A Comparative Study of Programming Languages for a Real-Time Smart Grid Application

Michael P. Rooney Jr.
Department of EE & CS
U.S. Military Academy
West Point, NY, USA
michael.rooney@westpoint.edu

Nakul Rao
Department of EE & CS
U.S. Military Academy
West Point, NY, USA
nakul.rao@westpoint.edu

Nicholas Liebers
Department of EE & CS
U.S. Military Academy
West Point, NY, USA
nicholas.e.liebers.mil@army.mil

Aaron St. Leger
Department of EE & CS
U.S. Military Academy
West Point, NY, USA
aaron.stleger@westpoint.edu

Suzanne J. Matthews
Department of EE & CS
U.S. Military Academy
West Point, NY, USA
suzanne.matthews@westpoint.edu

***Abstract*—With security an increasing concern, SCADA system designers should consider the programming language used to implement critical smart grid applications. In this paper, we compare the performance of an anomaly detection workflow implemented in a common programming language used in SCADA systems (C) to equivalent implementations in three less commonly-known languages (Numba Python, Cython, and Rust). We benchmark our implementations on two real-world datasets of synchrophasor data and compare their performance on two Arm-based single board computers. Our results demonstrate that the Numba Python implementations achieve real-time performance in many contexts that pure Python counterparts cannot. In all tested scenarios, the Rust implementations achieve real-time performance while consuming similar amounts of power to their C counterparts. Our results suggest that SCADA designers should take a closer look at Numba Python and Rust for performant WAMS applications.**

***Index Terms*—Wide Area Monitoring System, Synchrophasor, Linear State Estimation, Anomaly Detection, Rust, C, Numba, Raspberry Pi, Single Board Computer**

I. Introduction

Security is becoming an increasing concern in smart grid design. Cyber-attacks have already had real world impacts on power grids. For example, three provinces in Ukraine were remotely attacked resulting in a six-hour loss of power for 225,000 people [1]. Russian military groups have targeted the Ukraininan power grid via cyber-attacks on multiple occasions with varying levels of success [2].

As Supervisory Control and Data Acquisition (SCADA) designers consider how best to protect the grid, we posit that they carefully consider the *language* in which they implement their applications. For real-time applications, such as Wide Area Monitoring Systems (WAMS), it is not uncommon for designers to use performant compiled languages such as C and C++. However, these languages are notoriously difficult to program safely, making it easy for even experienced developers to inadvertently introduce vulnerabilities to their applications [3].

Another consideration when picking a language is its support of auxiliary libraries that facilitate the larger application workflow. For example, a SCADA application may want to query a SQL database, perform machine learning, or visualize data. Such tasks are exponentially more difficult and require significantly more code in older languages like C, compared to a more modern language like Python. However, Python is slower than its traditional compiled counterparts, and is typically inappropriate for real-time applications.

In this paper, we explore two research questions. First, *what language(s) should SCADA designers consider when implementing real-time applications for the smart grid*? Second, *what advantage (if any) do smaller single board computers such as the Raspberry Pi Zero provide in a real-time context?*

We compare the performance of a Linear State Estimation (LSE) real-time anomaly detection workflow that is implemented in four different languages: C, Rust, Numba Python, and Cython. We run our algorithms on real data derived from a $1000 : 1$ emulated smart grid testbed, and synthesized data, and compare their performance on two Arm-based single board computers to emulate how they would run in a real-world scenario.

Our results show that the Numba Python and Cython implementations execute the LSE anomaly detection workflow in real time on the Raspberry Pi 4B, a feat that is not possible with pure Python. In almost all cases, Numba is also faster than Cython. The Numba implementations are not capable of real-time detection on the more resource-constrained Raspberry Pi 2 Zero W; in contrast, the Rust implementations achieve real-time performance on all datasets and platforms tested. These results strongly suggest that SCADA developers should take a closer look at Rust and Numba when designing their own smart grid applications, especially for WAMS.

Supported by the Department of Defense

II. Background & Related Work

A. Languages under study

Due to the inherent speed of their resulting executables, C and C++ are extremely popular languages employed in smart grid applications (e.g., [4]–[6]). However, C is a difficult language to code safely; the lack of array bounds-checking and the need to manually manage memory make C applications a popular target for exploitation. It is also more difficult in C to integrate external libraries and maintain portability without greatly expanding the size of the underlying code base.

Modern languages like Python provide greater flexibility for SCADA developers. Unlike C, Python includes native protection against most types of memory errors. Python's readability and conciseness make it a popular language for rapid development, especially for the smart grid (see [7]–[9] for examples). The diversity of the Python Package Index (PyPI) enables SCADA developers to quickly "glue" together applications with functionality like querying databases, performing machine learning, and visualizing data. However, as an interpreted language, Python is inherently slower than traditional compiled counterparts, making it unsuitable for many real-time applications.

One solution is to use a language like Cython, a hybrid Python/C language that essentially compiles Python code down to C [10]. Cython differs from Python in a few key ways. First, to improve performance, programmers typically must manually type variables in the code. Additionally, when working with NumPy arrays (which are crucial for any Python application dealing with multidimensional arrays), there are extra steps required to ensure efficient array access. Most projects utilizing Cython are therefore written in a mixture of Python and Cython, with the latter used to speed up targeted sections of code [10], [11].

The Numba [11] just-in-time compiler for Python has recently emerged as a promising alternative to Cython. Available as a Python library, Numba allows users to target specific parts of a Python program for Numba compilation by using the `@jit` decorator, enabling a developer to code in nearly pure Python. Unlike Cython, Numba does not require manual typing to be added to Python code, and the library works seamlessly with NumPy. A key disadvantage of Numba is that it allows optimization only on a subset of the Python language.

We also evaluate Rust, a relatively new memory-safe programming language. Rust's single-ownership model enables it to ensure memory safety at compile time. Proponents of Rust describe it as a secure language with Python-like syntax and fast, C-like execution [12]–[14]. Recent studies [15], [16] suggest that Rust achieves similar performance to C on many classes of problems. While adoption of Rust is still relatively low (owing to the newness of the language) [13], some researchers [17], [18] have started implementing smart grid applications in Rust. A key novelty of our work is that we are the first to compare the performance of Rust to other languages for a smart grid application, and to implement an anomaly detection algorithm in Rust.

B. Case Study: Anomaly Detection with LSE

State Estimation (SE) is a key part of monitoring electrical power grids that estimates the present state (voltage amplitudes and phase angles) [19], [20]. SE algorithms estimate the "true" value of the power system states based on measurements acquired over an interval of time, and a mathematical model of the system [21]. Traditional SCADA measurements update at a slow pace of 3 to 5 seconds and are not time-synchronized [22], resulting in the use of iterative solvers. The most common method is to estimate states by minimizing the error between measurements and the power system model through a weighted least squares (WLS) algorithm. The difficulties associated with traditional SCADA systems coupled with the computational complexity of iterative solvers makes obtaining meaningful real-time results challenging [23].

Phasor Measurement Units (PMUs) are a significant advancement compared to traditional SCADA measurements. PMUs report data at up to 120 Hz, time-synchronizes data to a high-precision GPS clock enabling direct phase measurement, and timestamps the measurements so they can be time aligned for SCADA applications. High-fidelity synchronized measurements (synchrophasors) enable a linear solution (LSE) to the state estimation problem, reducing the number of calculations and ensuring a more accurate estimate [24], [25].

For our case study, we augmented an existing anomaly detection approach [6] for synchrophasor data with LSE. Prior work [6] has shown the feasibility of performing real-time anomaly detection on synchrophasor data using the Raspberry Pi 3 single board computer, and demonstrated that a single Raspberry Pi could analyze data from up to 50 PMUs in real-time [6]. A key observation of prior work is that the Raspberry Pi was barely taxed for the simple anomaly detection approach. By adding LSE prior to the anomaly detection step, we significantly increase the computational complexity of the detection task, as LSE approaches require linear algebra on complex voltages and currents. [26]. Additionally, the fusion of LSE with anomaly detection on PMU data provides insight into the nature of the anomaly. For example, if the detected anomaly is consistent with the LSE result, then it is indicative of a physical anomalous condition in the power grid.

A recent paper [27] explored the feasibility of LSE on the Raspberry Pi. However, the authors only created a prototype of LSE using standard Python, and did not incorporate it into a larger anomaly detection workflow. Our work optimizes and extends the work in [27], incorporating LSE into an anomaly detection workflow, and expands the set of languages and Raspberry Pi platforms used for benchmarking.

C. Platforms Under Study

The Raspberry Pi is increasingly being explored [6], [27]–[30] for smart grid applications, either as a proposed component of the grid or as a testbed for other Arm-based hardware. The two Raspberry Pi platforms under study are the Raspberry Pi 4B and the Raspberry Pi 2 Zero W. The Raspberry Pi 4B is the latest model of Raspberry Pi, with a 64-bit 2.4 GHz Arm A72 processor and options of 1 GB, 2 GB, 4 GB and

8 GB of RAM. The 4B draws 2.4 watts of power at idle and 6.4 watts under load. The version we tested had 2 GB of RAM, and retails for $45.00. The Raspberry Pi Zero 2W is a more power-efficient offering with a modest 1 GHz 64-bit Arm Cortex-A53 CPU and only 512 MB of RAM. It draws about 0.7 watts of power at idle and approximately 3 watts under load. The Raspberry Pi 2 Zero W retails for $15.00.

III. Methods

A. Overview of the LSE Anomaly Detection Workflow

The objective of the previously developed [6] anomaly detection algorithm is to detect anomalous conditions within raw PMU data. The approach looks at constraint anomalies in raw PMU data (anomalies are flagged when data exceeds expected range of values) and temporal anomalies by analyzing a time window of raw PMU data (anomalies are flagged when a predefined coefficient of variation is exceeded within a window). The LSE estimates power system states by minimizing the error between PMU measurements and the model of the power system. The closed form LSE solution using a WLS approach is (see [27] for details):

$$x = (H^T W H)^{-1} H^T W z \tag{1}$$

where x is a column matrix of estimated power system states (voltage phasors), z is a column matrix of time aligned PMU measurements (voltage and current phasors), W is a covariance weighting matrix based on measurement accuracy, and H is a matrix relating the states to the measurements based on the power system model.

The fusion of anomaly detection and LSE for WAMS in a distributed environment has several advantages over a centralized approach. WAMS must process data and report results quickly, ideally in real-time. However, communicating large amounts of PMU data has significant communication network requirements [27]. Fig. 1 illustrates how single board computers distributed throughout the power grid perform anomaly detection and state estimation. Information flows from the PMUs to a Phasor Data Concentrator (PDC) on its LAN, which extracts PMU data from network packets, time aligns the data, and locally archives the data or serves it to remote applications. In the proposed scenario, the PDC serves data to a Raspberry Pi that performs anomaly detection and LSE, only communicating meaningful results to the control center when needed. This distributed architecture shows promise [6] in obtaining real-time performance and minimizing communication requirements in WAMS.

B. Datasets Description

Benchmarking is performed on two sets of data, one from a 7-bus power system and another from a 57-bus power system. Both sets are representative of real-world power systems and provide benchmarks of varying scale. The 7-bus system data was obtained from a $1000:1$ scaled emulated Smart Grid Test-Bed developed at the U.S. Military Academy [31]. The power system emulated in the testbed is based on a 7-bus, 46kV, three-phase subtransmission system containing nine

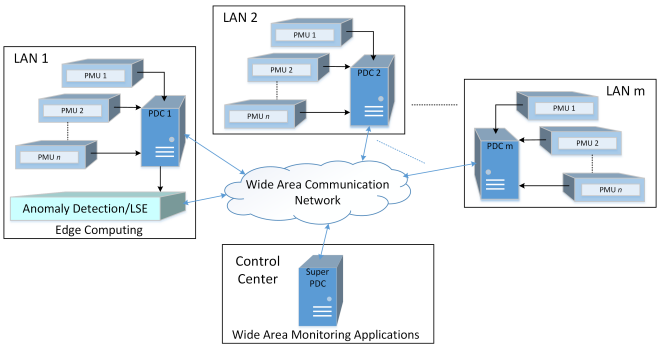

Fig. 1: Distributed architecture of state estimation and anomaly detection with PMU data

transmission lines and eight IEEE C37.118-compliant PMUs time synchronized to GPS satellite clocks. The PMUs sample data at 60 Hz, have 82 measurements (10 voltage magnitudes, 10 voltage phases, 11 current magnitudes, 11 current phases, 8 frequencies, 8 rate of change of frequencies, and 24 additional pieces of data corresponding to analog, digital, and PMU status), an anomaly detection window size of 1 second, and 14 states estimated (magnitude and phase of each bus voltage).

Data for the 57-bus power system was synthesized in software. The system model is based on the IEEE standard 57-bus power system which is a simplified approximation of the American Electric Power system in the Midwest United States from the 1960s [32]. The system contains 80 transmission lines. A true state of the 57-bus system was obtained from performing power flow on the system. PMU data was subsequently synthesized by modeling random measurement error on top of the power flow result. The 57-bus system synthesized data at 60 Hz, has 388 measurements (57 voltage magnitudes, 57 voltage phases, 80 current magnitudes, 80 current phases, 57 frequencies, 57 rate of change of frequencies), an anomaly detection window size of 1 second, and 114 states estimated (magnitude and phase of each bus voltage).

Each dataset had two sets of generated data. The first was a "clean" set of data. For the 7-bus system this was obtained by operating the the testbed under normal operating conditions. For the 57-bus system, this data was obtained by adding random measurement error within specification of measurements to true values. The second set of data was an "anomalous" set of data. For the 7-bus system this was obtained by operating the tested and perturbing the system to deviate voltages and currents from normal operating conditions. For the 57-bus system the anomalous dataset was obtained by adding random measurement error as before; however, a subset of the measurement error exceeded the specification of measurements to create anomalous conditions.

The 7-bus anomalous dataset consisted of 1.4 million measurements collected over a five minute period, corresponding to $37,475$ anomalous events, including $28,515$ constraint anomalies and $1,473$ temporal anomalies. The 57-bus anomalous dataset consists of 1.5 million measurements simulating

TABLE I: Average Detection Time Per Window (ms)

Data	C		Rust		Numba		Cython	
	RPi 4B	**RPi 2ZW**	**RPi 4B**	**RPi 2ZW**	**RPi 4B**	**RPi 2ZW**	**RPi 4B**	**RPi 2ZW**
7-bus (clean)	0.139	0.465	0.191	0.712	0.528	1.818	0.589	1.908
7-bus (anomalous)	0.152	0.509	0.225	1.152	0.557	2.513	0.643	2.383
57-bus (clean)	2.20	6.849	3.104	15.42	5.724	15.71	6.7016	16.03
57-bus (anomalous)	3.05	9.64	3.308	16.28	6.702	41.86	7.454	18.32

a collection over an approximately one minute period, corresponding to $63,340$ anomalous events, including 45 constraint anomalies and $63,295$ temporal anomalies. We verified the correctness of each implementation by ensuring that their outputs matched and that they detected all the anomalous events introduced into the testbed.

C. Experimental Methodology

The original LSE anomaly detection workflow was prototyped in pure Python; equivalent implementations were created in Numba Python, Cython, C, and Rust. Each implementation's LSE algorithm requires a unique formulation for solving (1) that corresponds to the associated power system architecture. Notably, the matrices required for the 57-bus LSE workflow are an order of magnitude larger than those required for the 7-bus.

To simulate a real-time stream, each dataset file is loaded into memory. On each window of data, LSE is first performed, followed by anomaly detection. For consistency, all three versions are compiled using LLVM, the default compiler for Rust and Numba Python. The C implementations were compiled with the Clang frontend for LLVM. For optimal performance, the Rust and C executables were compiled with the `-O3` flag, and Rust was compiled using its `-release` mode. Numba Python was compiled using `nopython` mode, which does not access Python's C API and is recommended for yielding the most performance [33].

For each simulation, platform, and respective datasets, we measure the execution time, power consumption, and the CPU and memory utilization. Valgrind's `massif` tool is used to measure the peak memory used during each simulation's execution. We use a Kill-A-Watt to measure the at-wall peak power consumption (in watts). We report the average execution time required to complete the LSE anomaly detection workflow in a single window.

IV. Results and Discussion

A. Execution Time Per Window

Table I depicts the average time it takes each implementation to perform the LSE anomaly detection workflow on a single window of data. Each window contains 82 measurements for the 7-bus implementations, and 388 measurements in the 57-bus implementations. Since the reporting rate of the system is 60 Hz, the applications must complete the LSE anomaly detection workflow in under 16.67 ms on each window of data to meet the definition of real-time.

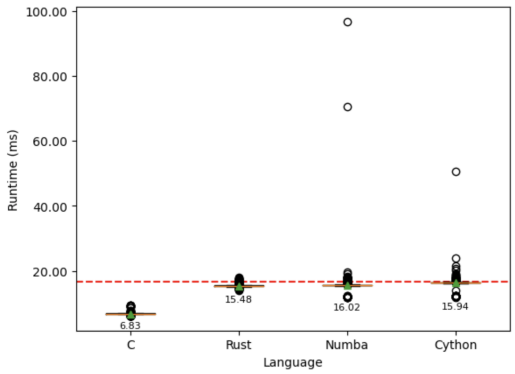

Fig. 2: Distribution of execution on Raspberry Pi 2 Zero W on 57-bus anomalous dataset without printing

On the more powerful Raspberry Pi 4B, all the implementations easily met the standard of real-time, regardless of the language used, or whether the dataset contained anomalies. Unsurprisingly, the C implementation was the fastest, averaging less than 0.152 ms on the 7-bus datasets and less than 3.05 ms on the 57-bus datasets. Rust, while slower, is not appreciably different. We note that while the Numba and Cython implementations were up to 4 times slower than the C versions, both these implementations were able to execute well under the definition of real-time on the 57-bus anomalous dataset, at 6.702 and 7.454 ms respectively. A pure Python implementation is not capable of real-time performance, taking an average of 64.532 ms to analyze a window for anomalies on the 57-bus anomalous dataset on the Raspberry Pi 4B. The use of Numba speeds up the Python code by an order of magnitude, enabling real-time performance on this device.

All implementations take longer to run on the more resource-constrained Raspberry Pi 2 Zero W. On the datasets with no anomalies, all implementations meet the definition of real-time. However, C and Rust are the only implementations that meet the definition of real-time on the anomalous 57-bus dataset, analyzing a window for anomalies in 9.64 ms and 16.28 ms, respectively. In contrast, the Numba and Cython implementations fall outside the definition of real-time, taking 41.86 ms and 18.32 ms respectively.

Table I includes the (oftentimes significant) overhead of printing detected anomalies to screen. To determine how I/O affects average execution time, we reran our 57-bus implementations on the anomalous dataset with printing turned off. Fig. 2 depicts the distribution of the detection times for these

runs on the 57-bus anomalous dataset on the Raspberry Pi 2 Zero W. While all implementations now meet the definition of real-time, we note that the Numba and Cython Python implementations have long distribution tails, with some windows taking nearly 100 ms for detection in Numba. We confirmed that these outliers contain anomalies; Cython and Numba simply take longer on such windows. In contrast, the C and Rust implementations had very tight distributions, suggesting more consistent peformance.

B. Power Consumption

Fig. 3 shows the peak power consumption measured at wall for each implementation. The C and Rust implementations consumed similar amounts of power, with the 57-bus dataset consuming slightly more power on the Raspberry Pi 2 Zero W than the 4B on the 7-bus datasets. The Python implementations have a greater discrepancy in power consumption. The 57-bus Python implementations consume 40%-48% more power than their 7-bus counterparts, suggesting that these implementations are stressing the platforms more significantly than the Rust and C implementations. This observation is in line with prior work [15], [16], [34] that conclude that interpreted languages consume more energy than traditional compiled languages.

C. Memory and CPU Utilization

The Python implementations also consume more memory than the equivalent C and Rust implementations. On the 7-bus dataset, the C and Rust implementations consume less than 1 MiB of memory, while the Python implementations consume 28 to 37 MiB of memory. For the larger 57-bus implementations, the C and Rust versions respectively consume 3.4 MiB and 5.2 MiB of memory, while the Numba and Cython versions take between 30.10 and 41.21 MiB of memory. Numba and Cython compile only sections of the Python code; the rest is still interpreted at run-time, leading to necessarily larger memory consumption.

During all benchmarks, the Raspberry Pi 2 Zero W ran consistently at 100% CPU utilization. In contrast, the Raspberry Pi 4B ran at around 25% CPU utilization, suggesting that the latter has room for additional tasks.

Our results suggest several things. First, while the Numba and Cython implementations are slower than the C versions, they are capable of completing the anomaly detection workflow in real-time on the Raspberry Pi 4B while executing an order of magnitude faster than their pure Python counterparts. Given the high interoperability the Python language provides, an argument can be made that that SCADA designers should take a closer look at Numba to greatly enhance the speed of their Python-based applications. In nearly all cases, the Numba Python implementations are faster than their Cython counterparts. Our results demonstrate that with the use of the Numba, it is possible to create real-time anomaly detection workflows using Python.

We note that Numba cannot speed up all aspects of a Python program. On more resource-constrained systems like the Raspberry Pi 2 Zero, Numba takes a long time to execute, especially on windows containing anomalous data. In contrast, Rust is capable of real-time performance on all platforms and datasets tested, while providing inherent memory-safety and fast execution. While a younger language, Rust has an emerging ecosystem of auxiliary packages, including bindings for popular libraries like TensorFlow and OpenCV.

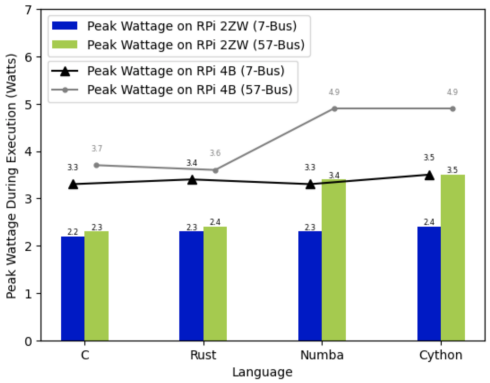

Fig. 3: Power consumption (in watts) at wall for each platform and dataset

V. Conclusion

This paper compares the performance of four languages on a WAMS smart grid application. We implemented four identical anomaly detection approaches that utilized linear state estimation, and benchmarked them on two sets of synchrophasor data. One set was obtained from a smart grid testbed and a second was synthesized from a power system and measurement model. We also compared the performance of each implementation on two models of Raspberry Pi, a popular single board computer that is increasingly used in the smart grid. We sought to answer two research questions: first, *what languages should SCADA designers consider when implementing real-time applications for the smart grid* (RQ1)? Second, *what advantage (if any) do smaller single board computers such as the Raspberry Pi Zero provide in a real-time context* (RQ2)?

With regards to RQ1, we recommend that SCADA developers take a closer look at the Numba and Rust languages. While Rust is slower than C, its ability to achieve real-time performance while guaranteeing memory safety makes it a very attractive alternative to C, especially in smart grid applications. Our results demonstrate that with Numba, Python applications are capable of real-time performance, especially when run on the Raspberry Pi 4B.

For RQ2, our results suggest that SCADA designers should consider deploying WAMS anomaly detection algorithms on the Raspberry Pi Zero 2 W, if the goal is to complete the process while consuming the least amount of power. Both the Rust and C languages consume less than 2.5 Watts of power on the Raspberry Pi 2 Zero W and are capable of detecting anomalies in real time. However, if the goal is to run the

anomaly detection process in the context of a larger workflow (involving steps such as machine learning) on the same device, the Raspberry Pi 4B offers more flexibility, consumes only moderately more power, and allows for the use of Python. We note that all the implementations, regardless of language or dataset, cause the Raspberry Pi4B run at around 25% CPU utilization. In a smart grid architecture that incorporates single board computers, our results demonstrate that the anomaly detection process can be completed for nearly "free", both in terms of power consumption and expense of hardware.

VI. Acknowledgement

Funding for this work is provided by the Department of Defense. The views expressed in this article are those of the authors and do not reflect the official policy or position of the Department of the Army, Department of Defense or the U.S. Government.

References

[1] D. E. Whitehead, K. Owens, D. Gammel, and J. Smith, "Ukraine cyber-induced power outage: Analysis and practical mitigation strategies," in *2017 70th Annual conference for protective relay engineers (CPRE)*. IEEE, 2017, pp. 1–8.

[2] (2022, APR.) Ukrainian power grid 'lucky' to withstand russian cyber-attack. BBC. [Online]. Available: https://www.bbc.com/news/technology-61085480

[3] M. Noseda, F. Frei, A. Rüst, and S. Künzli, "Rust for secure iot applications: why c is getting rusty," in *Embedded World Conference 2022, Nuremberg, 21-23 June 2022*. WEKA, 2022.

[4] R. C. Qiu, Z. Chen, N. Guo, Y. Song, P. Zhang, H. Li, and L. Lai, "Towards a real-time cognitive radio network testbed: Architecture, hardware platform, and application to smart grid," in *2010 Fifth IEEE Workshop on Networking Technologies for Software Defined Radio Networks (SDR)*, 2010, pp. 1–6.

[5] B. K. Bose, "Artificial intelligence techniques in smart grid and renewable energy systems—some example applications," *Proceedings of the IEEE*, vol. 105, no. 11, pp. 2262–2273, 2017.

[6] S. J. Matthews and A. St. Leger, "Leveraging single board computers for anomaly detection in the smart grid," in *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, Oct 2017, pp. 437–443.

[7] S. Schütte, S. Scherfke, and M. Tröschel, "Mosaik: A framework for modular simulation of active components in smart grids," in *2011 IEEE First International Workshop on Smart Grid Modeling and Simulation (SGMS)*, 2011, pp. 55–60.

[8] K. Anderson, J. Du, A. Narayan, and A. E. Gamal, "Gridspice: A distributed simulation platform for the smart grid," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2354–2363, 2014.

[9] L. Thurner, A. Scheidler, F. Schäfer, J.-H. Menke, J. Dollichon, F. Meier, S. Meinecke, and M. Braun, "Pandapower—an open-source python tool for convenient modeling, analysis, and optimization of electric power systems," *IEEE Transactions on Power Systems*, vol. 33, no. 6, pp. 6510–6521, 2018.

[10] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[11] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015.

[12] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017, pp. 156–161.

[13] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: rust as a case study," in *Symposium on Usable Privacy and Security*, 2021.

[14] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021.

[15] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?" in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 256–267.

[16] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[17] K. Hutto, S. Paul, B. Newberg, V. Boyapati, Y. Vunnam, S. Grijalva, and V. Mooney, "Puf-based two-factor authentication protocol for securing the power grid against insider threat," in *2022 IEEE Kansas Power and Energy Conference (KPEC)*, 2022, pp. 1–6.

[18] J. Keller, S. Paul, S. Grijalva, and V. J. Mooney, "Experimental setup for grid control device software updates in supply chain cyber-security," in *2022 North American Power Symposium (NAPS)*, 2022, pp. 1–6.

[19] Y.-F. Huang, S. Werner, J. Huang, N. Kashyap, and V. Gupta, "State Estimation in Electric Power Grids: Meeting New Challenges Presented by the Requirements of the Future Grid," *IEEE Signal Processing Magazine*, vol. 29, no. 5, pp. 33–43, Sep. 2012.

[20] Y. Xia, Y. Chen, Z. Ren, S. Huang, M. Wang, and M. Lin, "State estimation for large-scale power systems based on hybrid CPU-GPU platform," in *2017 IEEE Conference on Energy Internet and Energy System Integration (EI2)*, Nov. 2017, pp. 1–6.

[21] A. Jovicic, M. Jereminov, L. Pileggi, and G. Hug, "A linear formulation for power system state estimation including rtu and pmu measurements," in *2019 IEEE PES Innovative Smart Grid Technologies Europe (ISGT-Europe)*, 2019, pp. 1–5.

[22] G. A. Ortiz, D. G. Colomé, and J. J. Quispe Puma, "State estimation of power system based on SCADA and PMU measurements," in *2016 IEEE ANDESCON*, Oct. 2016, pp. 1–4.

[23] H. Karimipour and V. Dinavahi, "Accelerated parallel WLS state estimation for large-scale power systems on GPU," in *2013 North American Power Symposium (NAPS)*, Sep. 2013, pp. 1–6.

[24] K. D. Jones, J. S. Thorp, and R. M. Gardner, "Three-phase linear state estimation using Phasor Measurements," in *2013 IEEE Power & Energy Society General Meeting*, Jul. 2013, pp. 1–5.

[25] S. Soni, S. Bhil, D. Mehta, and S. Wagh, "Linear state estimation model using phasor measurement unit (PMU) technology," in *2012 9th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE)*, Sep. 2012, pp. 1–6.

[26] A. S. Dobakhshari, S. Azizi, M. Paolone, and V. Terzija, "Ultra Fast Linear State Estimation Utilizing SCADA Measurements," *IEEE Transactions on Power Systems*, vol. 34, no. 4, pp. 2622–2631, Jul. 2019.

[27] S. D. Hassak, A. St. Leger, H. Oh, and D. F. Opila, "Implementing a pmu based linear state estimator on a single board computer," in *2022 North American Power Symposium (NAPS)*, 2022, pp. 1–6.

[28] A. M. Damle and M. Kulkarni, "A low-cost embedded platform for synchronised wide area frequency measurement," in *2014 Eighteenth National Power Systems Conference (NPSC)*, 2014, pp. 1–6.

[29] P. Castello, C. Muscas, P. Attilio Pegoraro, and S. Sulis, "Low-cost implementation of an active phasor data concentrator for smart grid," in *2018 Workshop on Metrology for Industry 4.0 and IoT*, 2018, pp. 78–82.

[30] I. Sittón-Candanedo, R. S. Alonso, O. Garcìa, L. Muñoz, and S. Rodríguez-Gonzàlez, "Edge computing, iot and social computing in smart energy scenarios," *Sensors*, vol. 19, no. 15, 2019.

[31] A. St. Leger, J. Spruce, T. Banwell, and M. Collins, "Smart grid testbed for wide-area monitoring and control systems," in *2016 IEEE/PES Transmission and Distribution Conference and Exposition (T&D)*, 2016, pp. 1–5.

[32] IEEE 57-bus system. Illinois Center for a Smarter Electric Grid (ICSEG). [Online]. Available: https://icseg.iti.illinois.edu/ieee-57-bus-system/

[33] Numba Documentation, "Compiling python code with @jit," https://numba.readthedocs.io/en/stable/user/jit.html, accessed: 2023-05.

[34] S. Georgiou, M. Kechagia, and D. Spinellis, "Analyzing programming languages' energy consumption: An empirical study," in *Proceedings of the 21st Pan-Hellenic Conference on Informatics*, ser. PCI 2017. New York, NY, USA: Association for Computing Machinery, 2017.