Large Language Model powered Test Driver Generation for High-performance Computing Library

Ziran He*
College of Computer Science and Technology
National University of Defense Technology
Changsha, China
hzr@nudt.edu.cn

Guofeng Zhang*
College of Computer Science and Technology
National University of Defense Technology
Changsha, China
zhangguofeng16@nudt.edu.cn

Meixi Liu*
College of Computer Science and Technology
National University of Defense Technology
Changsha, China
liumeixi@nudt.edu.cn

Zhenbang Chen*
College of Computer Science and Technology
National University of Defense Technology
Changsha, China
zbchen@nudt.edu.cn

Abstract

High-performance computing (HPC) relies on hardware-specific libraries like BLAS and FFTW, whose correctness is often validated via differential testing. However, manual test cases limit coverage, and automated techniques such as symbolic execution and fuzzing struggle with HPC libraries due to implicit parameter constraints and the difficulty of generating valid, large-scale inputs, leading to inefficiency and poor automation.

This paper presents an LLM-powered approach to test driver generation that overcomes these challenges. By classifying interface parameters and modeling their semantic constraints, our method automatically generates executable, structurally valid drivers tailored to HPC libraries. This enables scalable input generation for comprehensive testing of complex APIs. We evaluate our approach on OpenBLAS and FFTW using symbolic execution and fuzzing, showing significant improvements in test automation and coverage compared to manual and baseline automated efforts.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

Keywords

High-performance computing library, Test driver generation, LLM, Symbolic execution, Fuzzing

ACM Reference Format:
Ziran He, Guofeng Zhang, Meixi Liu, and Zhenbang Chen. 2026. Large Language Model powered Test Driver Generation for High-performance Computing Library. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-NIER '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3786582.3786815

*Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
ICSE-NIER '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2425-1/26/04
https://doi.org/10.1145/3786582.3786815

1 Introduction

The growing computational power and complexity of modern applications have made high-performance computing (HPC) indispensable in scientific computing [21], engineering simulation [22], artificial intelligence [4], and big data analytics [23]. To harness modern hardware, developers rely on highly optimized computing libraries—such as BLAS [12], FFTW [8], and Intel MKL [5], that are tightly coupled with specific processor architectures. These mathematical libraries underpin critical software stacks, making their correctness and stability essential to system-wide reliability.

Testing these libraries is particularly challenging as they incorporate complex hardware-specific optimizations and large-scale floating-point computations [1, 6, 11, 20, 24]. Comprehensive testing requires generating vast, diverse inputs to cover various computational optimization paths and validating numerical precision, a task infeasible manually. Consequently, automated test generation using techniques like symbolic execution [2, 19] and fuzzing [3, 7] has become essential.

However, both approaches face significant obstacles in HPC libraries. Symbolic execution struggles with large data inputs and complex parameter dependencies, leading to state explosion and poor path coverage. Fuzzing, meanwhile, treats all inputs uniformly, resulting in inefficient exploration due to a vast, sparsely valid input space. In both cases, domain knowledge is needed to guide analysis, specifically to identify critical parameters and encode valid input constraints.

While prior work addresses API usage in general software [3, 9, 10, 14], these methods are ill-suited for HPC libraries, where semantic constraints are deeply tied to data shapes and mathematical semantics. To overcome these limitations, test drivers must be carefully designed to reflect library-specific usage patterns and parameter relationships—an effort that currently requires deep expertise and manual labor. Meanwhile, existing fuzzing test driver generation tools [15, 26] leverage LLMs to learn and generate legitimate drivers, yet their primary application focuses on API control logic and fails to address the critical parameter relationships required in HPC contexts.

To address this challenge, we propose an LLM-powered approach for generating test drivers for HPC libraries. Our method leverages the domain knowledge embedded in large language models [18]

```
void cblas_sgemm(
    const enum CBLAS_ORDER Order,      //Memory layout
    const enum CBLAS_TRANSPOSE TransA, //transpose A
    const enum CBLAS_TRANSPOSE TransB, //transpose B
    const int M,      //row of op(A) and C
    const int N,      //column of op(B) and C
    const int K,      //column of op(A) and row of op(B)
    const float alpha,//Scalar multiplier for op(A)*op(B)
    const float  *A,  //matrix A
    const int lda,    //Leading dimension of A
    const float  *B,  //matrix B
    const int ldb,    //Leading dimension of B
    const float beta, //Scalar multiplier for C
    float  *C,        //matrix C
    const int ldc);   //Leading dimension of C
```

Figure 1: The CBLAS_SGEMM function in BLAS

to extract implicit constraints from interface definitions and open-source implementations. By classifying parameters and encoding semantic relationships, our technique constructs valid, efficient test drivers that enable more effective symbolic execution and fuzzing.

The main contributions of this paper include:

- An LLM-based constraint extraction method that identifies parameter relationships in HPC library interfaces, improving the validity and reliability of generated test drivers.
- A parameter-aware driver generation strategy that automates test driver construction across diverse computing libraries, enabling end-to-end test generation.
- An evaluation on real-world libraries (OpenBLAS, FFTW) demonstrating the effectiveness and practicality of our approach.

2 Motivation

In the HPC libraries, traditional program analysis-based testcase generation techniques are significantly less effective. HPC libraries exhibit two key characteristics: (1) inputs are typically large-scale tensors, and (2) APIs have numerous parameters that are often interdependent. These features lead to two major challenges for program analysis.

First, analyzing large tensor inputs drastically degrades the performance of program analysis due to the sheer size and complexity of the data. Second, there exist strict semantic constraints between tensor structures and other API parameters, constraints that program analysis tools, lacking domain-specific knowledge, cannot effectively reason about. As a result, they generate many invalid or infeasible inputs, leading to inefficient exploration and poor testing coverage.We use the example from the BLAS library in Figure 1 to illustrate the key characteristics of HPC libraries.

2.1 Large-scale tensor data

In high-performance computing operators, the scale of tensor data can often be substantial. In the field of image processing, tensor data mostly conforms to fixed standard sizes such as `4 × 4`, `16 × 16`, or `64 × 64` [16]. However, in domains like deep learning and scientific computing, the number of elements within a tensor can reach tens of thousands [17]. With such data volumes, traditional unit testing methods require analyzing all data elements, which is undoubtedly very resource-consuming.

Let's take the CBLAS_SGEMM operator [13] as an example in Figure 1. This is a general matrix multiplication operation and performs the following calculation:

$$C = \alpha \times op\,(A) \times op\,(B) + \beta \times C \tag{1}$$

As shown in the Figure 1, the parameter list includes three matrices A, B, and C. The dimensions of these matrices are determined by M, N, and K, and they are accessed using `lda`, `ldb`, and `ldc` as their respective strides. For example, performing an operation on a single-precision floating-point matrix of size 1024×1024 would require 4 MB of memory space.

For symbolic execution, fully symbolizing a 1024x1024 matrix leads to significant scalability challenges. Any write operation on such a matrix introduces over 100,000 array theory constraints via SMT encoding, far beyond the capacity of SMT solvers, often resulting in timeouts or memory exhaustion. Moreover, symbolic execution must maintain symbolic memory states in a lookup table, where each matrix element corresponds to a symbolic expression. With the large number of such expressions, the efficiency of querying and managing the symbolic state degrades dramatically.

Fuzzing, on the other hand, relies critically on throughput: higher throughput enables more program executions and broader exploration of the input space within a given time. However, mutating large-scale matrices incurs substantial overhead during input generation, directly reducing throughput. Additionally, increased memory consumption per execution limits the number of parallel fuzzing instances, further constraining scalability.

Crucially, internal data variations within tensors typically affect only the output values and do not alter the control flow or execution path in HPC libraries. As a result, tracking fine-grained symbolic or concrete states inside tensors yields diminishing returns in terms of coverage. Therefore, our key insight is that tensor-typed input parameters should be decoupled in test driver generation, avoiding deep program analysis while still enabling effective testing.

2.2 Parameter constraint

There exist strict constraint relationships between the tensor data structure and other interface parameters. These parameters determine how data within the tensor is read during the concrete implementation of the interface. Failure to satisfy these constraints may lead to crashes.

Taking the CBLAS_SGEMM operator as an example, by reviewing the relevant documentation, we can identify the following constraint relationships among its parameters *Order*, *A*, *M*, *N*, and `lda`:

$$\begin{cases} A.size\,() = lda \times M;\ lda \geq N; if\ Order = CblasRowMajor \\ A.size\,() = lda \times N;\ lda \geq M; if\ Order = CblasColMajor \end{cases} \tag{2}$$

As we can see from the above constraints, the size of Matrix A is jointly determined by four parameters, and there are additional constraints among these parameters, which undoubtedly complicate our analysis process.

In the absence of prior knowledge, applying symbolic execution to explore the path space of the CBLAS_SGEMM operator faces a fundamental challenge: the inability to recover array sizes $A.size()$. Existing symbolic execution engines support symbolic values within memory regions (e.g., array elements) but do not symbolize the metadata of pointer objects themselves, such as allocated size. As a result, $A.size()$ is typically treated as a constant rather than a symbolic value, preventing the solver from accurately inferring tensor dimensions. Moreover, constructing inter-parameter

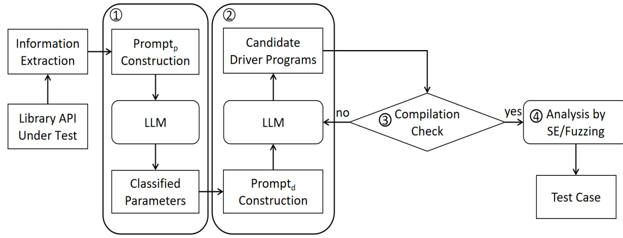

Figure 2: The workflow of our approach

constraints (e.g., between M, N, and K) requires prolonged analysis before any valid input is produced, leading to poor analysis efficiency and long initialization phases.

Similarly, fuzzing must generate matrices of appropriate sizes based on M, N, and K. While prior work uses DSL-based input generators or custom mutators, these require manually crafted drivers and domain-specific logic. Without access to such domain knowledge, purely random mutation has an extremely low probability of generating valid inputs, especially for operators involving multiple interdependent matrices. Standard fuzzers lack awareness of the semantic relationships between parameters and their corresponding tensors, making it difficult to maintain structural validity during mutation and severely limiting their effectiveness.

Without providing program analysis techniques with domain-specific knowledge of HPC libraries, significant effort is wasted in attempting to infer implicit parameter constraints from code. Therefore, our key insight is that these constraints should be explicitly encoded in the test driver a priori, enabling early and effective pruning of infeasible regions in the input space.

3 Approach

To address the challenges in generating effective test drivers for HPC libraries, we propose a LLM-powered framework that leverages the LLM's understanding of open-source computing libraries and its code generation capabilities. To mitigate the instability of LLMs in complex, multi-step synthesis tasks, we decompose driver generation into two sequential phases: (1) parameter classification, and (2) constraint-aware driver generation.

As illustrated in Figure 2, the workflow begins by extracting the target API's interface information, including the library name, function signature, and parameter list, from the HPC library under test. This information is used to construct the first prompt, $Prompt_p$, which guides the LLM to classify each interface parameter into one of several semantic categories. Based on the classification results, along with interface metadata and driver specifications, we then construct a second prompt, $Prompt_d$, to guide the LLM in generating an executable test driver.

The generated driver undergoes a compilation check. If compilation fails, the system iteratively refines the prompt (e.g., by including error feedback) and regenerates the driver, up to a predefined retry threshold. Once a compilable driver is produced, it is integrated into symbolic execution or fuzzing frameworks to generate test inputs and explore program behavior.

3.1 Parameter classification

In HPC libraries, interface parameters are typically symbolized (in symbolic execution) or mutated (in fuzzing) to generate diverse inputs. However, we observe that (1) parameters exhibit distinct semantic types, and (2) strong constraints exist between them. Ignoring these distinctions leads to invalid inputs and poor testing

```
1 fftw_plan fftw_plan_dft(
2     int rank,
3     const int *n,
4     fftw_complex *in,
5     fftw_complex *out,
6     int sign,
7     unsigned flags);
```

Figure 3: The fftw_plan_dft function in FFTW

effectiveness. To address this, we attempted various classification methods, such as by parameter type and by function. However, these approaches could not adequately handle complex-valued operations and variable dimensions. To improve generalizability, we classify HPC API parameters into four categories based on their computational roles:

- **Computational dimension parameters**: indicate the dimensionality of the current computation such—parameters may not always be present;
- **Control and coefficient parameters**: primarily include storage format, data size, coefficients, and similar values;
- **Dimension-dependent control and coefficient parameters**: pertain to data sizes, coefficients, etc., that are controlled by dimensionality and are often formatted as variable-length arrays;
- **Data parameters**: correspond to the tensor data itself.

Notably, the size of data parameters is typically determined by control and coefficient parameters, forming a hierarchical dependency. Taking the CBLAS_SGEMM function in Figure 1 as an example, matrices A, B, and C are extracted and treated as data parameters. And the M, N, K are control and coefficient parameters. As shown in Figure 3, there is an additional constraint layer for the fftw_plan_dft interface in FFTW: the size of n is determined by the value of $rank$. Due to this constraint, n belongs to the dimension-dependent control and coefficient parameters, while $rank$ is classified as a computational dimension parameter.

These semantic categories enable us to guide the LLM in inferring and encoding meaningful constraints during driver generation. For instance: (1) Computational dimension parameters determine the size of dimension-dependent control parameters, requiring size-aware symbolic modeling in symbolic execution; (2) Control and coefficient parameters govern the layout and size of data parameters, which must be respected during fuzzing to prioritize mutation on semantically influential inputs.

By incorporating this structured parameter taxonomy and its associated constraints, our framework guides the LLM to generate syntactically valid, semantically meaningful, and testing-effective drivers—significantly improving the quality and efficiency of automated test generation for HPC libraries.

3.2 Driver generation

Based on the parameter classification results, we combine the parameter categories, library name, interface name, and a driver program template into a new prompt to guide the LLM in generating candidate driver programs.

We have observed that in HPC libraries, the data parameters of an interface typically do not influence the program's branch conditions—these values participate in computation but not in control-flow decisions. This observation is derived by tracking and analyzing actual implementations in real computing libraries. Parameter constraints are modeled as two types: (1) constraints between control parameters and dimensions, and (2) constraints

Table 1: Coverage comparison conducted in the open-source libraries OpenBLAS and FFTW

Library	UT	SE	SE-LLM	Fuzz	Fuzz-LLM
OpenBLAS/level3	45.2%	52.7%	74.3%	69.6%	75.1%
OpenBLAS/generic	68.8%	71.3%	77.0%	73.4%	77.1%
FFTW/dft	65.7%	66.4%	71.6%	67.4%	71.9%
FFTW/kernel	59.7%	60.3%	62.7%	61.9%	63.4%

between control and data parameters. Therefore, we have designed two main parts for driver generation.

We first guide the LLM to represent the aforementioned dimension-dependent control and coefficient parameters. The array sizes of these parameters are determined by the computational dimensions and whether the operation involves complex numbers. We provide the LLM with either a symbolic execution interface or a fuzz testing data reading interface to determine the data to be analyzed and generate the corresponding call format. For example, for the code in Figure 3, we would symbolize the *rank* parameter, generate an array *n* whose size is determined by the value of *rank*, and then symbolize the internal data of *n*.

Since data parameters do not affect subsequent branch conditions, they can be processed independently: generated rapidly through random or fuzzing methods, then initialized directly according to their size. The LLM can assist in determining this scale, which typically relates to dimensions and control parameters.

For example, in the case of Figure 1, we initialize an array A filled with zeros based on the values of $Order$, M, N, and lda.

Through these two steps, we obtain an executable driver that fully considers the categories of control and data parameters, as well as the implicit constraints between parameters.

4 Evaluation

To automatically generate test cases for different computing libraries in high-performance computing scenarios, we designed an LLM-assisted test driver generation strategy. To evaluate the effectiveness of our method, we conducted assessments on selected interfaces of OpenBLAS and FFTW. However, due to the difficulty in accurately determining invocation relationships between interfaces and pinpointing the impact scope of tested interfaces, we compared our results with all unit tests [25] available in the open-source libraries. In essence, our experimental data consists of the coverage from both unit tests and our method.

Our experiments utilized the ChatGPT-4 model to test a subset of interfaces in OpenBLAS and FFTW. The experimental results are presented in Table 1. The symbolic execution tool employed in this paper is FDSE [27], while the fuzzing tool utilized is AFL++ [7]. We conducted experiments on both symbolic execution and fuzz testing with and without our method. Among these, the approach which does not employ our method uses manually generating test drivers to analyze all parameters without imposing additional constraints. The time budget for each driver generation is limited to one minute. Meanwhile, the time limit for analyzing a single API using both symbolic execution and fuzz testing is set to one hour.

We measured the time required to construct valid test drivers. Our approach successfully generated 155 and 155 valid test drivers for symbolic execution and fuzzing, respectively, within 53 and 57 minutes. For comparison, we manually implemented an equivalent number of test drivers (without encoding inter-parameter constraints), which took 183 and 237 minutes, respectively. The significant reduction in effort demonstrates the efficiency and practical potential of leveraging large language models for automated test driver generation for HPC libraries.

As shown in Table 1, the test drivers generated by our approach successfully execute under both symbolic execution and fuzzing frameworks, achieving significantly higher code coverage than both the original unit tests and the baseline drivers. In OpenBLAS, our method improved coverage by 29.1%/8.2% and 29.9%/3.7% across two major source directories, respectively. In FFTW, we observed coverage gains of 5.9%/3.0% and 6.2%/3.7%, respectively.

Compared to the manually generated baseline, our approach achieves average coverage improvements of 8.65% under symbolic execution and 3.8% under fuzzing. These results demonstrate that incorporating semantic parameter classification and constraint modeling significantly enhances test quality.

Notably, the improvement is more pronounced in symbolic execution. This is expected: symbolic execution is highly sensitive to input constraints, and invalid or unconstrained inputs lead to path explosion or early termination. By guiding the LLM to model inter-parameter dependencies (e.g., array sizes derived from dimension parameters), our method produces drivers that respect these constraints, enabling deeper exploration of feasible execution paths.

We have analyzed limitations in the current approach. In HPC libraries, hardware-dependent compilation flags can modify macros at compile time, causing certain code branches to remain unexecuted and reducing test coverage. Additionally, LLM exhibit hallucination during parameter classification, such as misclassifying complex-valued parameters as dimension-dependent. In cases of hallucination, the relevant outputs are regenerated.

5 Conclusion

This paper presents an LLM-powered test driver generation strategy for HPC libraries, capable of automatically generating drivers for both symbolic execution and fuzz testing tailored to different computing libraries. Through our approach, we leverage the domain knowledge of LLM to effectively conduct selective testing based on constraint relationships between parameters. This avoids the generation of large-scale data while ensuring the validity of the test programs. Simultaneously, it enhances the quality of the generated test cases, enabling coverage of more code branches.

6 Future Plans

To advance this work, we plan to evaluate more libraries beyond OpenBLAS and FFTW to assess generalizability. Currently focused on generating matrix dimensions, we will also generate complex matrix data for differential testing to uncover real faults. A key challenge is that compile-time macros control certain code paths, necessitating systematic variation of compilation flags and tighter integration of build configuration with test generation. Furthermore, our LLM-based framework requires refinement, especially in automatically verifying the correctness and effectiveness of generated test drivers.

Acknowledgments

This research was supported by National Key R&D Program of China (No. 2024YFF0908003) and the NSFC Program (No. 62172429).

References

[1] Patricia S. Abril and Robert Plant. 2007. The patent holder's dilemma: Buy, sell, or troll? *Commun. ACM* 50, 1 (Jan. 2007), 36–44. doi:10.1145/1188913.1188915

[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[3] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. 2023. Hopper: Interpretative Fuzzing for Libraries. *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security.* https://api.semanticscholar.org/CorpusID:261582917

[4] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Bryan Catanzaro, and A. Ng. 2013. Deep learning with COTS HPC systems. In *International Conference on Machine Learning.* https://api.semanticscholar.org/CorpusID:8604637

[5] Intel Corporation. 2024. Intel Math Kernel Library (Intel MKL). (2024). https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html Version 2024.0.

[6] Jack Dongarra, Laura Grigori, and Nicholas Higham. 2020. Numerical algorithms for high-performance computational science. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378, 2166, 20190066. doi:10.1098/rsta.2019.0066

[7] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20).*

[8] Matteo Frigo and Steven G. Johnson. 2024. FFTW: Fastest Fourier Transform in the West. (2024). https://www.fftw.org/ Accessed: 2024-09-24.

[9] Harrison Green and Thanassis Avgerinos. 2022. GraphFuzz: Library API Fuzzing with Lifetime-aware Dataflow Graphs. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 1070–1081. https://api.semanticscholar.org/CorpusID:248660124

[10] Kyriakos K. Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *USENIX Security Symposium.* https://api.semanticscholar.org/CorpusID:207905588

[11] Jie Jia, Yi Liu, Guozhen Zhang, Yulin Gao, and Depei Qian. 2022. Software approaches for resilience of high performance computing systems: a survey. *Frontiers Comput. Sci.* 17, 174105. https://api.semanticscholar.org/CorpusID:254616676

[12] Martin Kroeker, Xianyi Zhang, Qian Wang, et al. 2024. OpenBLAS: An Optimized BLAS Library. (2024). https://github.com/OpenMathLib/OpenBLAS Accessed: 2024-09-24.

[13] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 1–10. https://api.semanticscholar.org/CorpusID:3242763

[14] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2023. Prompt Fuzzing for Fuzz Driver Generation. *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security.* https://api.semanticscholar.org/CorpusID:266690829

[15] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2023. Prompt Fuzzing for Fuzz Driver Generation.

[16] Himanshu Kumar Mittal, Avinash Chandra Pandey, Mukesh Saraswat, Sumit Kumar, Raju Pal, and Garv Modwel. 2021. A comprehensive survey of image segmentation: clustering methods, performance parameters, and benchmark datasets. *Multimedia Tools and Applications* 81, 35001 – 35026. https://api.semanticscholar.org/CorpusID:231856162

[17] Shaoyuan Ou, Kaiwen Xue, Lian Zhou, Chun-Ho Lee, Alexander Sludds, Ryan Hamerly, Ke Zhang, Hanke Feng, Yue Yu, Reshma Kopparapu, Eric Zhong, Cheng Wang, Dirk Robert Englund, Mengjie Yu, and Zaijun Chen. 2025. Hypermultiplexed integrated photonics–based optical tensor processor. *Science Advances* 11. https://api.semanticscholar.org/CorpusID:279155948

[18] Hammond A. Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2021. Examining Zero-Shot Vulnerability Repair with Large Language Models. *2023 IEEE Symposium on Security and Privacy (SP)*, 2339–2356. https://api.semanticscholar.org/CorpusID:251563966

[19] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *USENIX Security Symposium.* https://api.semanticscholar.org/CorpusID:221178890

[20] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: Unprivileged containers for user-defined software stacks in hpc. In *Proceedings of the international conference for high performance computing, networking, storage and analysis.* 1–10.

[21] Alexandros Stamatakis. 2006. RAxML-VI-HPC: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics* 22 21, 2688–90. https://api.semanticscholar.org/CorpusID:13864758

[22] Mariano Vázquez, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmín Aguado-Sierra, Rutherford Aris, Daniel Mira, Hadrien Calmet, Fernando M. Cucchietti, Herbert Owen, Ahmed Taha, and José María Cela. 2014. Alya: Towards Exascale for Engineering Simulation Codes. *arXiv: Computational Physics.* https://api.semanticscholar.org/CorpusID:63058758

[23] Lidong Wang. 2017. Heterogeneous Data and Big Data Analytics. https://api.semanticscholar.org/CorpusID:64725916

[24] Xiaohui Wei, Shi-Yu Tong, Zhongao Sun, Xiang Li, and Hengshan Yue. 2025. ResCheckpointer: Building Program Error Resilience-Aware Checkpointing Mechanism for HPC Systems. *Journal of Computer Science and Technology.* https://api.semanticscholar.org/CorpusID:278847517

[25] xianyi and Lab of Parallel Software and Computational Science, ISCAS. [n. d.]. BLAS-Tester: a tester for BLAS libraries including OpenBLAS and Intel MKL. https://github.com/xianyi/BLAS-Tester.

[26] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective Are They? Exploring Large Language Model Based Fuzz Driver Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024).* Association for Computing Machinery, New York, NY, USA, 1223–1235. doi:10.1145/3650212.3680355

[27] Guofeng Zhang, Ziqi Shuai, Kelin Ma, Kunlin Liu, Zhenbang Chen, and Ji Wang. 2024. FDSE: Enhance Symbolic Execution by Fuzzing-based Pre-Analysis (Competition Contribution). In *Fundamental Approaches to Software Engineering.* https://api.semanticscholar.org/CorpusID:269137892