CHEMFUZZ: Large Language Models-assisted Fuzzing for Quantum Chemistry Software Bug Detection

Feng Qiu[1,2], Pu Ji[3], Baojian Hua[1,2], and Yang Wang[1,2]
[1]School of Software Engineering, University of Science and Technology of China, China
[2]Suzhou Institute for Advanced Research, University of Science and Technology of China, China
[3]Computer Science Department, The Johns Hopkins University, USA
bgkqf@mail.ustc.edu.cn pji7@jh.edu bjhua@ustc.edu.cn angyan@ustc.edu.cn

Abstract—Quantum chemistry software implements the first principle quantum computation and is indispensable in both scientific research and chemical industries. Any bugs in such software will lead to serious consequences, thus defeating its trustworthiness and reliability. However, bug detection techniques for such software have not been fully investigated. In this paper, to fill this gap, we propose a novel approach to fuzz quantum chemistry software with the aid of Large Language Models (LLMs). Our basic idea is utilize LLMs to mutate and generate syntactic and semantic valid input files from seed inputs, by proving valuable domain-specific knowledge of chemistry. With this basic idea, we have designed and implemented CHEMFUZZ, a fully automatic fuzzing framework to fuzz quantum chemistry software for bugs. Our evaluation of CHEMFUZZ leverages popular LLMs including GPT3.5, Claude-2, and Bart as test oracles to generate parameters to mutate inputs and analyze computation results. CHEMFUZZ detected 40 unique bugs, which have been classified and reported to developers, with a code coverage of 17.4%.

Keywords–Quantum Chemistry Software, Fuzzing, Large Language Models, Security Test

1. INTRODUCTION

We are entering an era of AI for science with scientific disciplines implemented on top of software infrastructures. Specifically, quantum chemistry software, computing equations and approximations derived from the postulates of quantum mechanics, have been extensively studied (*e.g.*, Gaussian[39], Siesta [40], Quantum Espresso [48], and Abacus [61]), with wide adoptions in diverse domains (*e.g.*, chemical industry [7], drug innovation [20], and material science [46]). Given the important applications of such software in science, they should be trustworthy and reliable.

Yet despite the pressing requirement of trustworthiness, bugs in quantum chemistry software are still inevitable, due to its considerable large code size and complex code logics. Furthermore, in recent years, in response to the strong demand for computational performance, an increasing number of quantum chemistry software are being deployed on cloud platforms [49] or supercomputers [54]. In such environments, any bugs might lead to not only program crashes, but also tenant environment corruptions. For example, a bug of wrong MPI number causes a crash of Siesta program during grid initialization [60]. As a result, securing the quantum chemistry software has become a pressing need.

Recognizing this need, a significant amount of studies have been conducted on scientific software, with diverse techniques employed. (*e.g.*, security testing [49] [51], fuzzing [76] [77] [19], probabilistic testing [74], and verification [75] [50]). Yet despite these research progresses, detecting bugs in quantum chemistry software remains challenging due to two unique characteristics of quantum chemistry software: **C1**): syntactic complexity; and **C2**): testing oracles. First, both input and output formats have complex syntactic structures, as they not only should represent interacted chemical parameters but also should obey certain spatial constraints (*e.g.*, symmetry). As a result, manually developing syntactically valid testing input is laborious and error-prone. Second, developing effective testing oracles for quantum chemistry software is challenging, as chemical domain knowledge is required in both input generation and result verification. Furthermore, developing effective testing oracles is difficult not only for software engineers but also even for chemistry scientists, due to the subtle properties of chemical molecules [70].

Our work. In this paper, to address these challenges, we present a Large Language Models-assisted fuzzing approach to detect bugs in quantum chemistry software. Our approach first generates random but valid input files, then feeds them to the quantum chemistry software as inputs. Next, our approach monitors program executions to record any potential crashes or illegal behaviors. Finally, our approach generated a report for execution consisting of crashes as well as testing metrics such as code coverage for subsequent analysis. Our approach is end-to-end and automated: during testing, it requires neither manual interventions nor user prompts.

To address the aforementioned first challenge C1, we utilized a syntactic mutation approach in which new syntactic correct input files are generated by syntactically mutating existing seeds. To address the challenge C2, we have leveraged transformer-based Large Language Models (LLMs) as our testing oracles, to exploit their potentials of overcoming domain knowledge problems.

To realize the whole process, we have designed CHEMFUZZ, a software prototype to implement our approach. CHEMFUZZ consists of two core modules: mutation and analysis. The mu-

tation module includes mutation operators, data padding, seed file validity checks, and collaborative efforts to create input file mutations. The analysis module encompasses code coverage collection, LLMs warning generation, a fitness function used for scoring, and a comprehensive assessment of the interest and effectiveness of seed files.

We have conducted extensive experiments to evaluate CHEMFUZZ in terms of effectiveness, performance of LLMs, and contributions of components. While our approach is general and can be built upon any quantum chemistry software and LLMs, we have conducted the evaluation on Siesta, a popular quantum chemistry software used by several thousand research all over the world [12], and used three most popular LLMs: GPT3.5 [43], Claude-2 [71], Bart [72] as the oracles. First, CHEMFUZZ is effective in detecting 40 unique bugs and 81 LLMs warnings, which have been classified and reported to developers. In the meanwhile, the maximum line coverage is 25,719, accounting for 17.43% of total lines, the function coverage is 939, accounting for 32.42% of all functions. Second, we compare and analyze the performance of three different LLMs: GPT3.5, Claude-2, and Bard, with respect to the validity of generated files and the average response time. Their response times are relatively close of about 10 seconds, while Claude-2 exhibits an impressive efficiency rate of 87.21% in generating higher-quality seed files. Third, we evaluate mutation operators, LLMs prompts, and fitness functions by analyzing their impacts on the effectiveness of CHEMFUZZ. Compared to its competitors, the current component has a stronger vulnerability detection capability and higher code coverage.

Contributions. To the best of our knowledge, this is the *first* work on utilizing a LLMs-assisted approach to fuzz quantum chemistry software for vulnerability detection. The main contributions of this paper are:

- **Infrastructure design.** We present the first security testing infrastructure for quantum chemistry software with a LLMs-assisted fuzzing approach.
- **Prototype implementation of CHEMFUZZ.** We implemented a prototype CHEMFUZZ to validate our design.
- **Evaluation of CHEMFUZZ.** We conducted extensive experiments to evaluate CHEMFUZZ in terms of effectiveness and performance.

Outline. The rest of our paper is organized as follows. Section 2 introduces the necessary background knowledge for our work. Section 3 explains the motivation and challenges. Section 4 and 5 presents the design of CHEMFUZZ and the evaluation, respectively. Section 6 discusses limitations and direction for future work. Section 7 introduces related work, and Section 8 concludes.

2. BACKGROUND

To be self-contained, in this section, we first present the necessary background knowledge for this work: quantum chemistry software, fuzzing, and large language models.

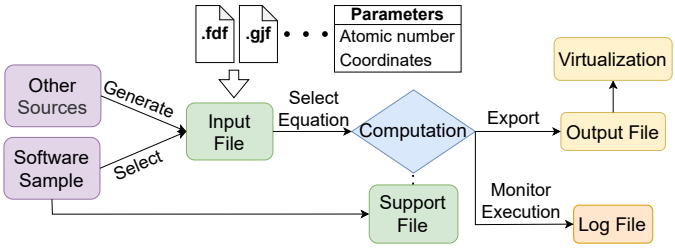

Figure 1: Typical workflow of quantum chemistry computation software.

2.1. Quantum Chemistry Software

Wide adoptions. Quantum chemistry software is widely adopted for research and industry purposes, such as Chemistry, Biology, Material Science, and Engineering. For example, Siesta, as one of the most important quantum chemistry software, currently owns several thousand users around the world and achieves more than thirteen thousand citations [41].

Workflow. Figure 1 presents a typical workflow for quantum chemistry software. First, input files (*e.g.*, FDF files, GJF files, etc.) are either selected from examples previously stored in software or generated from other sources. These input files contain parameters to pre-define and regulate computation. Next, if input parameters define an equation, the quantuam chemistry software will follow it; otherwise, the software will automatically select suitable equations based on input parameters. In addition, related support files (e.g. Pseudopotentials) from software samples may be used in helping the calculation. Subsequently, computational results will be exported into an output file. At the same time, execution behaviors monitored by software will also be recorded within a log file.

2.2. Fuzzing

Concept. Fuzzing is a popular and important software testing technique to detect vulnerabilities and bugs [30] [32]. The basic idea of fuzzing is to generating a gigantic amount of normal and abnormal inputs and feeding them into the targeted program to trigger unexpected program behaviors. Fuzzers can monitor the execution states of programs to detect possible issues such as memory corruption, program crashes, extreme resource usage, and input validation errors.

Category. Fuzzing techniques can be classified into two major categories based on input generations [11] [45]. The first method, called mutation-based fuzzing, collected a group of known inputs (called *seeds*) and perform random or heuristic modifications on them. Such changes maintain input validity but may cause new behaviors. The other method is generation-based fuzzing, which requires knowledge about input specifications to generate valid tests according to pre-defined configurations.

Feature. In contrast to other techniques, fuzzing displayed outstanding portability and accuracy. A fuzzing framework can be easily deployed and of good extensibility and applicability

[31]. It demands less knowledge of the target program and shows high accuracy since it is performed in real execution.

2.3. LLMs

Introduction. LLMs refer to a Transformer language containing a gigantic number of parameters and pre-trained by billions or trillions of available text data on the Internet [4] [42], (*e.g.*, Galactica[20], GPT-3.5 [43], and LLaMa [44]). Extending the scale of training allows LLMs to perform outstandingly in completing Natural Language Processing (NLP) tasks.

Emergent abilities. LLMs have shown diverse emergent abilities, which are not present in smaller models [42] [56]. First, experiments on a Massive Multi-task Language Understanding (MMLU) benchmark demonstrate LLMs' capability in multi-task Language understanding by accurately solving knowledge-based questions. Second, LLMs are well performed on unseen tasks. They can follow newly generated task instructions without explicit samples. Finally, with the chain-of-though (CoT) strategy [57], LLMs can solve such challenges by producing a series of intermediate reasoning steps before deriving the final answer.

Prompt engineering. Prompts quality influences LLMs outputs [8] [28] [64]. By using well-defined prompts, LLMs can indicate better communication interaction and enhance the ability to follow instructions for executing specific tasks. Existing studies [65] have demonstrated that the quality of LLMs outputs are significantly improved with optimized prompts.

3. Challenges and Motivation

In this section, we first present a detailed explanation of challenges caused by unique syntactic structures and semantics of quantum chemistry software (§ 3-A), then give our motivation (§ 3-B).

3.1. Challenge

Testing quantum chemistry software is challenging, due to the intense complexity of syntactic structures and semantics. Both input and output files consist of sophisticated chemical parameters, such as molecular geometry structure, calculation method, and other defined parameters used in the computation. Here, we use a Siesta input file in Figure 2 as an example to illustrate its syntactic and semantic challenges.

Syntactic challenges. The basic unit of the input file can be called a section, including a label and its data portion. Sections can be divided into two related categories: 1) Required section and optional section; and 2) Associated section and non-associated section.

Required and Optional. Required sections, as shown in Figure 2 basic part, define the fundamental molecular information for simulation, which are necessary and not removable. Simulations are not able to start without required sections. Oppositely, optional sections in the optional part define additional information, such as atomic moving orbitals. These sections are used to designate simulation methods or assist simulation to achieve more ideal results.

Associated and Non-associated. Most sections in input files are non-associated sections that define parameters independently.

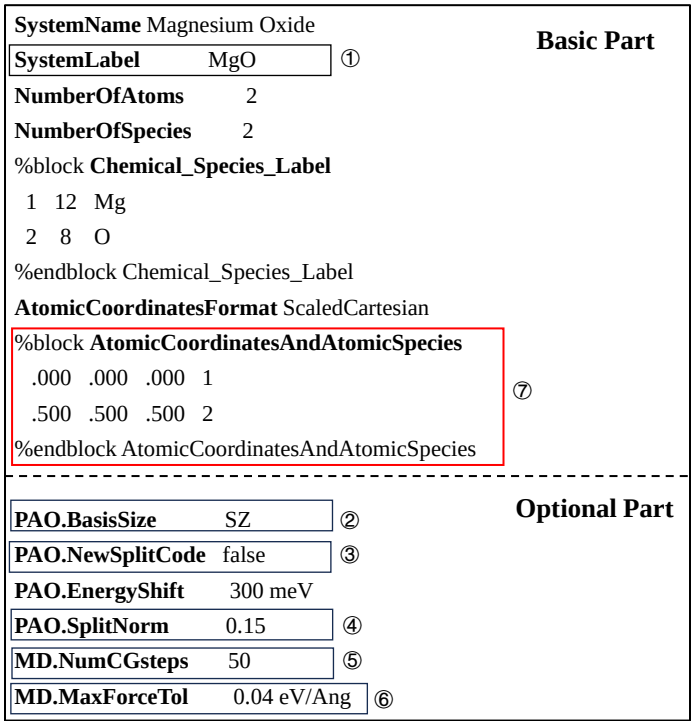

Figure 2: A Siesta sample input file illustrating the unique syntactic and semantic challenges to test quantuam chemistry software.

However, relationships exist between associated sections, like highlighted AtomicCoordinatesAndAtomicSpecies and AtomicCoordinatesFormat in Figure 2. The former defines atomic coordinates, and the latter regulates its format. Modifying each of them should affect correlated another section, which may eventually influence the entire simulation.

Semantics challenges. Semantics in data portions are also complicated due to various data types and the domain knowledge required in data selection.

Data. Data with a variety of types are defined in the data portion. Using Figure 2 as an example, data are represented within string, boolean, int, float, physical, and matrix formats. Although sections in dotted boxes 1 and 2 both use string data, the former data 'MgO' is a simple string describing the molecular system, and the latter data 'SZ' is selected from a pre-defined list of SZ, DP, SZP, and DZP. Boolean data in dotted box 3 is either a true or false statement but can be written in various forms (e.g., T/F, True/False, .true./.false., etc.). Float and int data, like dotted boxes 4 and 5, refer to real and integer numbers, and physical data in dotted box 5 is the combination of int or float data and physical units. Matrix data in dotted box 7 consist of multiple rows of different data, such as XYZ coordinates in float, atomic index in int, and element label in string.

Domain knowledge. Apart from the data type, domain knowledge is frequently required in selecting data values. These values should come from a reasonable domain, and rationality should be ensured. Otherwise, erroneous values will no longer

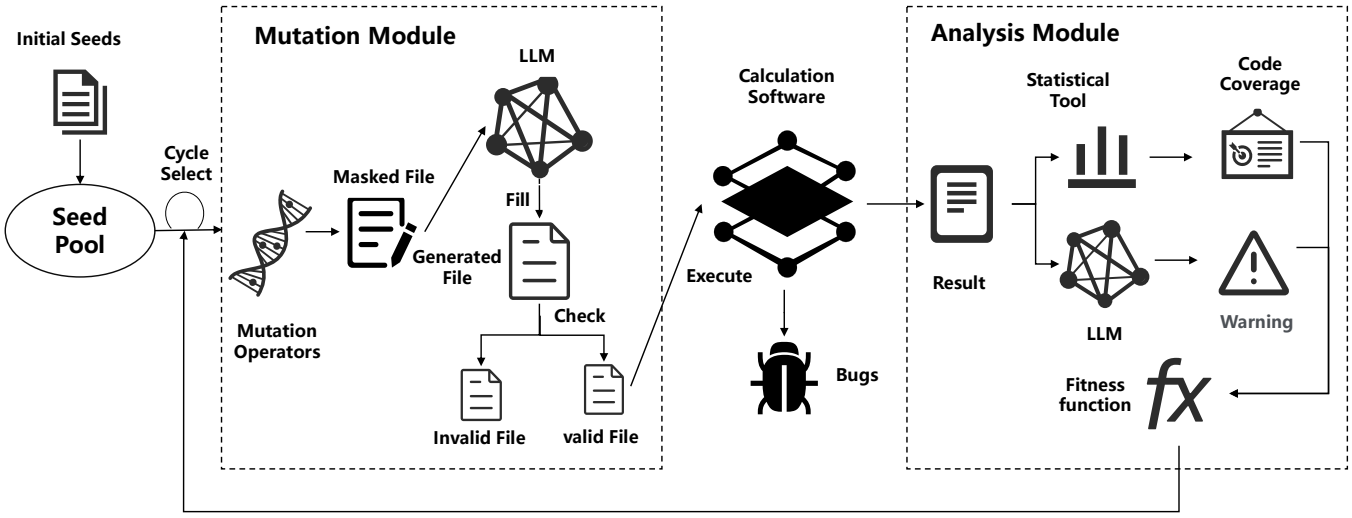

Figure 3: Overall framework of CHEMFUZZ.

hold physical significance, and the simulation cannot be implemented in the practical world—for example, the section PAO.EnergyShift in Figure 2, according to Siesta official documentation, is a standard for orbital-confining cutoff radius. It should be only selected with a small positive real number rather than a zero or negative value.

3.2. Motivation

After recognizing previous challenges, selecting suitable testing techniques and oracles becomes more essential. First, We aim to use mutation-based fuzzing to resolve structure complexity because such mutations on selected files maintain the greatest possible structure validity, as well as increase the possibility of finding bugs. Second, we consider LLM as a qualified oracle for semantic complexity and reduce domain knowledge barriers. Its gigantic training data and the emergent ability to follow instructions allow it to provide trustful and accurate data.

4. DESIGN

In this section, we present the design of CHEMFUZZ, by describing its overall architecture (§ 4-A), then its two main modules: the mutation (§ 4-B) and the analysis (§ 4-C).

4.1. Architecture

We present, in Figure 3, the overall framework of CHEMFUZZ. It employs a dynamic and mutation-based evolutionary fuzzing strategy and mutates the structure and data of seed files to generate effective new input files for quantum chemistry software. By executing these generated input files within the software, CHEMFUZZ detects potential vulnerabilities in crashes or endless loops. Subsequently, it analyzes the code coverage and warning information from the software's output to derive its fitness score. The testing process proceeds in iterations, and after each round, the files with higher fitness scores are retained to form the seed pool for the next round. Their fitness scores serve as weights when selecting files for mutation in the subsequent iterations.

The main components of CHEMFUZZ consist of two modules: the mutation module and the analysis module. The mutation module includes mutation operators, data filling using Large Language Models, and seed file validity checks. It's used to generate valid mutated input files for the target computation software. On the other hand, the analysis module involves code coverage collection, warning information generation based on LLMs, and fitness function calculation. It evaluates the ability of seed files to detect vulnerabilities and serves as a basis for selecting interesting and effective seed files. The above calculation process can be summarized as Algorithm 1. In this algorithm, we have designed a seed file elimination mechanism. After generating a certain number of files, only the top-performing ones are retained, allowing for the rational elimination of seed files with low code coverage and insufficient detection capabilities. The elimination process is controlled by two coefficients: the round factor f_1 and the retention factor f_2. Both f_1 and f_2 are typically positive numbers greater than 1, with f_1 being greater than f_2. According to the algorithm, if the initial number of seeds is k, after n elimination rounds, the number of seed files will be:

$$C = k * f_2^n$$

Therefore, by setting the round value n we can control the number of generated files, which will affect the depth of testing and the program's runtime. The exponential expansion of the file collection also enhances the overall coverage of fuzzing file testing, facilitating the rapid detection of potential vulnerabilities in the software.

Algorithm 1: Evolutionary fuzzing algorithm.

Input: Seed: the initial seed files,
Turn: the turn of evolution,
f_1: the round factor,
f_2: the retention factor.
Output: The generated files.

Function EvoFuzz (*Seed, Turn,* f_1, f_2) **:**
  SeedBank $\leftarrow$ *Seed;*
  InitializeFitness ();
  InitializeConfigure ();
  while $T \leq$ *Turn* **do**
    $N \leftarrow |SeedBank|$;
    for *i range in* $f_1 * N$ **do**
      CurrentSeed $\leftarrow$ *SelectByFitness(Seedbank);*
      Op $\leftarrow$ *RandomSelectMutationOp();*
      MaskedFile $\leftarrow$ *Mask(CurrentSeed, Op);*
      Cases $\leftarrow$ *LLM(MaskedFile);*
      ValidCases, InvalidCases $\leftarrow$ *Check(Cases);*
      Coverage, Result $\leftarrow$ *Execute(ValidCases);*
      WarningInfo $\leftarrow$ *LLM(Result);*
      FitnessScore $\leftarrow$ *FitnessFunction(ValidCases);*
      SeedBank $\leftarrow$ *SeedBank* $\cup$ *ValidCases;*
    SelectTopFitness(SeedBank, $f_2 * N$*);*
  return SeedBank;

4.2. Mutations

In the mutation module, our goal is to achieve diverse file mutations while satisfying the software input file constraints, to maximize the detection of program vulnerabilities. Therefore, our core approach is to apply mutation operators to the selected seed files to alter their file *structure* and leverage the large language model's excellent ability in structural text filling to modify the file *values*.

Mutation Operators. We define three types of mutation operators: *Add*, *Delete*, and *Modify*, randomly selecting an operator for mutation. Add and Delete are used to alter the structure of a file, while Modify is used to keep the file structure unchanged and only specify the need to modify a particular field within the file. During the mutation process, newly added sections from Add and the data portion of the selected sections from Modify are replaced with the "<Type-tag>" token, indicating the type of section in the tags such as "<integer>", "<float>" and "<matrix 3×2>", and a particular enumeration type, choosing from several optional values. The entire file mutation process is then completed by using a large language model to fill these tokens with generated data.

Add/Delete. The two operators are used to enhance the diversity of seed file structures. Under the condition, the Add mutation operator randomly selects a section label from all options and adds it to the original file. Specifically, if the chosen section belongs to an associated section, the associated sections are also added. The values of all newly added sections

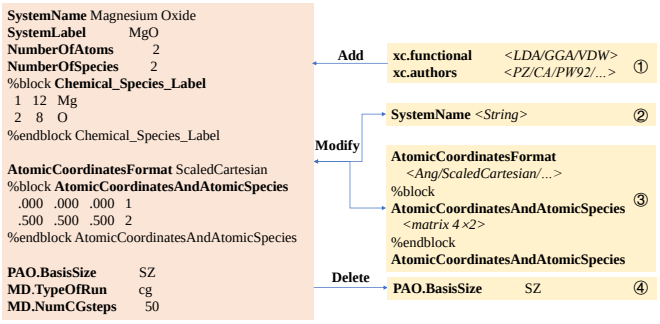

Figure 4: The effects of operators applied to sections.

TABLE I: Mutation operators applied in different sections.

	Add	Delete	Modify
Required & Associated	×	×	○
Optional & Associated	○	○	○
Required & No-associated	×	×	–
Optional & No-associated	–	–	–

are replaced with the "<Type-tag>" token in masked files. Similarly, the Delete mutation operator randomly selects one from the existing optional sections of the current file and removes it and all associated sections.

Modify. We parse the syntax structure of the file and record all sections, their type and associated relationships in a data structure. Then, we randomly select a section from the seed file, if the chosen section belongs to an associated section, the associated sections are also selected. The data portions of all the selected sections are marked as "<Type-tag>". Existing an operator that does not alter the file structure but only modifies the numerical values is beneficial for preserving some interesting seed files and detecting more program vulnerabilities.

Based on the way the mutation operators work, the system needs to know the list of labels for sections in the target software and their associate relationships. This task is not difficult, just a bit tedious, for the latest version, as there are approximately 200 sections in Siesta and approximately 250 in Quantum Espresso with different labels [12] [48]. We will collect and record this information from the software documentation into a configuration file.

File Generation. Table I presents the performance of different operators applied to various types of sections (discussion in 3-A) in quantum chemistry computation programs. "×" indicates that the operator cannot be applied to this section, "-" indicates that the operator will only be applied to this section, and "○" indicates that the operator will be applied to all fields associated with this section. All the efforts are dedicated to fulfilling the requirements of the input file format.

Figure 4 shows the effects of three operators applied to different types of sections. Four cases demonstrate the effects of "add" on an optional & associated section, "modify" on a required & associated section and a required & non-associated

section, and ”delete” on an optional & non-associated section, respectively. These cases confirm the operators' ability to mutate the results of seed files effectively.

After randomly applying a mutation operator to produce masked files, we utilize Generative Pre-trained Language Models such as GPT and Claude-2 to generate new files by filling in the masked-out locations.

When using LLMs, a well-defined prompt can enhance and strengthen the capability of a large language model [66]. For a sophisticated text generation task, the prompt should be correctly structured into the following components: the motivation of the task, output customization, specific rules and guidelines of the task, the key ideas, and example implementations [25] [26] [27]. Based on these theories, we have implemented our text generation prompt:

At the end of the task, a masked input file for the quantum chemistry calculation software Siesta-4.1.5 is provided in the format of “text“, with masked parts labeled using the <>tags.

Your task is to fill in the masked parts using text and generate an input file that can actually be run in the Siesta software, and respond with the filled-in file.

Tag Rules:

1. The first tag type indicates the data type, such as <boolean>, <float>, <matrix 3×2>and etc. You need to fill in specified parameters of the indicated data type.

2. The second tag type provides several options, such as <Ang/ScaledCartesian>. You need to use one of the options provided.

Filling task requirements:

1. The generated file should retain the unmasked parts of the input file and fill in all the masked parts.

2. The generated file should comply with the requirements of version 4.1.5 of Siesta, as specified in the Siesta usage document.

3. The generated file should adhere to the requirements of the quantum chemistry field in terms of calculation methods, physical and chemical properties, crystal structure, and other parameters.

4. The generated file should be filled with the extreme and reasonable values you consider as soon as possible.

5. The generated file should differ as much as possible from the files stored in your memory.

Response Format:

1. Your response should only include the filled-in result of the input file provided below in the format of “text“.

2. You must not include any additional text information.

The prompt provides detailed descriptions of various aspects of input text generation for quantum chemistry software, enabling LLM to directly generate interesting and contextually appropriate bi-directional completions for the intended task. Program errors often occur near extreme or boundary values [68] [69]. The prompt specifically emphasizes generating such values to identify potential software vulnerabilities effectively. Additionally, although the prompt is customized for Siesta 4.1.5, the similarity of input files among different software versions allows for minor modifications to apply to other versions or software for text-filling tasks.

Finally, the criterion for judging the validity of input files is their ability to pass the validation based on the tested software file format and enter the numerical computation phase. Except only for specific seed files that exhibit inconsistent behavior between the file parser and the document description.

4.3. Analysis

In the analysis module, Our goal is to analyze the results of seed files in the calculation software and calculate their fitness scores. We firstly measure code coverage during the execution process (§ 4-C), and secondly employ a large language model to detect any abnormal data in the execution results of the valid mutation file (§ 4-C). Finally, we determined the fitness score of the mutation files based on these findings (§ **??**), providing a basis for selecting subsequent seed files.

Coverage Collection. For large-scale C++/Fortran programs like quantum chemistry calculation software, using the *gcov* [58] and *lcov* [59] component from the GCC compilation suite to perform code coverage analysis is often a good practice.

We insert the ”-fprofile-arcs -ftest-coverage” flags during the compilation of the software source code using g++/gfortran. After the software execution, gcda files showing code coverage are generated, and the lcov tool is used to produce the final required code coverage report. In this context, we use line coverage, denoted as C, as a measure of the software's ability to detect program vulnerabilities, which will be utilized in the subsequent fitness function.

LLM Warnings. Similar to section 4-B, we leverage large language models to assist in identifying potential abnormal data in the calculation results of the chemical software. Building upon the principles of prompt engineering and tailored for Siesta version 4.1.5, we have developed a specific prompt to check for warnings in the calculation results:

The core part of the output of the quantum chemical computing software Siesta-4.1.5 is provided at the end of the task in the format of “‘text“‘.

Your task is to determine whether the software output has obvious anomalies.

Task requirements:

1: Only very obvious exceptions need to be flagged in the task.

2: Incorporating knowledge from the field of quantum chemistry, considering the physical significance of each value, numeric parameters in the result need to be examined to determine if it falls outside the normal data range.

Result requirements: The task results meet the requirements:

1: The result value is Abnormal/Normal.

2: When the answer is Normal, your answer does not need to include any additional information.

3: When the answer is Abnormal, your answer gives the warning text information and its reason in WarningLocation.

The output format requirements are as follow json file:

{”Result”: ”Abnormal”,
”WarningLocation”: [{
”located”: ”text1”,
”reason”: ”Why are to lead the warnning.”

}]}

The prompt defines the task objectives and response format, allowing us to easily analyze the warnings' positions and their underlying causes from the returned JSON object. Meanwhile, We define the number of warnings as W, which indicates the likelihood of the program generating abnormal results, serving as a crucial basis for the fitness of seed files.

Fitness functions. In the evolutionary algorithm of ChemFuzz, selecting unique and interesting files for mutation through the fitness function is crucial [32] [33]. Considering both the program's execution status (S), line coverage (C), and the warning size (W) when using the file as an output, we define a fitness function (**F**) to evaluate the effectiveness of seed files (f):

$$\mathbf{F}(f) = \begin{cases} C + W/10, & \text{if } S = 0 \\ 1, & \text{if } S = 1 \end{cases}$$

One small detail pertains to calculating the execution status (S), which is straightforward and trivial. It involves merely recording whether the file triggers a crash or enters an infinite loop during program execution. In such cases, S = 1; otherwise, S = 0.

The rationale for using it as the fitness function mainly includes the following three points: 1) Evaluating the execution status (S) based on whether errors occur provides the most direct evidence for determining the effectiveness of legitimate seed files in discovering program vulnerabilities. For instance, an input file that passes the format validation but causes the calculation software to enter an infinite loop or crash should be assigned a higher fitness score. 2) When the calculation software produces normal output results, line coverage (C) indicates the number of covered code lines by seed files, while the error warnings (W) provided by the large language model suggest the possibility of anomalies or errors in the results. Combining line coverage and LLM warnings as benchmarks for seed files' vulnerability discovery ability is highly reasonable. 3) The quantity of LLM warnings (W) typically falls within the range of 1 to 10, while line coverage (C) always ranges between 0 and 1. They often differ by one order of magnitude. Assigning a weight of 0.1 to W effectively balances the impact of the two factors.

Overall, employing this fitness function enables a comprehensive assessment of seed files' performance in detecting program vulnerabilities. Based on this, we assign fitness scores to the seed files, which serve as weights for the selection of file mutation.

5. Evaluation

In this section, we present experiments to evaluate ChemFuzz. First, we present the research questions (§ 5-A) guiding the experiments, and experimental setup (§ 5-B). Second, we show the experimental results in terms of the effectiveness (§ 5-C), LLMs performance (§ 5-D), and components contributions (§ 5-E). Finally, we discuss a case study of real-world vulnerabilities ChemFuzz detected (§ 5-F).

5.1. Research Questions

By presenting the experimental results, we mainly investigate the following research questions:

RQ1: Effectiveness. Is ChemFuzz effective in bug detection in real-world programs?

RQ2: LLMs performance. In the task of generating input files for chemical calculation software, what are the performance of different large language models?

RQ3: Components contributions. What are the contributions of the key components of ChemFuzz to its effectiveness?

5.2. Experimental Setup

Test program. We employed Siesta 4.1.5 [12] [40] [41] as the software under test, a stable version widely and extensively used in diverse fields for a considerable period. In addition, We collected 35 input files from the examples and tests of the Siesta software to serve as initial seeds for our study. These files are simple yet deterministic, providing a stable foundation for subsequent file mutations.

Configure. In the configuration file, we collected data on 76 labels corresponding to the sections in Siesta, including their data types, optionality, and associate relationships, as documented in the official documentation. The round factor f_1 and the retention factor f_2 are set to 2.0 and 1.5.

Environment. All the experiments and measurements are performed on a server with one 48 physical Intel E7-4830 core CPU and 64 GB of RAM running Ubuntu 18.04. We used the built-in gcov [58] and lcov [59] tools from gfortran 7.5.0 to collect code coverage information.

5.3. RQ1: Effectiveness

We conducted multiple rounds of experiments with ChemFuzz using different large language models, including GPT-3.5 [43], Claude-2 [71], and Bart [72].

ChemFuzz, in version 4.1.5 of the Siesta calculation software package, has detected 40 bugs that were previously unknown to Siesta developers. Table II presents the statistics of these bugs detected by ChemFuzz, which led to program crashes or endless loops. Various bugs were identified, including 5 file parse error, 3 stack overflow, 9 integer overflow, 4 endless Loop condition errors, 14 parameter validation errors, and 3 bugs that could not be attributed to a specific cause. We have reported all the bugs to the developers of the Siesta software.

Table III show different LLMs provided 81 warnings regarding the software's output results. From the table, two conclusions can be drawn: 1) The majority of LLMs' warnings are related to excessively large or small values of physical parameters in the computed results, exceeding reasonable ranges. This deserves attention. 2) There is a significant difference in the number of warnings provided by different LLMs, which may be attributed to varying sensitivity to data anomalies. It is necessary to consider their conclusions collectively.

Figure 5 presents a warning from GPT3.5, and this case originates from Siesta simulation calculations using the DFT method for MgO at an environmental temperature of

TABLE II: Summary of detected bugs.

Running state	Bug categories	Size
Crash	File parse error	5
	Stack Overflow	3
	Integer Overflow	9
	Others	2
Endless	Loop condition error	4
	Parameter validation error	14
	Others	3
Total	6	40

TABLE III: LLMs warning of software calculation result.

LLMs Warning	GPT-3.5	Claude-2	Bard	Total
Non-negative	4	0	1	5
Special value	6	2	2	10
Matrix asymmetric	2	1	0	3
Abnormally large	12	2	13	27
Abnormally small	15	3	10	28
Others	2	1	3	6
Total	41	10	30	81

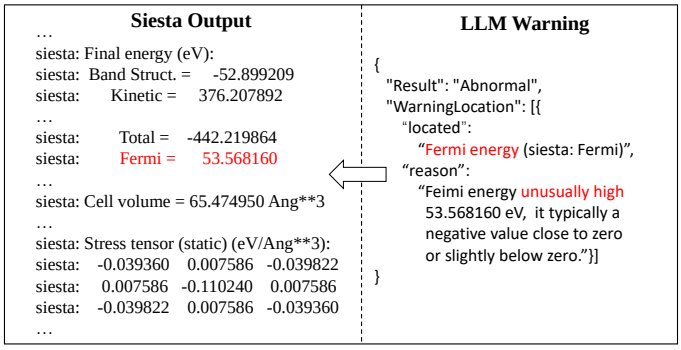

Figure 5: A warning to generated by GPT-3.5.

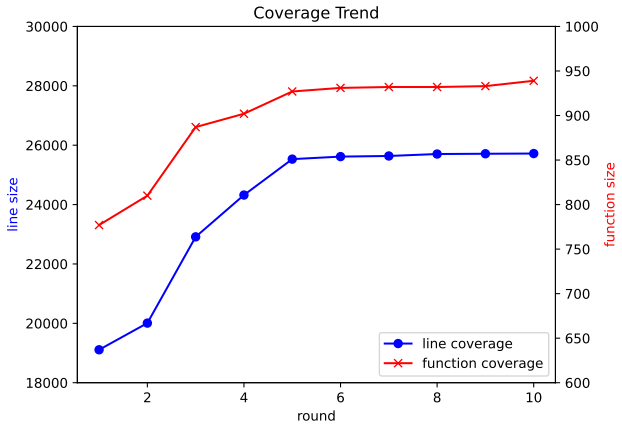

Figure 6: Coverage trend in the multi-round testing.

$500000K$. As a truth, in such high-energy environments, conventional DFT methods often lead to distortions [66]. The Language Model-based Learning system is capable of detecting when the output results contain an excessively large Fermi level, reaching $53.568160eV$, a value that is typically either a complex number or a very small positive number [67]. As a consequence, an alert is issued regarding the output results. This case serves as evidence for the effectiveness of LLM warnings.

These warnings merit the attention of both users and developers of quantum chemistry software. It is recommended to consult material databases or employ alternative software for comparative analysis to confirm the accuracy of the output results when necessary.

ChemFuzz achieved a maximum line coverage of 25,719, accounting for 17.43% of the total lines, and the highest function coverage of 939, accounting for 32.42%. Figure 6 illustrates the growth trend of code coverage during 10 retention rounds of mutation testing. Considering the extensive code of more than 140,000 lines and diverse computational methods of Siesta, our testing framework is evidently effective.

Summary: From Siseta 4.1.5, ChemFuzz detected 40 bugs and 81 LLMs warnings, the maximum line coverage is 25,719 LoC, accounting for 17.43% of total lines; the function coverage is 939, accounting for 32.42% all functions, demonstrates its effectiveness in detecting real-world code vulnerabilities.

5.4. RQ2: LLMs Performance

To answer RQ2, a series of multi-round experiments were conducted with ChemFuzz to demonstrate the performance of different LLMs, including GPT-3.5, Claude-2, and bard, in generating Siesta input files. These experiments took a total of 14 hours.

Table V presents the size of generated files and average generation times of all LLMs. In our research, we have observed that Claude-2 is capable of generating quantum chemistry software seed files with higher quality, achieving an impressive efficiency rate of 87.21%. The response times of the three models are quite similar, all around 10 seconds. Therefore, the Cladue-2 model is the most suitable and effective large model for our complex text generation task among the models evaluated.

In the text generation task, we found that the main reason for the invalidity of seed files lies in the inability to understand the task requirements and objectives sometimes, resulting in issues such as missing output files, incorrect formatting, and reproducing the input as output. These abnormal conditions warrant further optimization and improvement by LLM developers. As users, when setting prompts reasonably, we should prioritize selecting more proficient large models such as Claude-2 to accomplish the quantum chemistry software file generation task.

TABLE IV: The ability of LLMs under different prompts.

Model	GPT3.5				Claude-2				Bard			
	Valid	All	Rate	Coverage	Valid	All	Rate	Coverage	Valid	All	Rate	Coverage
ChemFuzz	176	213	82.63%	23176	155	178	87.08%	22741	98	127	77.17%	22512
No type tag	149	222	67.12%	22792	147	190	77.37%	22714	83	109	76.14%	22410
No boundary note	180	216	83.33%	22841	151	188	85.63%	22140	102	122	83.61%	21090

TABLE V: The performance of LLMs to generate quantum chemistry software input files.

LLM	Generated Files per API			Time per File (s)	
	Valid	All	Percentage	Valid	All
GPT3.5	132	161	81.99%	9.14	9.37
Claude-2	150	172	87.21%	10.05	9.72
Bard	122	148	82.43%	8.94	8.04

Summary: Claude-2 achieved the highest quality and effectiveness of 87.21%. The response times of three models are similar, all around 10 seconds.

5.5. RQ3: Evaluation of Key Components

Mutation Operators. In this study, we conducted an experiment of 33 hours to evaluate the impact of each mutation operator. The outcomes of generating approximately two hundred files are presented in Table VI, where we display the results after systematically removing each mutation operator from ChemFuzz. Notably, the employment of the complete set of mutation operators yielded a line coverage of 17.4%, which is significantly higher than the versions with Modify and Add removed. Although removing Delete had minimal impact on coverage, the runtime increased threefold in the complete version. Currently, the mutation operators demonstrate the best performance.

Fitness function. We compared our default fitness function against alternative functions, including random selection ($Random$), code coverage-based weighting (C), and a simple function ($C+W$) in experiments. Except for random functions, the fitness score remains unchanged at 1 for seed files with abnormal execution status(S).

Table VII summarizes the results. Our fitness function has achieved a balance between the effects of C and W, enabling all interesting files to attain higher fitness scores. As a result, in the statistics of experiments, the current function ($C + W/10$) demonstrated the highest code line coverage of 17.2% and the highest function coverage of 32.1%.

LLMs Prompt. To investigate the impact of large language model (LLMs) prompts on the framework's vulnerability detection capability, We conducted experiments using different LLMs on the same initial seed for an equal duration of time, while comparing the current prompt used in ChemFuzz with prompts that do not include type tags and or boundary indications. The experiments took a total of 19 hours.

TABLE VI: Evaluation of mutation operators.

OP	Runtime duration(h)	Line Coverage	
		Lines	Percentage
ChemFuzz	6.2	25678	17.4%
-Modify	4.6	23465	15.9%
-Add	4.2	19233	13.0%
-Delete	18.3	25788	17.4%

Table IV presents the statistical data of the experimental results, the coverage in the data is line coverage. Compared to the modified prompts, the current prompt text generation capability demonstrates a balance between code coverage and generation effectiveness across all three large models. Furthermore, the removal of type annotations significantly decreases effectiveness in GPT3.5 and Claude2, while eliminating reminders for large models to fill in boundary values results in a decline in code coverage for all large models.

Summary: Compared to other counterparts, the current key components exhibit higher effectiveness, demonstrating superior code coverage in testing.

5.6. Case Study

Figure 7 shows a bug in Siesta 4.1.5 where there is a lack of validation for a specific value in the MaxSCFIterations section. The calculation program reads the MaxSCFIterations value from the fdf file, which represents the number of iterations for the self-consistent field (SCF) calculation. The value is initially read into the fdf_integer variable and then used as the number of iterations nscf, without any validation.

Because the SCF calculation is a crucial step in quantum chemistry calculations [78], this bug causes the entire calculation process to terminate without producing any results if MaxSCFIterations is set to 0 in the input file. This behavior prevents the proper calculation of physical parameters and methods, rendering the calculation process non-functional.

This is clearly not in line with the expected behavior of the program. ChemFuzz detected this exceptional situation and reminds quantum chemistry software developers to pay more attention to input parameter validation and enhance program robustness.

6. Discussion

In this section, we discuss the limitations of this work, as well as directions for future work.

TABLE VII: Comparison of fitness function effectiveness.

Function	Line Coverage		Function Coverage	
	Lines	Percentage	Functions	Percentage
ChemFuzz	25376	17.2%	930	32.1%
C	24121	16.3%	892	30.8%
$C+W$	23725	16.1%	888	30.7%
$Random$	23109	15.7%	861	29.7%

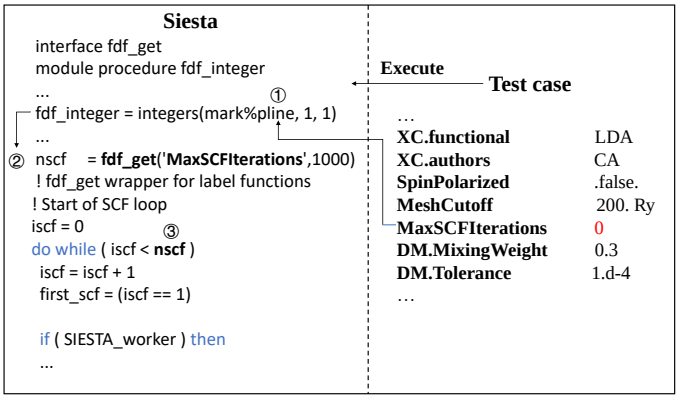

Figure 7: A real world bug of loop parameter validation detected by ChemFuzz, in Siesta 4.1.5.

Quantum Chemistry Software. Though experimental results indicated that our selection of quantum chemistry software - Siesta is effective, there exists several other influential software (e.g., Gaussian [39], Quantum Espresso [48], Abacus [61]). Despite software differences, we speculate that software selection only affects performance since they share similar workflows. We will continuously apply ChemFuzz on different quantum chemistry software in the future.

Oracle. ChemFuzz leverages GPT-3.5, Claude-2, and Bard as oracles in input generation and output verification. Although its accuracy, to some extent, is ensured by billions or even trillions of training data available on the Internet, parameters provided by GPT-3.5 may not completely ignore scientific domain-knowledge problems. Additionally, Lin et al. [65] research shows that LLMs occasionally follow popular misconceptions and provide fraudulent answers. Hence, we assume utilizing a LLM specially trained for science fields and scientific knowledge bases is essential and will help improve the effectiveness of ChemFuzz.

Fuzzing. As presented in our design, ChemFuzz contains mutation operators to modify seed files selected from seed-banks, which most initial seeds came from examples in quantum chemistry software. Existing research [31] [45] shows that generation-based fuzzing commonly results in better code coverage to trigger more vulnerabilities. As a result, we realize including generation operator features becomes a prime work orientation. While building ChemFuzz, we already gained the partial ability to deconstruct input files' structures to help us further enhance our fuzzing model. Despite ChemFuzz having proved its effectiveness through evaluation, we leave the generation-based feature as an important direction for future work.

Other vulnerabilities. ChemFuzz has detected certain vulnerabilities. Nevertheless, there are other types of vulnerabilities. Particularly, it is vital to inspect concurrency bugs in parallel computing of quantum chemical computation[62]. Besides, though ChemFuzz solve the complexity of input file structure, it is currently incapable of analyzing intricate output file structure. We leave these two as crucial future work.

7. Related Work

We classified related work into three categories: fuzzing, Large Language Model, and security testing.

Fuzzing. Fuzzing test techniques have been improved and are heavily used in different realms and applications. Guo et al. [14] proposed DLFuzz as a special fuzzing framework for leading deep learning systems to expose malicious behaviors. Deng et al. [15] built TitanFuzz to first use large language models to generate input programs that satisfy grammar syntax and constraints, which is used for fuzzing deep learning libraries. Luo et al. [16] perform fuzzing techniques in Industrial Control System (ICS) protocols by collecting valuable packets that trigger patch coverage and dividing them into pieces to form test cases with better quality for further fuzzing tests. Atlidakis et al. [17] introduce Pythia as a fuzzer that augments grammar-based fuzzing with coverage-guided feedback and a learning-based mutation strategy for REST API fuzzing. Pham et al. [18] define innovative mutation operators and introduce smart grey-box fuzzing to utilize structural specification of seed to form new input files so that deeper vulnerabilities could be discovered. Wang et al. [19] present QuanFuzz by generating search-based test cases as input to perform grey-box fuzz testing for quantum programs.

Our work, which differs from recent research, is an evolutionary fuzzing model specialized for quantum chemistry computational software.

LLM. Large Language Model is an emerging language model pre-trained by billions of text data on the internet and extensively used in research fields. Taylor et al. [20] present a large language model that could be used to save, combine, and predict scientific knowledge, and is pre-trained by a significant amount of specialized scientific datasets, papers, reference material, and other sources. Singh et al. [21] leverage LLM to rate robots' potential actions and even generate robot commands with no external instructions. They propose a programmatic LLM prompt structure to acquire that. Beltagy et al. [22] release a pre-trained BERT [26] language model called SciBERT, which brings up a better performance on downstream scientific Natural Language Processing tasks. Ouyang et al. [23] implement aligning language models with fine-tuning according to human feedback, leading to improvements in truthfulness and a decrease in irritational output. By using collected datasets to fine-tune GPT-3 and iterating the previous process for further reinforcement, they achieved resulting models called InstructGPT.

However, a key difference with our work is that we leverage pre-trained large language models as an oracle both in generating input parameters within an acceptable domain of quantum chemistry and in verifying the accuracy and rationality of computation results.

Security testing. Recently, security testing for software has become favored. Corteggiani et al. [24] claim Inception, a novel security testing framework for embedded firmware through generating and merging LLVM bitcode. Schieferdecker et al. [25] introduce MBST, which automatically generates systematic documentation of security test objectives, cases, and suites. Bozic et al. [38] provide an approach with experimental results to automatically test chatbots on web applications. Siboni et al. [73] proposed a security testbed framework to discover vulnerabilities and device breakdown for The Internet of Things (IoT) ecosystem.

In contrast to previous research, in our security test framework, we combine fuzzing techniques and LLM, two popular techniques given a lot of attention.

8. Conclusion

In this paper, we present ChemFuzz, the first approach to fuzzing quantum chemistry software via large language models. ChemFuzz utilized an evolutionary fuzzing method consisting of two components: a mutation to generate new program input files based on LLMs, and an analysis to obtain a fitness score. The prototype implementation and evaluation have shown that ChemFuzz can effectively detect vulnerabilities in real-world quantum chemistry software. Overall, the work in this paper represents a crucial first step towards vulnerability detection in quantum chemistry programs, benefiting the overall community of software testing and quantum chemistry software development.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work is partially supported by the National Natural Science Foundation of China (No.62072427, No.12227901), the Project of Stable Support for Youth Team in Basic Research Field, CAS (No.YSBR-005), Academic Leaders Cultivation Program, USTC.

References

[1] Liu R, Kumar A, Chen Z, et al. A predictive machine learning approach for microstructure optimization and materials design[J]. Scientific reports, 2015, 5(1): 11551.

[2] Yang X, Lo D, Xia X, et al. Deep learning for just-in-time defect prediction[C]//2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE, 2015: 17-26.

[3] Deng Y, Xia C S, Peng H, et al. Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models[C]//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023). 2023.

[4] Deng Y, Xia C S, Peng H, et al. Fuzzing deep-learning libraries via large language models[J]. arXiv preprint arXiv:2212.14834, 2022.

[5] Fioraldi A, D'Elia D C, Coppa E. WEIZZ: Automatic grey-box fuzzing for structured binary formats[C]//Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis. 2020: 1-13.

[6] Shao Y, Molnar L F, Jung Y, et al. Advances in methods and algorithms in a modern quantum chemistry program package[J]. Physical Chemistry Chemical Physics, 2006, 8(27): 3172-3191.

[7] Obot I B, Macdonald D D, Gasem Z M. Density functional theory (DFT) as a powerful tool for designing new organic corrosion inhibitors. Part 1: an overview[J]. Corrosion Science, 2015, 99: 1-30.

[8] Reynolds L, McDonell K. Prompt programming for large language models: Beyond the few-shot paradigm[C]//Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems. 2021: 1-7.

[9] Shimazaki T, Hashimoto M, Maeda T. Developing a high-performance quantum chemistry program with a dynamic scripting language[C]//Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering. 2015: 9-15.

[10] Adams S, de Castro P, Echenique P, et al. The Quixote project: Collaborative and Open Quantum Chemistry data management in the Internet age[J]. Journal of cheminformatics, 2011, 3: 1-27.

[11] "The Fuzzing Book, tools and techniques for generating software tests,": https://www.fuzzingbook.org/

[12] "Siesta,": https://departments.icmab.es/leem/siesta/

[13] Deglmann, Peter, Ansgar Schäfer, and Christian Lennartz. "Application of quantum calculations in the chemical industry—An overview." International Journal of Quantum Chemistry 115.3 (2015): 107-136.

[14] Guo J, Jiang Y, Zhao Y, et al. Dlfuzz: Differential fuzzing testing of deep learning systems[C]//Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018: 739-743.

[15] Deng Y, Xia C S, Peng H, et al. Fuzzing deep-learning libraries via large language models[J]. arXiv preprint arXiv:2212.14834, 2022.

[16] Luo Z, Zuo F, Shen Y, et al. ICS protocol fuzzing: Coverage guided packet crack and generation[C]//2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020: 1-6.

[17] Atlidakis V, Geambasu R, Godefroid P, et al. Pythia: grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations[J]. arXiv preprint arXiv:2005.11498, 2020.

[18] Pham V T, Böhme M, Santosa A E, et al. Smart greybox fuzzing[J]. IEEE Transactions on Software Engineering, 2019, 47(9): 1980-1997.

[19] Wang J, Gao M, Jiang Y, et al. QuanFuzz: Fuzz testing of quantum program[J]. arXiv preprint arXiv:1810.10310, 2018.

[20] Tan Y X, Zhang F, Xie P P, et al. Rhodium (III)-catalyzed asymmetric borylative cyclization of cyclohexadienone-containing 1, 6-dienes: an experimental and DFT study[J]. Journal of the American Chemical Society, 2019, 141(32): 12770-12779.

[21] Singh I, Blukis V, Mousavian A, et al. Progprompt: Generating situated robot task plans using large language models[C]//2023 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2023: 11523-11530.

[22] Beltagy I, Lo K, Cohan A. SciBERT: A pretrained language model for scientific text[J]. arXiv preprint arXiv:1903.10676, 2019.

[23] Ouyang L, Wu J, Jiang X, et al. Training language models to follow instructions with human feedback[J]. Advances in Neural Information Processing Systems, 2022, 35: 27730-27744.

[24] Corteggiani N, Camurati G, Francillon A. Inception:System-Wide security testing of Real-World embedded systems software[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 309-326.

[25] Schieferdecker I, Grossmann J, Schneider M. Model-based security testing[J]. arXiv preprint arXiv:1202.6118, 2012.

[26] Devlin J, Chang M W, Lee K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. arXiv preprint arXiv:1810.04805, 2018.

[27] Santu S K K, Feng D. TELeR: A General Taxonomy of LLM Prompts for Benchmarking Complex Tasks[J]. arXiv preprint arXiv:2305.11430, 2023.

[28] White J, Fu Q, Hays S, et al. A prompt pattern catalog to enhance prompt engineering with chatgpt[J]. arXiv preprint arXiv:2302.11382, 2023.

[29] Madaan A, Tandon N, Clark P, et al. Memory-assisted prompt editing to improve gpt-3 after deployment[J]. arXiv preprint arXiv:2201.06009, 2022.

[30] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.

[31] Li J, Zhao B, Zhang C. Fuzzing: a survey[J]. Cybersecurity, 2018, 1(1):

1-13.
[32] Liang H, Pei X, Jia X, et al. Fuzzing: State of the art[J]. IEEE Transactions on Reliability, 2018, 67(3): 1199-1218.
[33] Rawat S, Jain V, Kumar A, et al. VUzzer: Application-aware Evolutionary Fuzzing[C]//NDSS. 2017, 17: 1-14.
[34] DeMott J, Enbody R, Punch W F. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing[J]. BlackHat and Defcon, 2007.
[35] Odena A, Olsson C, Andersen D, et al. Tensorfuzz: Debugging neural networks with coverage-guided fuzzing[C]//International Conference on Machine Learning. PMLR, 2019: 4901-4911.
[36] Xie T, Tillmann N, De Halleux J, et al. Fitness-guided path exploration in dynamic symbolic execution[C]//2009 IEEE/IFIP International Conference on Dependable Systems & Networks. IEEE, 2009: 359-368.
[37] Wei M, Huang Y, Yang J, et al. Cocofuzzing: Testing neural code models with coverage-guided fuzzing[J]. IEEE Transactions on Reliability, 2022.
[38] Bozic J, Wotawa F. Security testing for chatbots[C]//IFIP International Conference on Testing Software and Systems. Cham: Springer International Publishing, 2018: 33-38.
[39] "Gaussian,": https://gaussian.com/
[40] Soler J M, Artacho E, Gale J D, et al. The Siesta method for ab initio order-N materials simulation[J]. Journal of Physics: Condensed Matter, 2002, 14(11): 2745.
[41] García A, Papior N, Akhtar A, et al. Siesta: Recent developments and applications[J]. The Journal of chemical physics, 2020, 152(20).
[42] Zhao W X, Zhou K, Li J, et al. A survey of large language models[J]. arXiv preprint arXiv:2303.18223, 2023.
[43] Brown T, Mann B, Ryder N, et al. Language models are few-shot learners[J]. Advances in neural information processing systems, 2020, 33: 1877-1901.
[44] Touvron H, Lavril T, Izacard G, et al. Llama: Open and efficient foundation language models[J]. arXiv preprint arXiv:2302.13971, 2023.
[45] Miller C, Peterson Z N J. Analysis of mutation and generation-based fuzzing[J]. Independent Security Evaluators, Tech. Rep, 2007, 4.
[46] He Q, Yu B, Li Z, et al. Density functional theory for battery materials[J]. Energy & Environmental Materials, 2019, 2(4): 264-279.
[47] Siboni S, Sachidananda V, Meidan Y, et al. Security testbed for Internet-of-Things devices[J]. IEEE transactions on reliability, 2018, 68(1): 23-44.
[48] "quantum-espresso,": https://www.quantum-espresso.org/
[49] Francis R R. Reliability of cloud computing in Quantum Chemistry calculations[C]//2012 International Conference on Cloud Computing Technologies, Applications and Management (ICCCTAM). IEEE, 2012: 119-120.
[50] Honarvar S, Mousavi M R, Nagarajan R. Property-based testing of quantum programs in Q#[C]//Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. 2020: 430-435.
[51] Mendiluze E, Ali S, Arcaini P, et al. Muskit: A mutation analysis tool for quantum software testing[C]//2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021: 1266-1270.
[52] Liu J, Lin J, Ruffy F, et al. Nnsmith: Generating diverse and valid test cases for deep learning compilers[C]//Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. 2023: 530-543.
[53] Fu J, Liang J, Wu Z, et al. Griffin: Grammar-free DBMS fuzzing[C]//Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022: 1-12.
[54] Hu W, Qin X, Jiang Q, et al. High performance computing of DGDFT for tens of thousands of atoms using millions of cores on Sunway TaihuLight[J]. Science Bulletin, 2021, 66(2): 111-119.
[55] Wei J, Tay Y, Bommasani R, et al. Emergent abilities of large language models[J]. arXiv preprint arXiv:2206.07682, 2022.
[56] Hendrycks D, Burns C, Basart S, et al. Measuring massive multitask language understanding[J]. arXiv preprint arXiv:2009.03300, 2020.
[57] Wei J, Wang X, Schuurmans D, et al. Chain-of-thought prompting elicits reasoning in large language models[J]. Advances in Neural Information Processing Systems, 2022, 35: 24824-24837.
[58] "gcov-tool—an Offline Gcda Profile Processing Tool," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[59] "LCOV - the LTP GCOV extension," https://ltp.sourceforge.net/coverage/lcov.php.
[60] "Siesta issue on gitlab, grid initialization crashes with floating point exceptions for certain numbers of MPI tasks," https://gitlab.com/siesta-project/siesta/-/issues/170.
[61] "Abacus," http://abacus.ustc.edu.cn/main.htm
[62] Alkan M, Pham B Q, Hammond J R, et al. Enabling Fortran standard parallelism in GAMESS for accelerated quantum chemistry calculations[J]. Journal of Chemical Theory and Computation, 2023.
[63] Liu P, Yuan W, Fu J, et al. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing[J]. ACM Computing Surveys, 2023, 55(9): 1-35.
[64] Zhou Y, Muresanu A I, Han Z, et al. Large language models are human-level prompt engineers[J]. arXiv preprint arXiv:2211.01910, 2022.
[65] Lin S, Hilton J, Evans O. Truthfulqa: Measuring how models mimic human falsehoods[J]. arXiv preprint arXiv:2109.07958, 2021.
[66] Foulkes W M C, Mitas L, Needs R J, et al. Quantum Monte Carlo simulations of solids[J]. Reviews of Modern Physics, 2001, 73(1): 33.
[67] "The Materials Project,": https://next-gen.materialsproject.org/
[68] Myers G J, Badgett T, Thomas T M, et al. The art of software testing[M]. Chichester: John Wiley & Sons, 2004.
[69] Ramachandran M. Testing software components using boundary value analysis[C]//2003 Proceedings 29th Euromicro Conference. IEEE, 2003: 94-98.
[70] Sims C. CIF2WAN: A Tool to Generate Input Files for Electronic Structure Calculations with Wannier90[J]. arXiv preprint arXiv:2006.12647, 2020.
[71] "Claude-2," https://www.anthropic.com/index/claude-2
[72] "Bard," https://bard.google.com
[73] Siboni S, Shabtai A, Tippenhauer N O, et al. Advanced security testbed framework for wearable IoT devices[J]. ACM Transactions on Internet Technology (TOIT), 2016, 16(4): 1-25.
[74] Krishnaswamy, Smita, Igor L. Markov, and John P. Hayes. "Tracking uncertainty with probabilistic logic circuit testing." IEEE Design & Test of Computers 24.4 (2007): 312-321.
[75] Liu, J., Zhan, B., Wang, S., Ying, S., Liu, T., Li, Y., ... & Zhan, N. (2019). Formal verification of quantum algorithms using quantum Hoare logic. In Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31 (pp. 187-207). Springer International Publishing.
[76] Wang, J., Gao, M., Jiang, Y., Lou, J., Gao, Y., Zhang, D., & Sun, J. (2018). QuanFuzz: Fuzz testing of quantum program. arXiv preprint arXiv:1810.10310.
[77] Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., ...& See, S. (2019, July). Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 146-157).
[78] Verma P, Truhlar D G. Status and challenges of density functional theory[J]. Trends in Chemistry, 2020, 2(4): 302-318.