling for jvm fuzzing. In *2023 The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023.
[35] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.