Evaluation of Obfuscation Strategies for Python Source Codes

João Pedro Vital Brasil Wieland, Flávio Luis de Mello

***Abstract*—This work presents a comparative analysis of the impact of different obfuscation techniques and tools on Python code performance. Five main obfuscation approaches were evaluated: PyArmor, Pyminifier, Cython, PyInstaller, and PyObfuscate, analyzing their impact on metrics such as execution time, memory usage, code size, and startup time. The methodology was based on an automated benchmarking tool, developed specifically for this study, capable of applying different obfuscation techniques to Python code and objectively measuring its performance. The results demonstrated that it is possible to effectively protect real-time Python applications with low performance impact, especially using approaches such as PyArmor or Cython. The work concludes that the appropriate selection of obfuscation tools should consider the specific context of the application, evaluating trade-offs between protection, performance, and ease of distribution, especially for critical real-time systems. Therefore, using a benchmark generation tool can be valuable in deciding the best strategy to implement in each project.**

***Index Terms*—Benchmark, Code Obfuscation, Python, Information Security**

I. Introduction

SOFTWARE development market is growing at an accelerated rate, and and also an increasing concern about how to protect the intellectual property of the source code of developed systems. In particular, artificial intelligence systems that use scripts , whether for inference or training, are susceptible to having their methods and techniques visible to anyone who has access to their files. There are some alternative approaches to protecting against unauthorized access, such as: obfuscation, watermarking , tamper-proofing, diversification and minification.

Obfuscation is a set of techniques that can be applied to make it difficult to read and understand source code. Several obfuscation strategies have been developed over time as the demand for more efficient strategies in terms of runtime and logic protection. A good obfuscation strategy will necessarily have great power to ensure maximum possible obscurity; resilience to be protected from automated tools; stealth to ensure that the obfuscated code snippet can blend in with the rest of the program; and low computational cost to avoid generating a large overhead. As the complexity of the obfuscation strategy used increases, the natural tendency is for there to be an impact on the application's runtime. Discussing the best obfuscation proposals, their advantages and measuring their impacts are essential issues for the market that increasingly tries to protect the intelligence of the systems developed.

Watermarking consists of adding a unique identifier to ensure that the code's copyright is not violated. For this technique to be effective, the watermark must be placed in a way that is not easily detected. This strategy has a trade-off between resilience, computational cost, stealth, and data size. The watermark can be static, stored within the executable, or dynamic, hidden in the source code and only extracted during runtime. This practice has proven effective in ensuring intellectual property without high computational costs during execution.

Tamper-proofing is a strategy that prevents the execution of code that has had some type of manipulation of its source code. This strategy needs to be implemented in several parts of the code to ensure that it is effective. The most common strategy is to compile the code fragments separately and each fragment is only executed when it is verified that there has been no manipulation.

Diversification strategy consists of making each copy of the code have a different internal structure to make it difficult for malware to attack the program. This tactic does not guarantee that a program is free of exploitable flaws, but it does make it difficult for attacks to be carried out on multiple machines running the same program since each execution instance is unique.

Minification consists of removing indentation and line breaks from code to reduce its size and optimize its transmission, it is widely used in inference scripts that seek efficiency. Among the strategies presented, it is the simplest to break down and there are already several tools for this. This technique can be applied to reduce the memory space used by scripts, which is a great advantage in the context in which it is applied.

The adoption of Java as the main programming language for developing applications for the Android operating system has led to even more attention being paid to the area of code protection, as there are tutorials and tools for reverse engineering [1] that are easily found and available on the World Wide Web, and due to the nature of the language. New languages, such as Dart , already implement obfuscation

This work was supported in part by Machine Intelligence and Computer Models Laboratory of Federal University of Rio de Janeiro (IM^{2}C/Poli/UFRJ) under grant Poli 19.257 of Coppetec.

João Pedro Vital Brasil Wieland is undergraduate on Computation and Information Engineering student at the Electronics and Computer Department from Polytechnic School at Federal University of Rio de Janeiro, Brazil (email: jpvbwieland@poli.ufrj.br).

Flávio Luis de Mello (D.Sc.) is associate professor at the Electronics and Computer Department from Polytechnic School at Federal University of Rio de Janeiro, Brazil (email: fmello@poli.ufrj.br).

strategies automatically when compiled for mobile programs to ensure that they are less vulnerable. At the mobile development ecosystem, minification is also used as a way to optimize data storage on devices.

This paper aims to discuss specifically code obfuscation techniques, especially scripts of interpreted programming languages, exams what are the impacts of their implementation and what are the advantages of their use. To this end, related works will be presented to show what the main scholars are developing on this subject. Finally, a qualitative and quantitative analyses of representative algorithms of each approach will be carried out.

II. Related works

MELLO [1] presents several attacks on artificial intelligence algorithms during the training and inference stages. He points out that attacks performed in the inference stages can benefit from having easy-to-read access to the inference scripts.

Hosseinzadeh et al. [4] analyze the main reasons for the use of algorithms and logic protection strategies by compiling data from several published articles. The main reasons given for the use of the techniques are: ensuring that reverse engineering of the code becomes more difficult, preventing mass attacks on vulnerabilities, ensuring that the code is not changed in an unauthorized way, and hiding confidential data such as cryptographic keys.

Balakrishnan and Schulze [2] enumerate the different obfuscation strategies that have traditionally been used for intellectual property protection. Furthermore, they highlighted that several strategies were originally developed to protect and avoid detection of computer viruses, since obfuscation algorithms can modify the structure of the code to be executed on the infected machine.

Collberg and Thomborson [3] points out that obfuscation strategies can be considered "security by obscurity", which the community considers ineffective because they only hide information from the attacker. However, he acknowledges that they can be used to make it harder for the attacker to reverse engineer the code, i.e., it is an effective technique to increase the complexity of the work of breaking the code and therefore, dissuade the attacker. The author also details two strategies to make the analysis of obfuscated code even harder: the implementation of Antidisassembly strategies so that disassembled programs cannot reliably print the execution, and Antidebugging strategies that attack the debugger program by writing to its memory area.

Skolka et al. [5] analyzed the strategies used in Javascript and measured the impact on performance. Minified codes, for example, had an improvement in performance compared to the original code, while obfuscated code had an increase of up to 37% in execution time. However, the analysis pointed out that the implementation of minification and obfuscation generate correctness problems in the code because the adopted strategy manipulates functions in such a way that they stop working. It is therefore necessary to verify the impacts on code execution to ensure that the methodology used does not harm the quality of the software.

Uchida et al [6] developed a framework to effectively place watermarks in scripts without major impacts on efficiency. It was demonstrated to be effective in protecting against data compression attacks and parameter modifications of computational intelligence models. This article demonstrates the efficiency of providing a document authorship signature, but does not add code obfuscation security.

Dong et al [7] performed an analysis of the protection strategies implemented in Android applications and proposed strategies to detect the technique used. Anti-tampering solutions have become essential in mobile applications due to the ease of reverse engineering compiled packages. The main methodologies in the mobile environment, identifier renaming, string encryption and Java reflection, are not widely adopted since there is no standard adopted by the industry.

Obfuscation techniques have been developed almost in parallel as a way to break the obfuscations that have been created over time. Udupa et al. [8] details reverse engineering techniques that can be used to recover code that has been protected. The paper also classifies obfuscation techniques into two types: surface obfuscation , which consists of only changing the syntax and semantics of the code to make it harder to read, and deep obfuscation, which actually manipulates the program's execution structure.

Kholia [9] demonstrates through his article how obfuscation done correctly is essential for the logical protection of systems and preventing the discovery of attack vectors. By analyzing Dropbox code and applying techniques to deobfuscate the code, it was possible to reverse engineer how the Dropbox client module code worked. This technique is interesting because it points to an efficient approach to trying to recover obfuscated code.

III. Benchmarking obfuscation strategies in python

A. Motivation and Objectives

Python is an interpreted language widely used for AI development, and therefore presents particular challenges for protection. The inherent transparency of its bytecode and its dynamic nature make reverse engineering relatively easy, increasing the demand for code protection tools.

While the code protection techniques discussed above have the potential to protect intellectual property, their implementation may be limited by their impact on code execution. This issue is particularly relevant for systems where performance requirements are critical. To objectively quantify this impact and provide practical guidelines for developers, this paper performed a systematic study to compare different obfuscation tools for Python.

B. Obfuscation Tools for Python

Several tools are available for obfuscating Python code, each implementing different techniques and offering varying levels of protection. For this study, five tools were selected, covering different approaches and complexities:

1) PyArmor [10] : Commercial solution that implements

advanced protection through bytecode encryption and runtime verification mechanisms. It uses a C runtime to decode and execute protected code, adding anti-debugging and anti-tampering mechanisms.

2) Pyminifier [11] : Open source tool focused on lightweight lexical obfuscation techniques, including identifier renaming, comment removal, and code compression. Focuses on simplicity and minimal performance impact.

3) Cython [12] : Although designed for performance optimization, compilation to native C code offers substantial protection against reverse engineering as a side effect, transforming Python code into platform-specific native libraries.

4) PyInstaller [13] : Packaging tool that generates executable, incorporating the Python interpreter and all dependencies. Provides basic protection as a side effect of the packaging process.

5) PyObfuscate [14] : More archaic implementation of obfuscation for Python, focusing on basic syntactic transformations like variable renaming and comment removal.

These tools represent a spectrum of obfuscation strategies, from superficial modifications to deep bytecode transformations, allowing for a comprehensive analysis of the trade-offs between protection and performance.

C. Benchmark Methodology

To ensure objective and reproducible assessments, an automated analysis tool was developed with a modular architecture composed of four main components:

- Tool Manager : Configures and runs different obfuscation solutions, automatically detecting availability and compatibility in the test environment.
- Test Orchestrator : Coordinates the complete workflow, including dependency detection, application of obfuscation techniques, and execution in an isolated environment.
- Metrics Collectors : Specialized components for accurate measurement of execution time, memory consumption, code size, and startup time.
- Results Analyzer : Processes collected data, calculates statistics and generates comparative visualizations.

The tool is designed for fully automated operation, minimizing experimental variability and ensuring consistency in measurements. A critical aspect of the implementation is the automatic detection of dependencies between Python files, allowing for proper obfuscation of complex systems with multiple interconnected components. The tool is described by Wieland [15] and was developed in a modular way for possible implementation of new analysis methodologies and analysis of any Python code.

To evaluate the impact of obfuscation techniques in different usage contexts, two sets of tests were defined:

1. Fundamental Operations Tests :
   - compute_intensive.py : Simulates computationally intensive workloads (matrix multiplication)
   - string_manipulation.py : Focuses on string operations (concatenation, substitution, encoding)
   - io_operations.py : Evaluates input/output operations with JSON serialization and file processing
   - mixed_operations.py : Combines the previous patterns into a more complex scenario, implementing a DataProcessor class that performs statistical processing, string manipulation, and I/O operations.
2. Real-world Application Study Case : Computer vision-based driver drowsiness detection system, representing a practical AI application with real-time requirements. This system uses OpenCV and dlib for image processing and facial detection, implementing proprietary algorithms for facial feature analysis.

This diverse test suite allows evaluating the behavior of obfuscation tools both in isolated workloads and in complex systems with external dependencies and real-time requirements.

Four key metrics were collected to quantify the impact of obfuscation:

- Execution time (seconds) : Direct measure of impact on overall performance
- Memory Usage (MB) : Peak consumption during full execution
- Code Size (KB) : Relevant for distribution and deployment
- Startup time (ms) : Particularly important for interactive applications

The experimental protocol followed a rigorous procedure with 1,000 iterations for each tool and test case combination, ensuring statistical significance of the results. All measurements were performed in a Linux environment with Python 3.11.5, on dedicated hardware to minimize external interference. The experiment took place on a MacBook Pro with ARM64 architecture and M1 Pro processor, with macOS 24.3. The processor contains 8 physical cores and 8 logical cores at 3,2 GHz, 16 GB RAM. The obfuscation tools versions are: Pyminifier 2.3.3, Cython 3.0.12, PyInstaller 6.12.0, PyArmor 9.1.2 and PyObfuscate 0.0.2.

IV. Results and Discussion

A. Fundamental Operations Tests

The results for fundamental operations revealed distinct patterns for each tool, as described at Table 1 .

TABLE I
OBFUSCATION TOOL METRICS OVER DIFFERENT COMPUTING PROFILE

Execution Time (s)				
Configuration	**Profile**			
	Compute Intense	**String Manipulation**	**I/O**	**Mixed**
Original	0.077	0.047	0.106	0.348
PyArmor	0.100	0.050	0.107	0.373
Pyminifier	0.077	0.047	0.106	0.352
PyObfuscate	0.077	0.047	0.107	0.353
Cython	0.048	0.047	0.102	0.348
PyInstaller	2.000	2.259	2.052	2.309

Memory Usage (MB)				
Configuration	Profile			
	Compute Intense	String Manipulation	I/O	Mixed
Original	10.66	11.85	17.82	16.78
PyArmor	14.92	15.94	26.33	21.96
Pyminifier	10.67	11.87	17.55	16.66
PyObfuscate	11.27	12.28	18.10	17.63
Cython	10.62	11.93	17.01	18.50
PyInstaller	1.35	1.38	1.36	1.35
Code Size (KB)				
Configuration	Profile			
	Compute Intense	String Manipulation	I/O	Mixed
Original	1.47	1.46	3.00	4.63
PyArmor	15.33	12.62	22.27	35.94
Pyminifier	1.16	1.02	2.18	3.29
PyObfuscate	41.27	25.87	36.43	69.26
Cython	0.21	0.21	0.20	0.21
PyInstaller	6,712.84	6712,09	6751.17	6752.69
Time to Startup (s)				
Configuration	Profile			
	Compute Intense	String Manipulation	I/O	Mixed
Original	1.88	1.94	1.93	1.92
PyArmor	1.93	1.92	1.93	1.94
Pyminifier	1.87	1.95	1.93	1.94
PyObfuscate	1.85	1.95	1.93	1.91
Cython	1.94	1.94	1.91	1.93
PyInstaller	2.11	2.50	2.07	2.08

PyArmor showed a moderate runtime penalty (+18.4%), with the impact being most pronounced for computationally intensive operations (+30.1%) and least for string manipulation (+6.2%). The significant increase in code size (+940.3%) reflects the addition of protection and runtime code.

Pyminifier showed the lowest overall performance impact, with virtually negligible penalty on runtime (+1.2%) and memory (+1.8%), while significantly reducing code size (-52.7%). This tool proved to be ideal for performance-critical cases, offering basic protection without compromising efficiency.

Cython was the only tool to improve performance, with an average reduction of 24.7% in execution time, particularly for computationally intensive operations (-38.2%). This result confirms the dual benefit of the approach: protection via compilation to native code and performance gain. The drastic reduction in code size (-86.2%) is another significant advantage.

PyInstaller demonstrated the largest negative impact across all metrics, with dramatic increases in execution time (+564.3%), memory usage (+25.6%), and especially code size (+46,352.1%). These results indicate that, while valuable for simplified deployment, it should be used with caution when performance is a priority.

B. Drowsiness Detection System

The results for the drowsiness detection system revealed particularly interesting patterns, with notable differences from the baseline tests (see Table 2 and Table 3).

TABLE II
OVERALL METRICS FOR EACH OBFUSCATION TOOL FOR AN AI APPLICATION

Tool	Execution Time (s)	Memory Usage (MB)	Code Size (KB)	Startup Time (ms)
Original	23.56 ±0.23	281.80 ±3.42	10.00 ±0.00	1.78 ±0.47
PyArmor	23.69 ±0.10	281.58 ±2.97	24.93 ±0.00	1.76 ±0.43
Pyminifier	23.67 ±0.06	282.01 ±3.15	4.73 ±0.00	1.76 ±0.34
Cython	23.67 ±0.13	281.42 ±3.28	0.20 ±0.00	1.89 ±0.52
PyInstaller	48.44 ±2.35	3.25 ±0.58*	136,673.77 ±0.00	4.51 ±1.02

*±: Standard deviation between different runs. *: Anomalous value due to limitations in the measurement methodology.*

PyArmor demonstrated a surprisingly small impact on runtime (+0.6%), significantly lower than that observed in the baseline tests (+18.4%). This discrepancy suggests that for systems that rely heavily on native libraries like OpenCV and dlib, the overhead of obfuscation is proportionally lower, as most of the processing occurs in unobfuscated native code.

Pyminifier and Cython showed similar behavior to PyArmor, with a negligible impact on execution time (+0.5%). For Cython, it is particularly interesting to note the lack of performance gain observed in computationally intensive operations, suggesting that the bottleneck is in the native libraries and not in the Python code itself.

PyInstaller continued to show the largest negative impact (+105.6% in runtime), although significantly smaller than in the baseline tests. The massive increase in code size (over 13,000 times larger) reflects the inclusion of the full Python interpreter and all dependencies, including OpenCV and dlib.

PyObfuscate (not included in the table) completely failed to process the complex system, compromising its functionality. This incompatibility highlights the risks of using unmaintained tools in modern applications with multiple dependencies.

TABLE III
THE DIRECT COMPARISON BETWEEN THE FUNDAMENTAL TESTS AND THE REAL SYSTEM IS PARTICULARLY REVEALING: VALUES REPRESENT PERCENTAGE INCREASE IN EXECUTION TIME COMPARED TO THE ORIGINAL CODE

Tool	Computational Operations (%)	String Manipulation (%)	Detection System (%)
PyArmor	+30.1	+6.2	+0.6
Pyminifier	+0.9	+1.3	+0.5
Cython	-38.2	-0.3	+0.5
PyInstaller	+2,498.7	+457.2	+105.6

C. Practical Implications

The results present significant practical implications for developers and companies. For systems that rely heavily on native libraries like OpenCV and dlib, the obfuscation overhead is generally lower than expected, especially for

PyArmor and Cython. This suggests that intellectual property protection in computer vision and AI applications can be implemented with minimal performance impact.

Choosing between different tools should consider not only performance impact, but also factors such as ease of distribution, compatibility with external libraries, and financial cost. For example, while PyArmor offers excellent protection with controlled impact, its cost (between $990 and $9,990 for commercial licenses) can be prohibitive for smaller projects.

For critical applications where code legitimacy is essential, as suggested by Hosseinzadeh et al. [4], techniques such as those implemented by PyArmor are particularly valuable because they incorporate anti-tampering mechanisms that can detect unauthorized modifications.

PyInstaller's significant impact on startup time (+1,254.6% in the fundamental tests) may be intolerable for interactive applications, although it may be acceptable for systems that operate in batch mode without real-time requirements.

V. Conclusions

THIS work presented a systematic and empirical analysis of the impact of different obfuscation strategies on the performance of Python codes, with a specific focus on computer vision applications. The results indicate that it is possible to effectively protect real-time Python applications with controlled performance impact, especially using tools such as PyArmor and Cython.

The significant disparity between the impact observed in the fundamental tests and in the complex drowsiness detection system highlights the importance of evaluating obfuscation techniques in the specific context of the target application. This observation confirms the concerns raised by Skolka et al. [5] about the need to verify the impacts on code execution for each application context.

The developed benchmark tool represents a significant practical contribution, allowing developers to objectively evaluate different obfuscation strategies on their own code before implementing a definitive solution. This personalized evaluation capability is particularly valuable considering the diversity of available approaches and the lack of widely adopted industry standards, as pointed out by Dong et al. [7] .

The results suggest that the appropriate selection of obfuscation tools should consider the specific application context, evaluating trade-offs between protection, performance, ease of distribution, and cost. For critical real-time systems, such as the drowsiness detector studied, PyArmor and Cython emerged as the most efficient solutions, with the choice between them depending mainly on economic considerations and the value of the intellectual property to be protected.

As future work, we suggest expanding the analysis to include direct evaluation of the effectiveness of protection against reverse engineering attempts, complementing performance metrics. Additionally, the study of hybrid approaches, combining different tools for specific system components, represents a promising direction to optimize the trade-off between protection and performance.

References

[1] MELLO, Flávio Luis de. A Survey on Machine Learning Adversarial Attacks. Enigma : Journal of Information Security and cryptography, Brasília, v. 7, n. , p. 1-7, Dec. 2020. doi: doi.org/10.17648/jisc.v7i1.76

[2] BALAKRISHNAN, Arini; SCHULZE, Chloe. Code Obfuscation Literature Survey. 2005. Available at: https://pages.cs.wisc.edu/~arinib/writeup.pdf. Accessed on: December 19, 2005.

[3] COLLBERG, Christian S.; THOMBORSON, Clark. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection: tools for software protection. IEEE Transactions on Software Engineering , [s. l], v. 28, no. 8, p. 735-746, Aug. 2002. doi: 10.1109/TSE.2002.1027797

[4] HOSSEINZADEH, Shohreh et al . Diversification and obfuscation techniques for software security: A systematic literature review. Elsevier: Information and software Technology. Turku, Finland, p. 72-93. jul. 2017. doi: 10.1016/j.infsof.2018.07.007

[5] SKOLKA, Philippe. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In: WWW '19: THE WORLD WIDE WEB CONFERENCE, 1., 2019, San Francisco. Proceedings of International World Wide Web Conference 2019. New York: Acm, 2019. p. 1735-1746. doi: 10.1145/3308558.3313752.

[6] UCHIDA, Yusuke et al . Embedding Watermarks into Deep Neural Networks. In: International Conference on Multimedia Retrieval, 1., 2017, Bucharest. New York: ACM, 2017. p. 269-277. doi: 10.1145/3078971.3078974.

[7] DONG, S. et al. (2018). Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In: Beyah, R., Chang, B., Li, Y., Zhu, S. (eds) Security and Privacy in Communication Networks. SecureComm 2018. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 254. Springer, Cham. doi: 10.1007/978-3-030-01701-9_10 . Accessed on: October 15, 2022.

[8] SK UDUPA, SK Debray and M. Madou, "Deobfuscation: reverse engineering obfuscated code," 12th Working Conference on Reverse Engineering (WCRE'05) , 2005, pp. 10 pp.-54, doi: 10.1109/WCRE.2005.13.

[9] KHOLIA, Dhiru. Looking Inside the (Drop) Box. In: Workshop on Offensive Technologies, 7., 2013, Washington, Dc. Proceedings of 7th USENIX Workshop on Offensive Technologies. New York: Usenix, 2013. p. 1-7. Available at: https://www.usenix.org/conference/woot13/workshop-program/presentation/kholia. Accessed on: October 16, 2022.

[10] ZHAO, Jondy. PyArmor's Documentation . 2020. Available at: https://pyarmor.readthedocs.io/en/latest/. Accessed on: October 20, 2022.

[11] MCDOUGALL, Dan. Pyminifier - Minify, obfuscate, and compress Python code . 2014. Available at: https://liftoff.github.io/pyminifier/. Accessed on: November 12, 2022.

[12] Behnel, Stefan; Bradshaw, Robert ; Seljebotn, Dag Sverre; Ewing, Greg; Stein, William; Gellner, Gabriel. Welcome to Cython’s Documentation. 2025. Available at: https://cython.readthedocs.io/en/latest/. Accessed on: May 21, 2025.

[13] Cortesi, David; Bajo, Giovanni; Caban, William. PyInstaller Manual. 2025. Available at: https://pyinstaller.org/en/stable/ . Accessed on: May 21, 2025.

[14] _____. PyObfuscate. 2023. Available at: https://pyobfuscate.com/ . Accessed on, May 21, 2025

[15] Wieland, João Pedro Vital Brasil. Benchmarks of Obfuscation Strategies from Python Codes. Undergraduate on Computer and Information Engineering, Politechnique School, Federal University of Rio de Janeiro, 2024.

João Pedro Wieland received the B.Eng. degree in Computer and Information Engineering from the Federal University of Rio de Janeiro (UFRJ), Rio de Janeiro, Brazil, in 2025. He is currently pursuing the M.Sc. degree in Systems and Computer Engineering at the Graduate Program in Systems and Computer Engineering (PESC), COPPE/UFRJ, Rio de Janeiro, Brazil.

He has experience in software development and applied research in machine learning, with a focus on large language models (LLMs) for assistive technologies and software automation. He has participated in academic-industry collaboration projects involving IoT and AI in agriculture and public education systems. His current research interests include program analysis, automated accessibility, and generative AI in engineering applications.

João Pedro Wieland is a recipient of the Young Scientist Award (Prêmio Jovem Cientista) granted by the Brazilian National Council for Scientific and Technological Development (CNPq) in 2012.

Flávio Luis de Mello received his DSc. in Theory of Computation and Image Processing from the Federal University of Rio de Janeiro - UFRJ (2006), MSc. in Computer Graphics from the Federal University of Rio de Janeiro - UFRJ (2003), Undergraduate degree in Systems Engineering from the Military Institute of Engineering - IME (1998).

He developed command and control systems and implemented military messages interchange applications during twelve years as a Brazilian Army officer. He was responsible for developing software applications based on machine learning and knowledge reasoning from Mentor Group.

Dr Mello currently is Full Professor at the Electronic and Computer Engineering Department (DEL) of Polytechnic School (Poli) at the Federal University of Rio de Janeiro (UFRJ). He is head of the Machine Intelligence and Computing Models Laboratory (IM^2C).