Grammar-based Fuzzing of Data Integration Parsers in Computational Materials Science

Jan Arne Sparka[1], Sebastian Müller[1], Martin Kuban[2], Claudia Draxl[2], and Lars Grunske[1]

[1]Humboldt Universitat zu Berlin Institut fur Informatik
[2]Humboldt-Universitat zu Berlin Institut fur Physik

March 17, 2023

Abstract

Context: Computational materials science (CMS) focuses on *in silico* experiments to compute the properties of known and novel materials, where many software packages are used in the community. The NOMAD Laboratory1 offers to store the input and output files in its FAIR data repository. Since the file formats of these software packages are non-standardized, parsers are used to provide the results in a normalized format. **Objective:** The main goal of this article is to report experience and findings of using grammar-based fuzzing on these parsers. **Method:** We have constructed an input grammar for four common software packages in the CMS domain and performed an experimental evaluation on the capabilities of grammar-based fuzzing to detect failures in the NOMAD parsers. **Results:** With our approach, we were able to identify three unique critical bugs concerning the service availability, as well as several additional syntactic, semantic, logical, and downstream bugs in the investigated NOMAD parsers. We reported all issues to the developer team prior to publication. **Conclusion:** Based on the experience gained, we can recommend grammar-based fuzzing also for other research software packages to improve the trust level in the correctness of the produced results.

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

EXPERIENCE REPORT

Grammar-based Fuzzing of Data Integration Parsers in Computational Materials Science

Sebastian Müller[1,4,5] | Jan Arne Sparka*[1,4,6] | Martin Kuban[2,3,4,7] | Claudia Draxl[2,3,8] | Lars Grunske[1,9]

[1]Institut für Informatik,
Humboldt-Universität zu Berlin, Berlin, Germany
[2]Physics Department, Humboldt-Universität zu Berlin, Berlin, Germany
[3]IRIS Adlershof, Humboldt-Universität zu Berlin, Berlin, Germany
[4]Contributed equally
[5]Orcid: 0000-0002-3057-1125
[6]Orcid: 0000-0002-5886-4595
[7]Orcid: 0000-0002-1619-2460
[8]Orcid: 0000-0003-3523-6657
[9]Orcid: 0000-0002-8747-3745

Correspondence
*Jan Arne Sparka, Institut für Informatik, Humboldt-Universität zu Berlin, Berlin, Germany. Email: sparkaar@hu-berlin.de

Context: Computational materials science (CMS) focuses on *in silico* experiments to compute the properties of known and novel materials, where many software packages are used in the community. The NOMAD Laboratory[1] offers to store the input and output files in its FAIR data repository. Since the file formats of these software packages are non-standardized, parsers are used to provide the results in a normalized format.
Objective: The main goal of this article is to report experience and findings of using grammar-based fuzzing on these parsers.
Method: We have constructed an input grammar for four common software packages in the CMS domain and performed an experimental evaluation on the capabilities of grammar-based fuzzing to detect failures in the NOMAD parsers.
Results: With our approach, we were able to identify three unique critical bugs concerning the service availability, as well as several additional syntactic, semantic, logical, and downstream bugs in the investigated NOMAD parsers. We reported all issues to the developer team prior to publication.
Conclusion: Based on the experience gained, we can recommend grammar-based fuzzing also for other research software packages to improve the trust level in the correctness of the produced results.

KEYWORDS:
Search Based Testing Grammar Based Fuzzing Test Case Generation.

1 | INTRODUCTION

In computational materials science (CMS) both existing and hypothetical materials are simulated at the atomistic level for in-depth studies of materials properties and comparison to experimental findings. The work horse for this type of investigation is density-functional theory (DFT)[2]. The practical application of this theory requires the implementation of the underlying equations and possible approximations in specialized software packages, so-called *DFT codes*. Today, there are more than 45 different DFT codes used by the scientific community. They differ not only in the employed approximations and algorithms, but also in programming languages and file formats. The *NOMAD Laboratory* [1] was created to provide a centralized repository for the data produced by such codes[1] . To achieve this goal, the results of DFT calculations are extracted and translated to a common data model. Within NOMAD, this extraction is solved by implementing a parser for each code. These NOMAD parsers are software themselves. Thus, they are also susceptible to logical errors and bugs – making them potentially another bottleneck

[1] https://nomad-lab.eu/ last accessed on 24.02.2023

in the trust chain of the scientific workflow. In order to increase trust in the correctness of the materials data stored within the NOMAD Repository, we propose using grammar-based fuzzing[3] that can help to detect possible issues with the parsers. However, as generating a grammar for each of the codes' in- and output files is a largely manual process, we restrict ourselves to four codes[2], namely VASP[4], FHI-aims[5], Quantum Espresso[6], and `exciting`[7].

The main goal of this paper is to increase trust in the NOMAD parsers. Furthermore, we share our experience and findings of using grammar-based fuzzing on these parsers. For this purpose, we address the following research questions:

RQ1 Is grammar-based fuzzing in the context of CMS code parsers feasible? If there are major challenges in extending grammar-based fuzzers to work with CMS code parsers, what are they, and how can they be overcome?

RQ2 Can we uncover any issues in the four selected parsers using grammar-based fuzzing?

RQ3 Do these detected issues indicate an actual issue with the implementation of the respective parser, or did fuzzing generate physically/chemically impossible inputs that lead to the observed behavior?

Accordingly, our paper provides the following primary contributions:

(i) We publish our manually generated grammars, as well as configurations of the employed fuzzer. As the final step to answer RQ1, we also show results from an initial fuzzing run.

(ii) As the first study, we apply systematic fuzz testing to four of the used parsers in the NOMAD Repository. This will answer RQ2.

(iii) We conduct an empirical study measuring issues and failures of the four selected parsers, as well as label encountered issues according to their severity. Furthermore, we discuss the relevance of the findings w.r.t. their impact on the CMS field.

(iv) We found 118 unique issues, analyzed their causes, and reported the analyzed issues as part of a responsible disclosure to the relevant stakeholders.

(v) Finally, we provide a deeper qualitative analysis of selected issues. These last three contributions will answer RQ3.

The remainder of this paper is structured as follows: In Sec. 2, we will present background in both fields. Then in Sec. 3, we showcase our grammar generation process. Furthermore, we provide insights into our prototype tool, its development, and our experimental setup. In Sec. 4, we publish the results of our experiments. In Sec. 5, we evaluate and discuss our findings. In Sec. 6, threats to validity and future work are presented. Finally, in Sec. 7, we briefly discuss our results and draw conclusions.

2 | BACKGROUND

The NOMAD parsers obviously rely on domain-specific terminology and design choices. In the following, we briefly introduce concepts that are relevant in this context. This includes an overview of the methodology that is used to generate the regular input for the NOMAD parsers, an overview of the NOMAD infrastructure, as well as an introduction to grammar-based fuzzing.

2.1 | DFT

In computational materials science, **D**ensity **F**unctional **T**heory[2] (DFT) is employed to investigate different materials properties from first principles, that is, without any empirical parameters. The results are comparable to experimental findings. In these calculations, the materials are described by their periodically repeated building blocks (unit cells). An example for it, is provided in Fig. 1 which shows the conventional unit cell of silicon in its most common form, the diamond structure. The material's properties depend on the probability distribution of the electrons in this cell, given the arrangement of the atom cores, by numerically solving the so-called Kohn-Sham equations. To obtain accurate results, memory and CPU time requirements can be high, in particular for complex crystal structures. Performing high-precision DFT calculations thus requires compute servers, clusters, or even large-scale HPC facilities.

[2] As of 17 March 2021, these four codes represented 82.76% of all calculations stored in the NOMAD Repository.

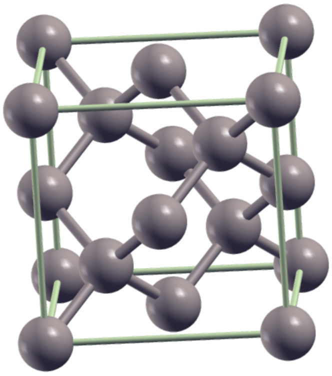


Figure 1 Ball-and-stick model of the conventional unit cell of silicon in the diamond structure.

2.2 | NOMAD

The **No**vel **Ma**terials **D**iscovery (NOMAD)[1] Repository has been online since 2014 and contains about 140 million DFT calculations[3]. It provides a platform for researchers to upload all necessary data that fully characterize a calculation. In doing so, these researchers make their own research specifically – and the field generally – compliant with the *FAIR*[8] principles:

- **F**indable: Storing all data in a single repository that is used across the scientific field increases the findability of data.
- **A**ccessible: With NOMAD being open for everyone to contribute to, and to easily request data from, the accessibility requirement is given as well.
- **I**nteroperable: NOMAD implements a standardized and open format to exchange data between different researchers.
- **R**eusable: The data can be used in various contexts, even very different to the purpose they have been created for.

Currently, there are parsers for more than 45 different codes available in the NOMAD Repository. In Fig. 2, we show the uploads of nine codes for which the repository is most widely used. NOMAD stores parsed data in a standardized common file format[9], while keeping the corresponding raw data as well.

The *nomad-lab* client[4] is a Python-based application that implements the current NOMAD infrastructure[1]. It is equipped with error handling and a structured logging system for exceptions in the parser execution. This allows for continuing to parse a file after an exception occurred. To identify similar error classes, the logging system provides `exception hashes`. These are hashes of the exception log event, where numbers, lists, and matrices are removed before hashing. Thus, errors become more independent of numeric values that were encountered in the logging process.

2.3 | Grammars

Definition 1. A **grammar** is a 4-tuple $G = (V, \Sigma, R, S)$, where V is a set of non-terminals; Σ the terminal alphabet; $R \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ the production rules; and S the start non-terminal.

Intuitively, a grammar describes how to construct words from an alphabet using a set of rules. All producible words from a given grammar G form the language over that grammar $L(G)$. A probabilistic grammar is a grammar where each rule is annotated with a probability, which describes the likelihood of deriving it. For each non-terminal, the probabilities of all its derivation options sum up to one. The language of a probabilistic grammar is the same as the corresponding non-probabilistic grammar, as long as all probabilities are greater than zero.

In this work, we employ grammars to describe the structure of the files that are used as input to the NOMAD parsers. In the following, we will refer to these files as `input files` only. Whenever we mean the input files of a DFT code, we refer to them

[3] https://nomad-lab.eu/prod/v1/gui/search/entries last accessed on 24.02.2023
[4] https://pypi.org/project/nomad-lab/ last accessed on 24.02.2023

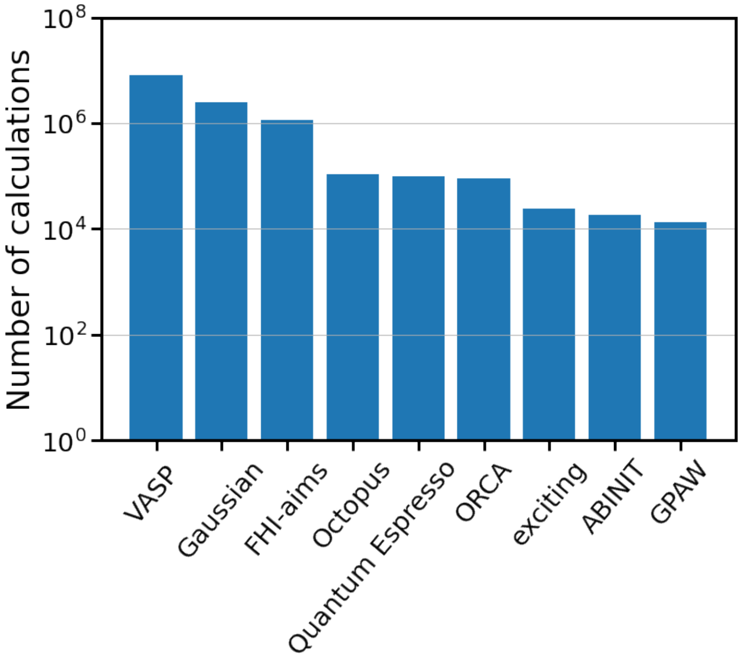


Figure 2 Number of uploaded calculations in the NOMAD Repository (as of Dec. 9, 2021); note the logarithmic scale. To increase readability, we only show codes with at least 10,000 calculations.

as *code input files*. From each start symbol in our grammars, we are able to generate input files conforming to this structure by successively applying the production rules. For each production rule, the left-hand side is replaced with one of the possible choices of the right-hand side of the respective rule. In context-free grammars, the left-hand side is exactly one non-terminal symbol. We note, that any numeric values are generated randomly. Therefore, the specific values read by the NOMAD parsers in the generated files might not conform to semantic constraints. For example, a value that specifically gives the length of an array is unlikely to match the grammar-generated array's length.

2.4 | Grammar-based Fuzzing

Fuzzing is a test-case generation technique that generates test inputs in a randomized fashion[10,11,12]. This technique is aimed at provoking erroneous program behavior. However, when working on a program that uses highly structured input data, just randomly generating inputs will result in only testing the input validation of the program under test (PUT)[13]. To overcome this limitation, grammar based fuzzers (e.g., Nautilus[14], EvoGFuzz[15], syntax machine[16]) were developed, that make use of a grammar for the structured inputs.

Furthermore, researchers employed probabilistic grammars in order to target specific regions of the PUT. These probabilistic grammars may be used to generate files that execute specific branches while being evaluated[13]. Using this technique, the fuzzer is able to generate test inputs that are (by definition of the grammar) syntactically valid, but can also be aimed at specific areas of the PUT that are under-tested according to some metric. Therefore, (probabilistic) grammar-based fuzzing is a sub-form of white-box fuzzing, in which a model of the PUT is available to the fuzzer. Here, the grammar is the model that is made available.

3 | GRAMMAR-BASED FUZZING FOR NOMAD PARSERS

Figure 3 presents the workflow that concerns our grammar-based fuzzing approach to test the NOMAD parsers. In Sec. 3.1 we describe the process of obtaining these grammars. For the generation of inputs for the PUT, we make use of TRIBBLE[17], which is a state-of-the-art grammar-based fuzzing tool. It allows generation of files, using probabilistic context-free grammars. This makes TRIBBLE well suited for our use-case. The files generated by TRIBBLE are post-processed to obtain suitable input for the

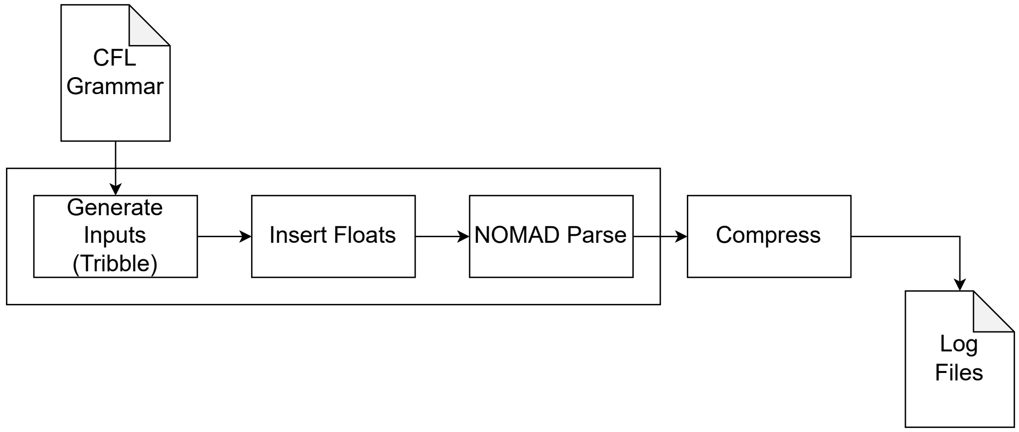


Figure 3 Workflow of testing the NOMAD parsers using input files generated by TRIBBLE.

```
file = header init scl timings (warnings)? footer;

header =
"
================================================================================
| EXCITING "("NITROGEN-13"|"CARBON"|"BORON"|"BERYLLIUM"|"BORON9")" started =
| version hash id: "("34c50d72ea9084531836c8a31250d12a863d7371"|"2595
    c4c2acb272ab81b33780c70930bf0ed17920")?" ="("
|                                                                              =
| MPI version using       " num " processor(s)                                 =")?"
|                                                                              =
| Date (DD-MM-YYYY) : " (date|str) "                                           =
| Time (hh:mm:ss)   : " (time|str) "                                           =
|                                                                              =
| All units are atomic (Hartree, Bohr, etc.)                                   =
================================================================================
";
lattice_vectors = " Lattice vectors (cartesian) :
" vectorentry vectorentry vectorentry "
";
vectorentry = " " nump "        " nump "        " nump "
";
```

Figure 4 Example of one of the generated grammar files, an excerpt from the grammar for the `exciting` code's *INFO.OUT* file.

NOMAD parsers. The inputs are then processed, using the experimental setup described in Sec. 3.2. The analysis pipeline used to process the large amount of thus obtained execution logs is described in Sec. 3.3.

3.1 | Grammar Generation

In order to answer RQ1, we manually create grammars for the selected parsers. For each of the four parsers, we use 5 code outputs for the carbon diamond structure, and 5 code outputs for the silicon diamond structure. Thus, we use a total of 40 samples across both material structures. All code outputs (hereafter referred to as material samples) are taken directly from the NOMAD Repository. These samples are selected in such a way that they stem from different researchers, in order to represent a wide-enough range of possible configurations. The corresponding material IDs can be found in the online resources accompanying this paper[5]. All of our manually created grammars are also available for review[6]. In total, the grammar creation process consists of:

[5] https://github.com/hub-se/fuzzingcmscodeparsers/blob/main/Data/ids.txt last accessed on 28.02.2023
[6] https://github.com/hub-se/fuzzingcmscodeparsers/tree/main/Data last accessed on 28.02.2023

```
==========================================================================
| EXCITING BERYLLIUM started                                             =
| version hash id: 2595c4c2acb272ab81b33780c70930bf0ed17920              =
|                                                                        =
| MPI version using      +195336911 processor(s)                         =
|                                                                        =
| Date (DD-MM-YYYY) : 07-11-2019                                         =
| Time (hh:mm:ss)   : qpvmM*                                             =
|                                                                        =
| All units are atomic (Hartree, Bohr, etc.)                             =
==========================================================================

 Lattice vectors (cartesian) :
        $$posfloat$$     $$posfloat$$     $$posfloat$$
        $$posfloat$$     $$posfloat$$     $$posfloat$$
        $$posfloat$$     $$posfloat$$     $$posfloat$$
```

Figure 5 TRIBBLE generated file from the given input grammar. To this end, TRIBBLE successively derives rules to their respective terminal symbols.

(i) collecting material samples,

(ii) abstracting the contained files of each material sample,

(iii) using available documentation to complete (or even specify missing) rules in the grammar,

(iv) checking if the grammar still fits all material samples, and

(v) checking if the grammar is syntactically correct.

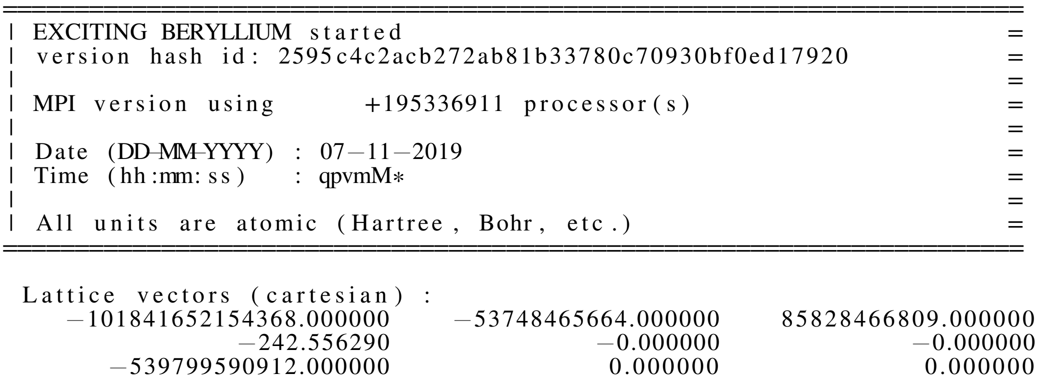
We use the TRIBBLE format to describe our grammars. In Fig. 4, we introduce our running example from the `exciting` code's *INFO.OUT* file. Note that we manually omit and align values for improved readability within this paper only.

The initial planning of the project contained the generation of grammars for **all** files that are passed to the parser, i.e., all possible raw data files read in and created by the DFT codes. Therefore, all in- and output files contained in the material samples are used for this initial step. To start our grammar creation process, we use a single material sample per code. Using the concrete sample, we first replace all numbers with an abstract number token, split into *floats* and *ints*. We then replace *strings* that appear to be variable (i.e., strings that are not magic strings). To end this initial grammar creation, we extract syntactic blocks that inherently belong together. For the `exciting` code, we also use the input and *species* files (information on the atomic species), as well as the tutorials available online[7]. We use these resources to generate grammars for all possible input files of that code at that time. (Note that the NOMAD parsers also require the output files of the respective code; the input alone would not have any meaning in a repository.) In terms of effort, based on our experience, it takes half a person day to generate the rough grammar files from a sample. In contrast, the generation of a complete grammar for all files described by the supplementary material of `exciting` takes several person days. After the generation of the input grammars for the `exciting` code, we realize that this is too time-consuming for the scope of this study.

To alleviate this, we only generate grammars for the main output file. The selected NOMAD parsers require at least these output files of each code, (the so-called *mainfile*) to start processing the data. This file contains the main results of the calculation. Occasionally, additional files are required, e.g., to read advanced results. However, most parsers require only a single file for operation, thus we focus on this basic functionality.

In a next step, these initial grammars are further refined using the until now unused material samples. We do this by replacing perceived constant strings in the grammars that vary between different material samples by a union of all observed variants (see Fig. 4). We are aware that we have miscategorized some actually constant strings with variable strings. During the later debugging step, many of these cases are fixed, due to their syntax breaking nature. Over all, the manual generation of grammars from samples takes a significant amount of time that varies between the different codes (two person weeks for Quantum Espresso, four for `exciting`, eight for FHI-aims, and almost sixteen weeks for VASP).

[7] http://exciting-code.org/ last accessed on 24.02.2023



```
================================================================================
| EXCITING BERYLLIUM started                                                   =
| version hash id: 2595c4c2acb272ab81b33780c70930bf0ed17920                    =
|                                                                              =
| MPI version using      +195336911 processor(s)                               =
|                                                                              =
| Date (DD-MM-YYYY) : 07-11-2019                                               =
| Time (hh:mm:ss)   : qpvmM*                                                   =
|                                                                              =
| All units are atomic (Hartree, Bohr, etc.)                                   =
================================================================================

  Lattice vectors (cartesian) :
     -101841652154368.000000     -53748465664.000000     85828466809.000000
                  -242.556290               -0.000000              -0.000000
         -539799590912.000000                0.000000               0.000000
```

Figure 6 File enriched with randomly generated float values. The floatifier.py script replaces meta-tokens by their corresponding float notation.

As seen in the running example, the input grammar is then used by TRIBBLE to generate concrete input files, a sample of which is shown in Fig. 5. Note that with our grammars, we only produce files where any floats are marked by a token. This is done because several formats for floats are used. Creating a float grammar that is (i) correct, (ii) has no class imbalance, and (iii) reflects all needed float formats is difficult and goes past the scope of this feasibility study. Thus, we replace the tokens with actual numeric float values in a second pass using a random-number generator. Figure 6 shows the example after this step.

The next step is re-evaluation. Here, we re-check the grammars manually (at least eight person days per code) against the inputs to see if we inserted any regressions. The reason for the considerable effort invested here is that some mainfiles in the material samples are several million lines long. Thus, cross-checking each of them takes time.

We then syntactically harden our created grammar files. During this step, we repair syntactic problems with the grammars themselves: This includes missing semicolons, missing brackets, or even mismatched quotation marks. The hardening process takes one person hour for all codes combined.

Finally, we start semantically hardening our created grammar files. To this end, we iteratively generate NOMAD parser inputs and feed them to the respective parsers. As an initial observation, we realize that a number of trivial issues re-occur for almost all testing. Among others, these recurring errors were:

(i) Assuming that any floats would be accepted, but only floats given in a special fixed-point format were actually parsed;

(ii) Missing magic numbers for some program options, as well as

(iii) Specific time/date formatting.

Consequently, testing the parsers using the grammars in this state would lead to shallow testing, with issues recurring extremely often. Our tool is intended to test the parsers in a more thorough fashion. Therefore, we fix a number of the more severely breaking bugs iteratively. While we do not want the parsers to crash in the exact same way each time, we do want to see those issues in the final tool, still. Thus, for less severe issues, we tweak the probabilities so that they would still be generated, but become less likely. This stage requires domain knowledge of the PUT. We were able to fix several previously undiscovered errors that prevented parsing. Generating the required amount of inputs takes about two person days – and only after are we able to analyze and check for problems in the grammars that prevent parsing. Using the analysis pipeline described in Sec. 3.3, we are able to reduce the effort of analyzing the observed errors to about one person day per code.

3.2 | Experimental Setup

We ran our experiments locally on a *Dell R920* compute server with 4 *Intel Xeon E7-4880 v2* processors (60 cores) @2,5 GHz, 1024 GB main memory, and an *openSUSE Leap 15.2* OS (running *Linux* kernel 5.3.18). All employed external tools and the versions used in this study are made public in Tab. 1.

The four selected parsers are tested sequentially in order to enable time tracking for fuzzing each individual parser. We employ the concept of experimental runs to mitigate the threat that randomness poses to such a study. We use 31 runs, which are split

Tool	Version
Python for experiments	3.6.12
Tribble	1.0.0
nomad-lab	0.10.2
nomad-lab[parsing]	0.10.2
Python for analysis	3.7.13
Jupyter	1.0.0
NumPy	1.21.5
matplotlib	3.4.2

Table 1 All tools and their versions as employed in this study.

into an initial feasibility run that is statically seeded with an initially chosen seed. This is followed by 30 experimental runs to evaluate the effectiveness of our approach.

For each parser, a run consists of the following sequential steps: We first generate 100 main output files using our earlier created grammars. We then call `nomad parse` on each of those files. This parsing process is done in parallel by our worker pool of 30 threads. We log the complete output for parsing each file in a single shared logfile per parser and run. We also track the timing, both in total, and for each individual `nomad parse` call.

We strive to make our experiments as reproducible as possible. To that end, we track all random seeds across the experimental runs. We offer two ways to add seeds to the command call. To reproduce a specific file for a parser from our results, we provide a reproduction string that needs to be given to the script. The reproduction string consists of (i) the parser information, (ii) the run seed, (iii) the file path, and (iv) the file seed. In order to make the tool extensible, we also offer a seed flag in the script. This flag is the basic seed value from which all run seeds are then derived. The number of runs is a changeable parameter of the published tool.

We define an experiment run as crashed, if and only if the run has produced no standard output. We define a run as failed, if there is output, but the parsing left an error in the log. We finally define a run as successful if the parsing has output and no error in the log. Note that we still say a run is successful, if there are warnings present in the log file.

3.3 | Analysis of experimental results

To gain an in-depth understanding of the errors found by the fuzzing runs, we manually analyzed each unique error and traced the emitted exceptions back to the parser's source code. This can only be achieved by capturing both input to, and output from the parser, as well as metadata, such as the runtime or the name of the input file (see Sec. 3.2).

We used NOMAD's parser error logging system to simplify the error classification process (see Sec. 2.2). Figure 7 describes the full workflow that is used to generate and analyze the execution traces. These traces are stored in a single output file. The output file is parsed into `LogBlock` objects, each describing a single call to `nomad parse`. Thus, the standard output (`STDOUT`) and error (`STDERR`), as well as the respective metadata are programmatically accessible. The `LogBlock` objects are added to a `LogSet` object, which represents the execution traces of all fuzzing runs. This object, besides convenience functions, allows us to find unknown errors using the function `find_unknown_errors`, given an `ErrorCollection` object.

In the beginning of the workflow, `ErrorCollection` is empty, as no errors are identified. An `Error` object is a dictionary describing the error emitted by the NOMAD client logging system (see Sec. 2.2). A domain scientist now checks the errors found by the function `find_unknown_errors`, traces them back to the source code of the NOMAD client, and classifies and comments the `Error` objects for improved readability. Any `Error` object is then saved to `ErrorCollection`, such that subsequent calls to `find_unknown_errors` will ignore this error and output only unclassified errors. This loop is repeated until no new errors are found.

To further simplify the manual analysis of any reported errors, `LogSet` containing all fuzzing execution traces is split into different subsets. Each of these subsets contains a unique set of errors. These errors are identified by the exception hash and the name of the error as reported by the NOMAD logging system.

For the classification of the errors, we decided on the following classes:

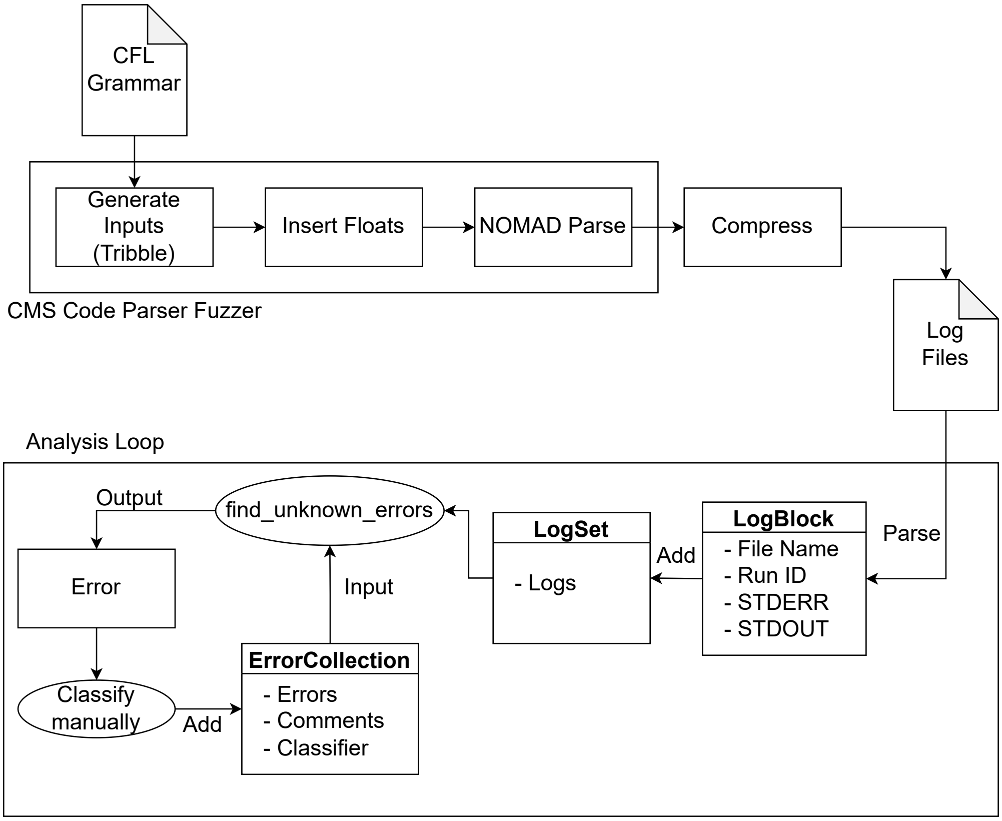


Figure 7 Overview of the tool and the analysis loop allowing for semi-automatic classification of errors.

- Critical: Errors being a potential threat to the NOMAD service
- Logical: Errors in the program logic of the NOMAD parsers (and their libraries)
- Semantic: Errors stemming from the semantic-agnostic nature of the grammar
- Syntactic: Errors coming from syntactic mismatch of the generated input file
- Downstream: Errors being a consequence of some problem earlier in the execution of the code, the fail-over of which leads to an issue in a subsequent submodule of the parsing infrastructure.

Note that if an error could belong to two or more classes simultaneously, we classify it in the more severe category.

4 | QUANTITATIVE RESULTS OF THE FUZZING CAMPAIGN

In this Section, we answer our first two research questions (see Sec. 1) using the results of our experimental runs.

4.1 | Answering RQ1: Feasibility

To answer RQ1 (see Section 1), we dedicated one experiment run of 100 input files per parser – in the remainder of this paper, we call this specific run the *feasibility run*. We found that our grammars (apart from the VASP parser) have at least 2 successful parser executions each, i.e., the parsers neither throw an error nor crash. While the VASP parser had no successful runs in the final statically seeded feasibility run, we know from our work on correcting the grammars that the grammar is indeed able to generate successful runs. This is due (at least in part) to the fact, that the VASP parser classifies many of the detected issues as errors, while the other parsers would classify similar issues only as a warning. We can thus conclude that our grammars do work to fuzz the selected NOMAD parsers in a grammar based fashion. The exact numbers of successful, failing, and crashing parsing calls can be found in Table 2.

Parser	Successful	Failing	Crashing
VASP	0	99	1
FHI-aims	59	34	7
Quantum Espresso	10	90	0
`exciting`	2	98	0

Table 2 Statistics of *feasibility runs* per parser.

Yes, grammar-based fuzzing is indeed feasible in the context of CMS code parsers. The biggest challenge we encountered was the semantics representation within our manually generated grammars. This challenge can be overcome by encoding the semantic constructs explicitly.

4.2 | Answering RQ2: Issue detection

Parser	Successful	Failing	Crashing
VASP	3	2928	69
FHI-aims	1856	1070	74
Quantum Espresso	413	2578	9
`exciting`	66	2894	40

Table 3 Statistics of all 30 runs per parser. Each run consists of 100 parser executions.

Table 3 shows statistics of the parser executions, sorted by the success level of the parsing process. Notably, the proportions of successful, failed, and crashed code executions is different for all 4 parsers. For the `exciting`, the Quantum Espresso, and the VASP parser, the vast majority ($\geq 86\%$) of runs are failing or crashing. For the FHI-aims parser, only $\sim 38\%$ of the runs are failing or crashing. This has two major reasons: (i) The syntax of the DFT code output files is different. For example, an output file of the VASP code contains a flavor of *.XML, which has strong constraints on the semantic correctness of the file. This comprises, among others, descriptions of arrays, which contain the size of the array as an attribute. In these cases, the semantic-agnostic nature of our fuzzing approach leads to inconsistencies. An output file of the FHI-aims code is made to be read by humans. Therefore, it does not contain much information about data structures. (ii) The parsers differ in robustness w.r.t. erroneous input. As such, for the FHI-aims parser, several values are replaced by hard-coded default values, if a certain quantity cannot be read from the input. This improves robustness, but imposes challenges on the correctness of the data processing. The replacement behavior can also be seen by considering the proportions of successful runs in Table 3: More than 60% of the runs using the FHI-aims parser are successful. For the Quantum Espresso parser, this ratio is significantly smaller with $\sim 13\%$. For the `exciting` parser, only $\sim 2\%$ of the runs are successful. Due to the above-mentioned strictness in correctness-checking, for the VASP parser, < 1‰ of the runs are successful.

Many of the failed runs still contained valuable information: Due to the error handling system of the NOMAD client, it is possible to continue parsing different sections after errors have been encountered. Therefore, even with few successful runs, it is possible to discover errors in all parts of the parsers' source code. Finally, the crashing runs indicate program breaking behavior, which, if not handled correctly, can lead to an interruption of service.

Yes, we can uncover issues in the parsers. Table 3 shows that we are able to even find crashing inputs for each of the four selected parsers.

5 | QUALITATIVE EVALUATION OF THE FOUND RESULTS

We find, through our analysis, that the parsers are inconsistent in their treatment of faults. Some parsers classify a file that cannot be read or where specific properties are absent as an outright error, while other parsers only provide a warning. This is a challenge for our evaluation, as this hampers comparing the parsers and their respective crashing, failing, or successful runs. Therefore, we show our assessments of each parser separately. We want to stress that every issue presented here already passed through a responsible disclosure. Thus, we do not present any zero-day vulnerabilities in this paper.

5.1 | Answering RQ3: Validity of the issues

Figure 8 presents box-and-whisker plots of the frequency of detected errors over all fuzzing runs. The visualization is by parser and sorted by error severity. Critical issues are rare and only found for the VASP parser. Logical errors are also not very common, but the probability of discovering semantic errors is high. In three cases, the same issue was triggered several times in the same parser execution. This is possible because the parser analyzes recurring subsections of the input. Due to the above-mentioned limitations of the fuzzer, it is improbable to generate semantically totally correct input data. Thus, the high number of semantic errors is expected. In a number of cases, the syntax of the fuzzer's output is also incorrect. This is partially due to the freedom of the grammar to generate specific keywords as random strings. Finally, the downstream errors are caught by the NOMAD error handling system. These are triggered, for example, if the parser cannot process a specific section of the input, that is required for a subsequent step. In some cases, these unprocessable sections are replaced by predefined defaults. These defaults may trigger inconsistencies in secondary downstream tools/submodules.

	Critical	Downstream	Logical	Semantic	Syntactic
VASP	6 (3)	2665 (6)	19 (4)	18280 (12)	1417 (6)
FHI-aims	0 (0)	1564 (7)	89 (4)	1060 (28)	2337 (3)
Q. Espresso	0 (0)	128 (5)	1234 (3)	4558 (19)	1391 (4)
`exciting`	0 (0)	4515 (3)	40 (2)	625 (8)	326 (1)

Table 4 Amounts of uncovered issues across all experimental runs. Numbers in brackets indicate unique issues.

Table 4 presents the total number of uncovered issues, as well as the number of unique issues across all fuzzing runs (brackets). As also evident from Fig. 8, the total number of issues is higher than the number of executions of the PUT, because several issues on the same input can occur repeatedly . We will now present a subset of those issues, where we give specific explanations as to what caused the issues. (All issues referenced here are also provided in the accompanying online materials.) They can be identified by their "error id", which we give in the examples below. The first issue (error id "vasp:critical:0") is a critical memory problem in the VASP parser. To trigger it, two conditions must be met:

(i) The parser input file must contain a section for an electronic density-of-states (DOS) spectrum, which has no entries.

(ii) A variable that describes the number of spin channels for the calculation must be set to a large number.

We note that neither of both conditions can be met with the output of a legitimate VASP calculation. We traced the error back to the following chain of events: The VASP parser reads the contents of the DOS section to a `NumPy` array[8]. The created data

[8] https://numpy.org/doc/1.21/reference/generated/numpy.array.html last accessed on 24.02.2023

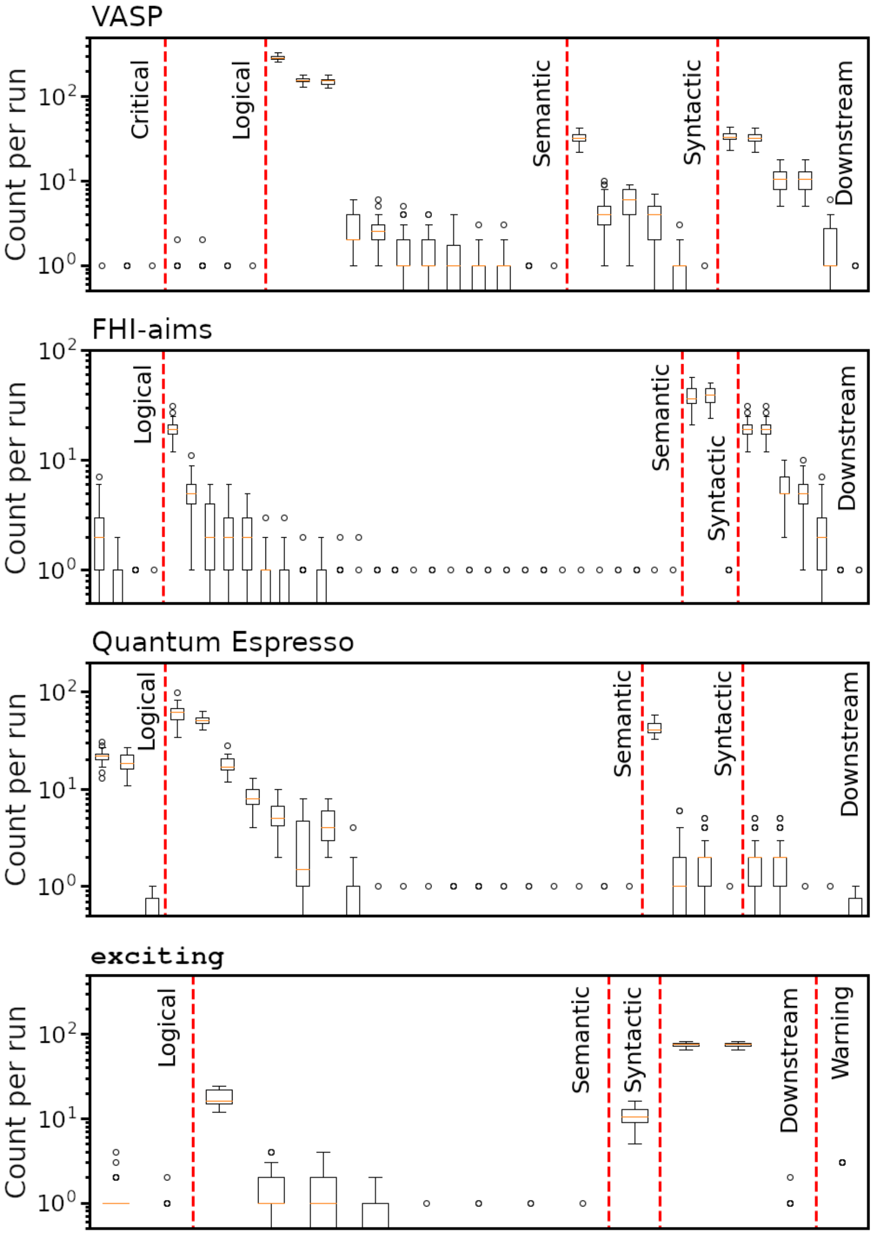


Figure 8 Box-and-whisker plots of errors for the different parsers under test, split by severity. Note the logarithmic scale.

structure contains no entries. In a later step, this array is reshaped using the spin value read from file. We note here that – in the version of NumPy that we used – an empty array can be reshaped to have any size. This process creates a large empty data structure, which (only within NumPy) does not consume additional memory. However, being physical quantities, these values have units. When the expected unit is converted, the external dependency (Pint) takes care of multiplying the values with the correct conversion factors. However, during the conversion, the data is copied. Then, the memory requirements may exceed the available memory, because both the original and the converted values are held in memory. This triggers the described memory error. We note that Pint provides the option to do the conversion in place.

The second critical issue that we discovered (error id "vasp:critical:1") is created by the same kind of erroneous input file, but has different consequences. When the reshaped and converted NumPy array described above is small enough to fit into memory, the processing proceeds. However, when the data is prepared to be stored, the array is transformed to a Python list. The Python list allocates memory for each element upon creation, possibly exceeding the systems resources. The effects of this problem vary: We can fill arbitrary amounts of disk space, drain the available RAM of the server running the NOMAD parser process, or even fully deny the server's ability to service jobs.

The third critical issue (error id "vasp:critical:2"), is again related to (ii), but occurs in a different part of the code. The DOS's of different calculations are aligned via their respective Fermi level, which corresponds to the highest occupied electronic state in a system. This value is stored separately for each spin channel. The parser creates a Python list that has the length of the number of spin channels. Again, if this number is (non-physically) large as chosen in our case, a list can be created that goes beyond the available memory, leading to this error.

The next issue (error id "exciting:logical:0") concerns the parser for the `exciting` code, more specifically the atom-relaxation section. Typically, this section contains the forces acting on each atom in the unit cell. This data is read into a two-dimensional `NumPy` array. To trigger the problem, the input must contain only one line, *i.e.*, only the three force components of a single atom in the unit cell. This should not happen, as mono-atomic systems do not exhibit atomic forces. When this list contains only one line, however, the array is cast as a one-dimensional array. In a later step, the code attempts to normalize this array along the second axis, which does not exist, causing the parser to crash. This error may occur in a regular code execution: The `exciting` code prints the respective section also for mono-atomic unit cells. If a user still performs atom relaxation, the respective section is written to the `exciting` output file, which would lead to a crash in the NOMAD system. We tested and verified this behavior with `exciting` version Boron.

The next example (error id "aims:logical:0") is a logical problem in the FHI-aims parser. The output of an FHI-aims calculation contains a list of so-called k-points with a set of values associated to them. If the fuzzer generates a file where these values are missing, the parsing fails because of an error in the `numpy.reshape()` routine. This indicates a logical error, where the program logic does not consider the case of missing values. This error was found because the fuzzer generates syntactically incorrect input, as the case of missing values is built into the grammar.

The last logical bug (error id "espresso:logical:0") we want to illustrate arises in the Quantum Espresso parser. Here, if the parsing of a section of the input file fails, the parser sets a default value that is not defined in the NOMAD data model. Interestingly, this error was triggered because of a syntactic inaccuracy in the grammar: A value in the parser input file that holds strictly positive integers is sometimes fuzzed as a negative integer, which causes the parser to return a wrong value.

Lastly, for the sake of completeness, we present a semantic and a syntactic issue. Both were observed for the FHI-aims parser. The semantic issue (error id "aims:semantic:8") illustrates another problem with default values. Due to semantic errors introduced by our grammar, the section that holds the number of spins could not be processed by the parser. Therefore, the parser used a default value for the number of spin channels to be used in the subsequent processing of the data. Using this default value, can thus lead to unexpected behavior: If an unrelated value in the parsed file triggers an error, the default value overwrites the number of spin channels, irrespective of its true value. We discovered this issue because input that should raise errors could be parsed without issues.

A syntactic issue (error id "aims:syntactic:0") arises when the FHI-aims output file contains a very specific formatting of the start time of the simulation. This is not correctly represented by our grammar. Thereby, one of the downsides of the usage of manually produced grammars is illustrated: The bugs that are produced may be false positives, because of false assumptions introduced by the interpretation of the employed samples. If we had a perfect formalism, which represented exactly what the program is supposed to do, any "bug" we find is indeed a bug. Because our formalism and generation are imperfect, we find "bugs" that are not real bugs, just results of the imperfections.

> Yes, issues detected also indicate actual bugs in the implementations of the parsers. However, separating physically nonsensical inputs from actual issues turns out to be misleading: Most of the severe issues were physically nonsensical, thus only concern corrupted output files of code runs or malicious intent.

6 | THREATS TO VALIDITY AND FUTURE WORK

In this paper, we rely on grammar based fuzzing to test parsers of the NOMAD Repository. In this context, we see the following threats to the validity of our work:

Internal:

Using fuzzing can potentially generate files and inputs that are nonsensical in the respective field. While these issues are actually issues of the program under test (i.e., uncaught malformed inputs), it is unlikely that such problematic outputs are generated by the codes in practice. We try to mitigate this problem by having an expert in the CMS field look over and classify each of the detected issues. Additionally, there is the potential of a sampling bias, as the selected samples do not represent all functionalities of the DFT codes and thus cannot test the parsers' behavior in all circumstances. Owed to the complexity of the samples, it is necessary to debug the grammar using the parsers themselves and TRIBBLE. This concerns, e.g., wrongly positioned quotation marks, so that strings are not correctly closed. Another threat to the validity of our results may be, that we execute the inputs in parallel: We might thereby also run into resource congestion issues where the Linux kernel may prematurely kill the parsing of a file that still could be correct. We tried alleviating this threat by running the experiments multiple times. Generating the grammars for each parser by hand from a set of input and output files has the potential of resulting in incorrect grammars. To mitigate this problem, we ask domain experts to check our generated grammars for sampling bias, i.e., the grammars being too close to particularities of the used samples. However, a verification study using a larger set of sample files should be done to ensure that our results are not just the product of bias. We leave this for future work.

External:

At present, about 45 different codes are supported by the NOMAD Laboratory, all corresponding parsers are in use. As such, we see as a threat that our work may not be generalizable. Nevertheless, having only worked on 4 of these parsers, this selection currently represents over 82% of all uploaded calculations[9]. We leave for future work, extending the work to cover an even higher percentage of the calculations. Further, the sampling-based generation of the grammars leads to very specific grammars. As noted above, these do not cover all functionalities of the DFT codes, and thus also not of the NOMAD parsers. We leave for future work extending the grammars to cover more diverse samples. If there are too few different researchers, using differing parameters, there may not be enough diversity across samples to generate accurate grammars. We envision a verification study that uses more different parameters as future work. Additionally, there is another potential source of sampling bias. We only use samples from the NOMAD repository where cubic Si or cubic C structures were investigated. These represent a small fraction of all calculations within the NOMAD repository. This is in line with the current validation strategies in the material sciences – for these two material structures the expected values are mostly well-known and can thus be used to find abnormal outliers. However, in our study we use the samples as a basis for generating the grammars: Therefore, we might miss other material structures that instrument the DFT code parsers differently. This needs to be investigated in follow-up studies.

7 | DISCUSSION AND CONCLUSIONS

In this work, we have applied grammar-based fuzzing to selected parsers of the NOMAD infrastructure. This was achieved by manual generation of grammars. We employed the program TRIBBLE to generate corresponding input data for the parsers. We collected the execution traces and parser outputs and employed an analysis pipeline where classification of the analyzed errors was performed by domain experts. Finally, we report all discovered issues to the NOMAD development team.

In the following, we want to share our observations of the up- and downsides of manual grammar generation. Starting with the drawbacks, the major concern is the long time that is required to write and harden the grammar. This is especially problematic if the samples that are used for generating the grammar are long and complex. For our application, this applies mostly to the VASP parser sample input files, which contain several million lines each. This complexity bears higher risk of human errors, especially if the grammar generation is performed by non-domain experts. If the grammar is created by domain experts, it is likely that biases are introduced, which are more threatening to the validity of the grammar.

Specific formatting choices (partially stemming from the 1990s) in the DFT codes are one typical source of human error. While the parsers require the specific format (float notation, time and date, etc.), a human might misunderstand and thus overgeneralize the input formatting. Furthermore, magic numbers introduce *breaking points* to the grammars, as any generated sample without the correct magic number leads to failures of the PUT at the exact same point. This prevents further and more insightful testing.

Manually created grammars also come with a number of major advantages. Typically, in grammar-based fuzzing, the same golden-truth grammar is used for both generating the parser, and testing the generated parser. This means, the capabilities of the parser-generator are tested and not the parser itself. This problem is solved by generating an independent grammar. In our

[9] As of 17 March 2021.

case, we avoided checking the NOMAD parsers' source codes before performing the fuzzing experiments, in order to minimize direct bias.

When the input format is more complex, generalization of input files into a grammar is not unique. Even the same developers of such a grammar might partition the exact same input files differently each time. Through our manual grammar creation, we provide a secondary point of view. The original parser developers may not have thought about this point of view before. Therefore, a major advantage of manually creating grammars, is the increased bug-finding capability.

In conclusion, our study uses manually created grammars to fuzz NOMAD parsers. We show, through our experiments, that the above-mentioned approach is indeed feasible. This answers RQ1. Therefore, it is our belief that it should be possible to use the same approach for research projects of a similar kind. More specifically, it is possible to extend our approach to other NOMAD parsers. In addition, the extension of grammar-based fuzzers to the CMS domain leads to novel and re-usable grammars. Because they are manually created, they are comprehensible to humans. This improves their value for future applications.

In our study, we demonstrate the ability of our approach to uncover issues in the selected CMS code parsers, thus answering RQ2. Due to the nature of these issues, we have reason to assume that other CMS code parsers may suffer from similar issues. Further studies on the remaining parsers should prove fruitful.

Separating issues caused by scientifically nonsensical inputs from actual issues, as we originally intended to do for RQ3, turns out to be overly simplified. All severe bugs in our study are indeed a consequence of nonsensical inputs. Even so, these bugs are severe threats to the availability of the service. While we originally intended to separate nonsensical and "real" inputs, the above threats show that this separation is counterproductive. The severity of the discovered issue should be evaluated regardless. We leave for future work to also test the correctness of parsed values.

In conclusion, we have found a number of critical bugs in the VASP parser, as well as a sizable number of less severe issues in the other three examples. Most of the issues stem from a lack of input sanitization. They have been reported to the NOMAD developers and have since been fixed. As publicly available web services may face corrupted or even malicious input, any data that is provided by users should be checked to contain only physically meaningful values. As a recommendation, taking inputs that come from users is always a potential danger. It is good to recheck that the input matches the expectations.

ACKNOWLEDGMENT

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 414984028 – SFB 1404 FONDA.

References

1. Draxl C, Scheffler M. The NOMAD Laboratory: from data sharing to artificial intelligence. *Journal of Physics: Materials* 2019; 2(3): 036001.

2. Hohenberg P, Kohn W. Inhomogeneous electron gas. *Physical Review* 1964; 136(3B): B864.

3. Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. In: ACM. ; 2011: 283–294.

4. Hafner J. Materials simulations using VASP—a quantum perspective to materials science. *Computer physics communications* 2007; 177(1-2): 6–13.

5. Blum V, Gehrke R, Hanke F, et al. Ab initio molecular simulations with numeric atom-centered orbitals. *Computer Physics Communications* 2009; 180(11): 2175–2196.

6. Giannozzi P, Baroni S, Bonini N, et al. QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of physics: Condensed matter* 2009; 21(39): 395502.

7. Gulans A, Kontur S, Meisenbichler C, et al. Exciting: a full-potential all-electron package implementing density-functional theory and many-body perturbation theory. *Journal of Physics: Condensed Matter* 2014; 26(36): 363202.

8. Wilkinson MD, Dumontier M, Aalbersberg IJ, et al. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific data* 2016; 3(1): 1–9.

9. Ghiringhelli LM, Carbogno C, Levchenko S, et al. Towards efficient data exchange and sharing for big-data driven materials science: metadata and data formats. *NPJ Computational Materials* 2017; 3(1): 1–9.

10. Miller BP, Fredriksen L, So B. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 1990; 33(12): 32–44. doi: 10.1145/96267.96279

11. Godefroid P, Levin MY, Molnar DA, others . Automated whitebox fuzz testing.. In: . 8. The Internet Society. ; 2008: 151–166.

12. Fioraldi A, Maier D, Eißfeldt H, Heuse M. AFL++ : Combining Incremental Steps of Fuzzing Research. In: Yarom Y, Zennou S., eds. *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*USENIX Association. USENIX Association; 2020.

13. Soremekun E, Pavese E, Havrikov N, Grunske L, Zeller A. Inputs from Hell Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* 2020.

14. Aschermann C, Frassetto T, Holz T, Jauernig P, Sadeghi A, Teuchert D. NAUTILUS: Fishing for Deep Bugs with Grammars. In: The Internet Society. The Internet Society; 2019.

15. Eberlein M, Noller Y, Vogel T, Grunske L. Evolutionary Grammar-Based Fuzzing. In: Springer. ; 2020: 105–120.

16. Hanford KV. Automatic Generation of Test Cases. *IBM Syst. J.* 1970; 9(4): 242–257. doi: 10.1147/sj.94.0242

17. Havrikov N, Zeller A. Systematically Covering Input Structure. In: IEEE. IEEE; 2019: 189–199