A comparison of programming languages' applicability for encrypted data analysis*

Leila Rzayeva[1,*,†], Abilkair Imanberdi[1,†] and Yerassyl Yermekov[2,†]

[1] *Research and Innovation Center "CyberTech", Astana IT University, Business center EXPO, 55/11 Mangilik El ave., block C1, Astana, Kazakhstan*

[2] *Information security systems, L.N. Gumilyov Eurasian National University, 2 Satbaev str., ENU Educational building, Astana, Kazakhstan*

Abstract

A sufficient level of data processing performance is important for real digital forensics use cases, where data should be processed as fast as possible to streamline the forensic analysis process and reduce crucial delays. While constantly improving the hardware is a simple and intuitive method to achieve performance increases, the software optimization provides significantly higher performance gains. This paper presents a comparative analysis of mainstream programming languages applicability for the statistic analysis of encrypted data. A set of statistical data test programs were developed in Python and Go programming languages in order to compare their performance and utilization metrics—execution time, CPU utilization and RAM usage. The comparison was performed on two separate computers—the first one employing a server-oriented AMD EPYC central processing unit, and the second one employing a mobile AMD Ryzen CPU. Overall, four independent tests were performed, the results of which were averaged from two samples. Results were consistent with expectations—a mostly compiled programming language with better optimization techniques, such as Go, provides significantly better time metrics than the Python version of the same data encryption detection algorithm—the Go version is 2.5 to 5 times faster than the Python version.

Keywords

encryption analysis, statistical methods, autocorrelation, performance metrics, digital forensics

1. Introduction

Nowadays, more and more crimes are carried out partially or entirely in cyberspace. Digital forensics processes have become an important part of investigations. Modern mobile devices are equipped with large amounts of memory, often more than 128 gigabytes, and in the case of forensic analysis of such data dumps, the time factor becomes important, especially when the discovery process has to be performed in a limited amount of time, often days and hours instead of months.

The reliable detection of data encryption is an important part of the digital forensics process as well, and it determines the direction of the following actions and tests. For example, an analysis of a smartphone's internal memory is being performed. If the internal memory is not encrypted and a valid file system structure is found, the analysis can go down to the file system level. Otherwise, if the data block is determined as encrypted with a valid file system, a conclusion is made that the file system uses FBE (File-Based Encryption); if data encryption is detected, but without a valid file system, that means a high probability of a full-disk encryption being used. Data retrieval strategies significantly differ for all of the described cases. A variety of statistical and non-statistical methods are being used for data encryption detection. Statistical analysis methods often utilize an empirical byte distribution model and a theoretical uniform distribution model to determine whether these models are similar. Non-statistical methods can employ different data analysis algorithms, which do not use empirical distribution models or any statistical models. Such methods can use pattern analysis, rule-based searches, etc. Data encryption detection methods' performance is highly

*CSDP'2025: Cyber Security and Data Protection, July 31, 2025, Lviv, Ukraine

* Corresponding author.

† These authors contributed equally.

l.rzayeva@astanait.edu.kz (L. Rzayeva); a.imanberdiyev@astanait.edu.kz (A. Imanberdi); 990726350870@enu.kz (Y. Yermekov);

0000-0002-3382-4685 (L. Rzayeva); 0009-0005-6144-2392 (A. Imanberdi); 0009-0001-3957-4787 (Y. Yermekov)

dependent on a number of factors, originating from both hardware and software limitations, but the most important factor is software optimization.

2. Literature review

The problem of programming languages' differences in execution optimization is actively researched for various use cases.

For example, a 2020 research study, carried out by Paweł Dymora and Andrzej Paszkiewicz from Rzeszów University of Technology (Rzeszów, Poland) [1], includes Python, Java, and Go programming languages in the context of supporting Industry 4.0 decision-making processes, a process especially critical in minimizing time delays. The results demonstrate a non-linear increase in execution time and resource usage for each programming language, with Python being the fastest language for the large decision tree algorithm. On the other hand, the Java programming language performed the worst in execution time and resource utilization. The Go version has displayed intermediate results.

A 2023 research by Atishay Jain from Dr. Akhilesh Das Gupta Institute of Technology and Management (New Delhi, India) [2] focuses on the applicability of Java and Python programming languages for machine learning purposes. The results display that the Java programming language is significantly faster for basic operations and slightly faster for complex programs, such as the Tic-Tac-Toe player algorithm.

A 2022 study by Luka and Luca Olivari from the Šibenik University of Applied Science (Šibenik, Croatia) [3] is mainly focused on the performance of the Ant Colony Optimization (ACO) algorithm for the Traveling Salesman Problem. The algorithm was implemented in Python, C, C#, R, and MATLAB languages and tested on a high-performance laptop. The Python version has exhibited the worst result, being on average 12 times slower than the pure C implementation. MATLAB, C#, and R versions of the same algorithm displayed a non-linear relationship between data size and performance relative to the base C version, with the MATLAB version displaying the highest non-linearity.

A common conclusion of these studies is that the interpreted implementations of programming languages (such as R, MATLAB, and default CPython for Python) are noticeably slower and more resource-intensive than more minimalistic and compiled programming languages and their implementations (C#, Go, C, etc).

In addition to the general performance of languages, the implementation of cryptographic algorithms is an important factor. For example, studies show that methods for analyzing encrypted data, in particular using cryptographic structures such as hybrid crypto-code structures based on false codes, can differ significantly in efficiency [4]. The development and implementation of post-quantum cryptographic algorithms is a separate research area that requires attention to practical challenges and solutions [5, 6]. Ensuring security in software is also an important aspect, as confirmed by the analysis of vulnerabilities in mobile frameworks [7] and the search for hard-coded credentials [8]. The generation of reliable pseudo-random sequences, which are the basis for many cryptographic systems [8–10], also requires careful design to increase their cryptographic robustness [11–13]. All these studies emphasize that the choice of tools and their implementation have a direct impact on the ultimate efficiency and security of systems.

3. Data analysis speed variability factors

As mentioned before, the performance of statistical and non-statistical data encryption detection methods depends on a number of factors, such as CPU performance (measured in operations per second), data storage sequential and random access speed (measured in megabytes per second) and latency (measured in milliseconds), target operating system threads/processes management strategies and the level of the program optimization, that is, if the data analysis algorithm is implemented in an efficient way using efficient and highly optimized tools [14–16].

Modern programming languages have different levels of source code and binary optimization methods. Also, the level and quality of the optimizations depend on the used compiler or interpreter (in case of the existence of multiple compilers or interpreters, such as with C/C++ and Python). For example, a standard CPython Python interpreter [17] generally performs worse than optimized, but unmaintained PyPy interpreter [18], which employs JIT compilation.

3.1. Compiled and interpreted programming languages

Programming languages can be differentiated into two main categories: compiled and interpreted. It is possible to create a compiler for an interpreted language and vice versa, but most programming languages usually utilize one of the two program execution methods.

Compiled programming languages perform the compilation process to transform the source code into a binary file (so-called ahead-of-time compilation [19]). The resulting binary file is usually platform-dependent, with the main portability limitations being the operating system and the target computer's architecture. Compiled programming languages typically offer high performance levels, direct hardware access, and advanced parallelism support.

Interpreted programming languages are programming languages that use an interpreter for code execution [20]. The interpreter is a program that directly executes the written code without precompiling it, usually by converting the source code into an intermediate representation and running it line by line. Because of that, interpreted languages are highly portable and versatile, as the code can be run directly on multiple platforms with a small number of adaptation changes or no changes. However, interpreted programming languages usually have worse performance metrics than compiled languages, which is caused by the intermediary layer between the hardware and the program itself. Also, some interpreters (for example, CPython) severely limit the program's ability to use multi-threading and multi-processing, because they can only run in one thread.

3.1.1. JIT compilation for interpreted languages

JIT (just-in-time) compilation is a form of dynamic compilation, which performs the compilation of the code at run time, unlike classic ahead-of-time compilation [21]. This allows for immediate performance optimizations. A common JIT tactic is as follows:

- Initial interpretation—the program is being compiled into an unoptimized bytecode and executed.
- Performance monitoring—while the unoptimized program is running, it is being analyzed for commonly reused code sections (also known as "hot spots") and execution patterns.
- Optimization—detected "hot spots" and patterns are being dynamically compiled with optimizations applied. An optimized variant can include data type changes, idiom simplification, code reordering and exclusion, conditional statements optimization, etc.
- Execution—unoptimized bytecode is replaced with an optimized variant.
- Re-evaluation and decision making—if the optimized version performs worse or emits errors, it is replaced by another optimization variation or by the initial version.

In general, JIT compilation combines both positive and negative aspects of the ahead-of-time compilation and straight interpretation. It is commonly used in Java, JavaScript, PHP, WebAssembly, Ruby, C#, Python (as an experimental branch in the 3.14 release), and other languages.

4. Data Test Suite composition and components

Data Test Suite (DTS) is a program that implements a previously developed complex method for detecting data encryption (Figure 1). DTS utilizes both statistical and non-statistical methods to determine whether the provided partition image file (Mode 1) or a set of individual files (Mode 2) is

encrypted. Two versions of this program were developed: version 1 was written in Python 3.12 and uses various external libraries for data manipulation and analysis, such as NumPy [22], SciPy [23], Seaborn [24], Matplotlib [25], and others; version 2 is a complete rewrite of the previous version in Go 1.24.1, which uses a minimal number of external libraries. Most notably, this version employs the github.com/burntsushi/rure-go [26] binding package for a fast regular expression engine found in the Rust programming language's standard library [27], the github.com/montanaflynn/stats package for statistical functions [28], and the github.com/vimeo/go-magic interface package for the libmagic library, used for MIME type extraction [29].

4.1. Autocorrelation test

The autocorrelation function is a correlation of a signal (or, in our case, a data block) with a delayed (shifted) copy of itself [30]. This method is commonly used for detecting hidden and/or weak periodicities in a data set. This method (without prior data normalization) can be used to detect structured and repeatable sections in data blocks. The standard deviation value for the set of average autocorrelation coefficient values (per block) is calculated. Encrypted and highly compressed data blocks exhibit low autocorrelation coefficient results and low standard deviation of the set of points (large input data is split into smaller blocks), while plaintext blocks exhibit varying values of the coefficient and high standard deviation values. In the DTS, a block size of 1 kibibyte is being used with a maximum lag value of 50 bytes.

4.2. Image file system detection test

This test uses the GNU Parted disk management utility [31] to determine if the input file contains a valid partition or partition table. GNU Parted is an advanced open-source command-line partition management software that supports both block devices and image files. The output of the utility is parsed in order to determine the file system type. This test is used to branch the analysis strategies using the following rule set:

- IF a file system is detected and the autocorrelation test result is "not encrypted"—the image is not encrypted and can be mounted for further analysis.
- ELSE IF a file system is detected and the autocorrelation test result is "encrypted"—the image is a product of a file-based encryption software.
- ELSE IF no file system is detected—the image is a product of a full-disk encryption or is not an image.

4.3. Compression test

As is known, encrypted data are hard to compress. This property can be used to determine if input data is encrypted or not. It is worth mentioning that this test does not differentiate between encrypted and already compressed data, as they exhibit similar statistical parameters. A set of compression utilities is used to calculate the average compression ratio of the input data. If the average compression ratio is below 1,1, the data are considered encrypted. Compression ratio can be less than 1,0, which means that the compression algorithm has added a layer of redundancy (block structures, recovery data, etc).

4.4. Kolmogorov-Smirnov test

The Kolmogorov-Smirnov goodness-of-fit test is a statistical test used to determine how well a set of observations corresponds to a statistical model [32]. Encrypted data exhibit near-uniform byte distribution patterns, while non-encrypted data usually have some discrepancies in their patterns. The Kolmogorov-Smirnov test result is a maximum value of the distance between empirical and theoretical models, although the P-value is the most commonly used criterion.

4.5. MIME type test

MIME string is a unified data type descriptor, originally designed to describe email attachment types [33]. A MIME string consists of two parts: type and subtype, separated by a slash symbol. Main MIME type entries are “application”, “audio”, “image”, “message”, “multipart”, “text”, “video”, “font”, “example”, “model”, and “haptics”. MIME strings can be used to determine if a file is previously compressed and can be excluded from a batch scan, because compressed files, as was mentioned before, compressed and encrypted files have similar statistical properties. MIME detection is implemented using the vimeo/go-magic binding to the highly optimized libmagic library and has high levels of performance. This makes the performance testing redundant, and this test is excluded from the performance testing. MIME type detection test branches the batch mode file evaluation process as follows: IF the file MIME string corresponds to compressed data (archived or not), THEN the test for this file is skipped, ELSE the file is being analyzed as usual.

4.6. Signature detection test

File signatures are sequences of bytes, which are embedded in the file structure to denote its type and properties. The file signature analysis is used to find such bytes in a file system image. If the data file is encrypted (or contains random data), the number of signatures per megabyte is usually low (below 150 signatures per MB, usually around 60-70 signatures per MB). Images with unencrypted data and without empty blocks usually demonstrate significantly higher signature values per MB because they contain individual files. Signature search test in Go version is implemented using the rure-go binding package, which is claimed to be faster than the native Go implementation of the regular expressions matching engine. The Python version utilizes the standard library regex functionality.

5. Performance testing

5.1. Testing methodology

The performance testing will be carried out on two separate computers with processors for different use cases. Both Python and Go versions will be tested in Mode 1 (Mode 2 uses statistical tests, identical to Test 1, and non-statistical tests, such as MIME type or file system detection, are negligibly fast). Speed of individual tests, RAM usage, and CPU utilization will be measured using built-in time count functions and top software.. Two passes of every benchmark were performed, with the results being averaged. A 1-gibibyte random data file was used to compare the performance of both tests on both systems.

5.2. Test system specifications

Test system 1:

- HP 255 G9 laptop.
- AMD Ryzen 5 5625U CPU (6 cores, 12 threads, Zen3 architecture, 2.3 GHz base clock, 4.3 GHz boost clock, 14830 Passmark points in multi-core, 2877 Passmark points in single-core [34]).
- OpenSUSE Tumbleweed operating system.
- 16 GB DDR4-3200 RAM in dual-channel mode.
- Kingston XS1000 external SSD for dataset storage.

Test system 2:

- Remote virtual server on Microsoft Hyper-V hypervisor.

- 8 virtual cores of AMD EPYC 7513 CPU (32 cores, 64 threads, 2.6 GHz base clock, 3.65 GHz boost clock, 59330 Passmark points in multi-core, 2479 Passmark points in single-core [35]).
- Ubuntu 24.04 LTS operating system.
- 8 GB of virtual RAM.
- 200 GB of virtual disk space (allocated on a solid-state drive).

5.3. Testing results

Performance test results are shown in Table 1.

Table 1
Performance test results

Test name	System 1, Python (s)	System 1, Go (s)	System 1, difference (%)	System 2, Python (s)	System 2, Go (s)	System 2, difference (%)	System 2 to 1, difference, Python (%)	System 2 to 1, difference, Go (%)
Autocorr	318.81	278.04	114.66	503.24	722.24	69.68	157.85	259.76
Counter	41.87	22.09	189.55	78.07	53.09	147.07	186.47	240.32
K-S	2.31E-04	1.98E-05	1163.46	2.36E-04	2.96E-05	797.37	102.38	149.38
Compression	63.92	62.61	102.09	145.33	143.32	101.40	227.37	228.91
Signatures	3160.30	353.58	893.81	5454.62	913.91	596.84	172.60	258.48
Entropy	4.76E-03	2.17E-05	21895.65	9.95E-03	5.34E-05	18634.35	209.01	245.59
Total	3584.90	716.32	500.46	6181.27	1832.56	337.30	172.43	255.83

CPU utilization for autocorrelation, entropy and other statistical tests average is at 100% with fluctuations between 90% and 140% for single core (average 12.5% per whole CPU with 8 cores), as they utilize only one thread and one core for calculations (Figures 1 and 2).

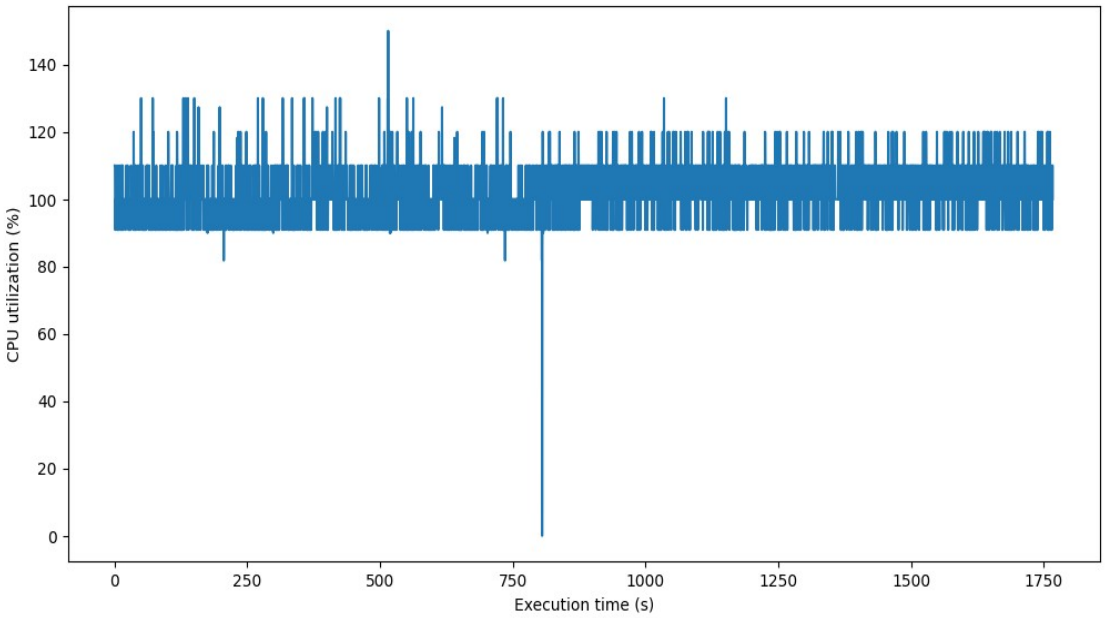

Figure 1: CPU utilization plot of the Go version of the test suite

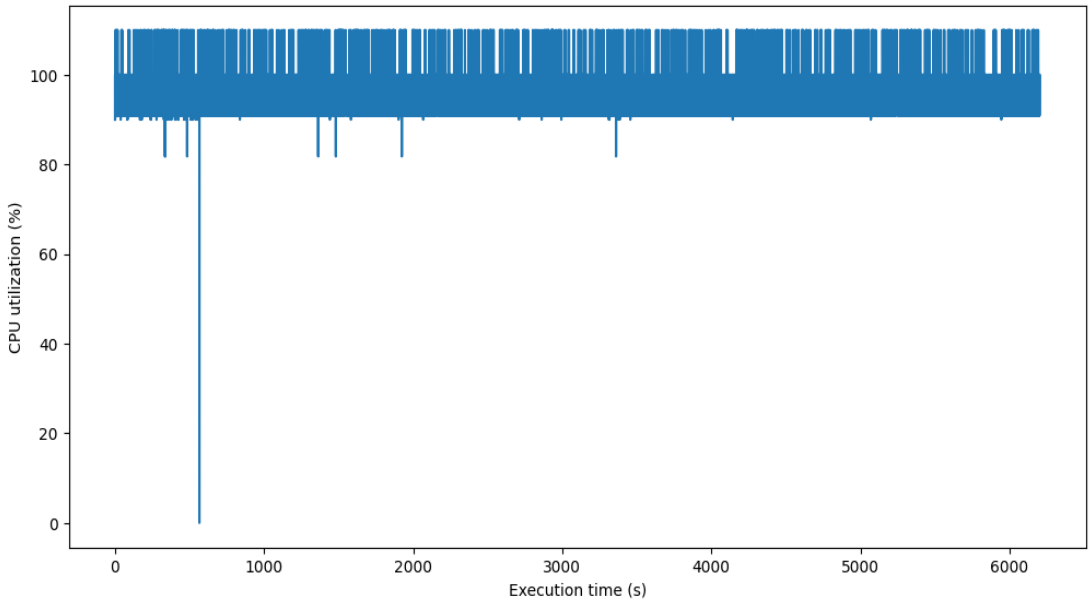

Figure 2: CPU utilization plot of the Python version of the test suite

RAM usage for the Python version of the test suite is at 140-181 MiB for autocorrelation and counter calculation phases and 143 MiB for the signature search test (Figure 3).

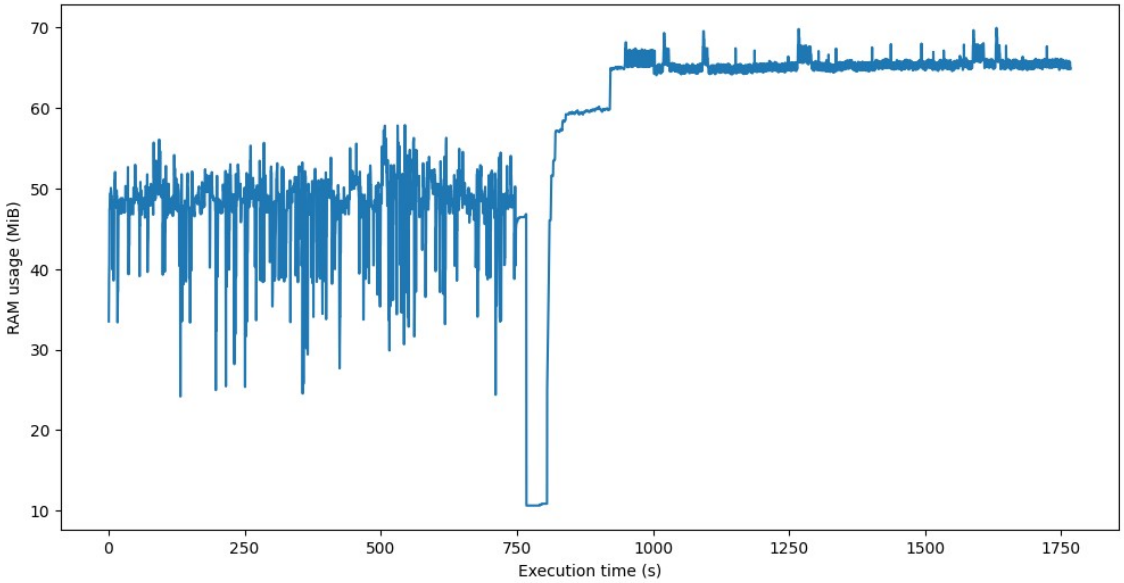

Figure 3: RAM usage plot of the Go version of the test suite

RAM usage for Go version of the test suite with 1 MiB block size is 25-56 MiB (fluctuating) for autocorrelation and counter calculation phases and 68-69 MiB for signature search test (Figure 4).

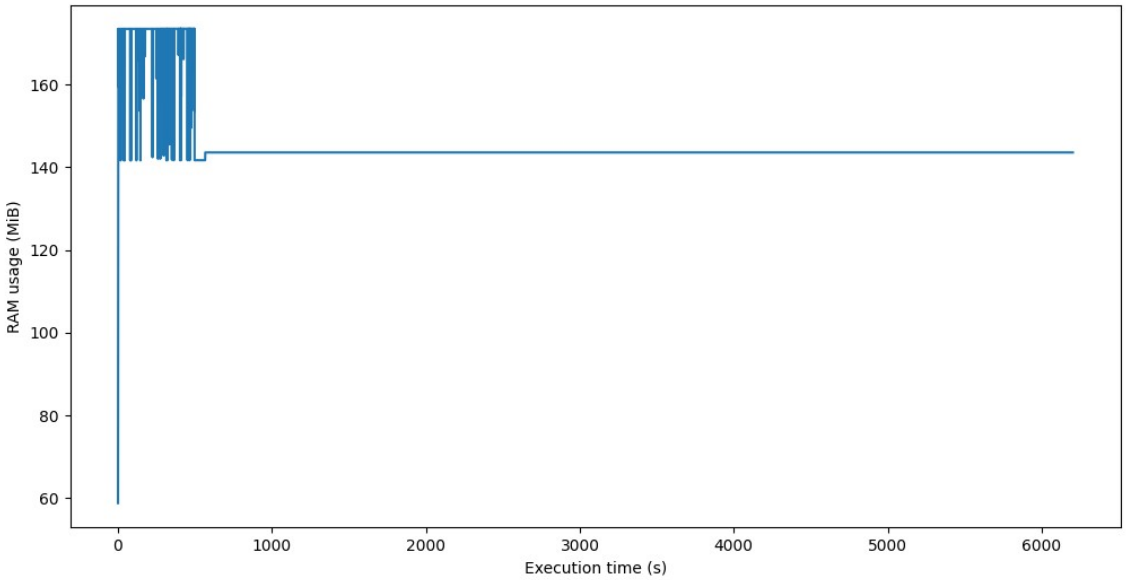

Figure 4: RAM usage plot of the Python version of the test suite

6. Results analysis

As is clearly visible, performance test results are highly dependent on the programming language's internal optimization mechanisms, such as runtime code optimization, garbage collection, etc. The Go version of the test suite performs 4.2 times better on average than the Python version.

System 1, counterintuitively, demonstrates better results than System 2, despite having a laptop CPU instead of the more powerful server type. This can be explained by the fact that server CPUs are optimized to handle a large number of parallel tasks with lower core performance expectations. Consumer CPUs are targeted for strong single-core workloads, such as games. This claim is further confirmed by the lower single-core score for the System 2 (Section 4.2).

A sudden drop of CPU utilization to 0 is due to the compression test execution: this test does not run in the main thread, but is performed using exec() calls, and computing processes are separate for each compression algorithm.

RAM usage plots also show better optimization techniques (garbage collection, dereferencing), applied during the build process of the Go version, as its RAM usage is 2 times lower on average. The Python interpreter does not employ advanced memory usage optimization techniques, which explains almost constant RAM usage for most of the run.

Conclusions

This comparative analysis clearly demonstrates the fact of prevalence of the Go programming language compared to the Python programming language in data encryption analysis tasks. This claim is supported by the performed double testing on computers with different usage scenarios.

The Go version of the Data Test Suite is on average four times faster than its Python counterpart and has better resource handling patterns, induced by the highly-optimized Go compiler with advanced compiling techniques and code optimizations. The performance increase and resource optimizations of the Go version are deemed significant.

Also, a distinctive difference between the performance of consumer and server CPUs was identified and explained. The main cause of a such performance difference is explained by different work load scenarios for these processors.

Acknowledgements

This study was carried out with the financial support of the Committee of Science of the Ministry of Science and Higher Education of the Republic of Kazakhstan under Contract №388/PTF24-26 dated 01.10.2024 under the scientific project IRN BR24993232 “Development of innovative technologies for conducting digital forensic investigations using intelligent software-hardware complexes”.

Declaration on Generative AI

While preparing this work, the authors used the AI programs Grammarly Pro to correct text grammar and Strike Plagiarism to search for possible plagiarism. After using this tool, the authors reviewed and edited the content as needed and took full responsibility for the publication’s content.

References

[1] P. Dymora, A. Paszkiewicz, Performance Analysis of Selected Programming Languages in the Context of Supporting Decision-Making Processes for Industry 4.0, Appl. Sci. 10.23 (2020) 8521. doi:10.3390/app10238521

[2] A. Jain, Comparative Analysis of Java and Python in Machine Learning: Investigate and Compare the Suitability and Performance of Java and Python for Machine Learning Tasks, Int. J. Res. Publ. Rev. 4.12 (2023) 1151–1167. doi:10.55248/gengpi.4.1223.123305

[3] L. Olivari, L. Olivari, Influence of Programming Language on the Execution Time of Ant Colony Optimization Algorithm, Teh. Glas. 16.2 (2022) 231–239. doi:10.31803/tg-20220407095736

[4] Serhii Yevseiev, et al. Development of Niederreiter Hybrid Crypto-Code Structure on flawed Codes, Eastern-European J. Enterp. Technol. 1.9(97) (2019) 27–38. doi:10.15587/1729-4061.2019.156620

[5] P. Vorobets, et al., Implementing Post-Quantum KEMs: Practical Challenges and Solutions, in: Cybersecurity Providing in Information and Telecommunication Systems II, 3826, 2024, 212–219.

[6] A. Horpenyuk, et al., Analysis of Problems and Prospects of Implementation of Post-Quantum Cryptographic Algorithms, in: Classic, Quantum, and Post-Quantum Cryptography (CQPC-2023), 3504, 2023, 39–49.

[7] T. Fedynyshyn, et al., Security Implications of Mobile Development Frameworks: Findings from Static Analysis of Android Apps, in: 17th Int. Conf. Trends in Radioelectronics, Telecommunications and Computer Engineering, 2024, 444–448.

[8] O. Mykhaylova, et al., Hardcoded Credentials in Android Apps: Service Exposure and Category-based Vulnerability Analysis, in: Cybersecurity providing in information and telecommunication systems II, 3826, 2024, 206–211.

[9] V. Maksymovych, O. Harasymchuk, I. Opirskyy, The Designing and Research of Generators of Poisson Pulse Sequences on Base of Fibonacci Modified Additive Generator, in: Advances in Intelligent Systems and Computing, Springer International Publishing, Cham, 2018, 43–53. doi:10.1007/978-3-319-91008-6_5

[10] V. Maksymovych, et al. P. Development of Additive Fibonacci Generators with Improved Characteristics for Cybersecurity Needs. Appl. Sci. 12 (2022) 1519. doi:10.3390/app12031519

[11] I. Opirskyy, et al., Pseudorandom Sequence Generator based on the Computation of ln 2, in: CQPC-2024: Classic, Quantum, and Post-Quantum Cryptography, 3829, 2024, 79–86.

[12] O. Harasymchuk, et al. Generator of pseudorandom bit sequence with increased cryptographic security, Metall. Min. Ind.: Sci. Tech. J. 5 (2014) 25–29.

[13] V. Maksymovych, et al., A New Approach to the Development of Additive Fibonacci Generators based on Prime Numbers, Electronics, 10(23) (2021) 2912-1–2912-10. doi:10.3390/electronics10232912
[14] R. Chernenko, et al., Encryption Method for Systems with Limited Computing Resources, in: Cybersecurity Providing in Information and Telecommunication Systems, vol. 3288 (2022) 142-148.
[15] M. Iavich, et al., Classical and Post-Quantum Encryption for GDPR, in: Classic, Quantum, and Post-Quantum Cryptography, vol. 3829 (2024) 70–78.
[16] A. Ilyenko, et al., Practical Aspects of using Fully Homomorphic Encryption Systems to Protect Cloud Computing, in: Cybersecurity Providing in Information and Telecommunication Systems II, vol. 3550 (2023) 226-233.
[17] Guido van Rossum, CPython, Software, v. 3.13.3, 2025. https://github.com/python/cpython
[18] PyPy Community, PyPy, Software, v. 7.3.19, 2025. https://github.com/pypy/pypy
[19] Contributors to Wikimedia Projects, Ahead-of-Time Compilation, Wikipedia, 2007. https://en.wikipedia.org/wiki/Ahead-of-time_compilation
[20] Contributors to Wikimedia Projects, Interpreter (Computing), Wikipedia, 2002. https://en.wikipedia.org/wiki/Interpreter_(computing)
[21] D. Notario, Does the JIT Take Advantage of My CPU?, 2005. https://learn.microsoft.com/en-us/archive/blogs/davidnotario/does-the-jit-take-advantage-of-my-cpu
[22] C. R. Harris, et al., Array Programming with NumPy, Nature 585.7825 (2020) 357–362. doi:10.1038/s41586-020-2649-2
[23] P. Virtanen, et al., SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, Nat. 17.3 (2020) 261–272. doi:10.1038/s41592-019-0686-2
[24] M. Waskom, Seaborn: Statistical Data Visualization, J. Open Source Softw. 6.60 (2021) 3021. doi:10.21105/joss.03021
[25] J. D. Hunter, Matplotlib: A 2D Graphics Environment, Comput. Sci. & Eng. 9.3 (2007) 90–95. doi:10.1109/mcse.2007.55
[26] A. Gallant, J. Semaan, K. Adapa, Rure-Go: Go Bindings to RUst's REgex engine, Software, v. v0.0.0-20231211185014-8a0f52724b91, 2023. https://github.com/BurntSushi/rure-go
[27] Rust Community, Regex, Software, v. 1.11.1, 2024. https://crates.io/crates/regex
[28] M. Flynn, Stats—Golang Statistics Package, Software, v. 0.7.1, 2023. https://github.com/montanaflynn/stats
[29] Vimeo, J. Ruggles, H. Donnay, Go-Magic: Go Library for Getting MIME Type using Libmagic, Software, v. 1.0.0, 2018. http://github.com/vimeo/go-magic
[30] Probability and Random Processes for Electrical and Computer Engineers, Cambridge University Press, 2006.
[31] GNU Foundation, GNU Parted, Software, v. 3.6, 2023. https://git.savannah.gnu.org/cgit/parted.git
[32] Contributors to Wikimedia projects, Kolmogorov–Smirnov test, Wikipedia, 2001. https://en.wikipedia.org/wiki/Kolmogorov–Smirnov_test
[33] Internet Assigned Numbers Authority, Media Types, 2025. https://www.iana.org/assignments/media-types/media-types.xhtml
[34] PassMark Software, AMD Ryzen 5 5625U Benchmark. https://www.cpubenchmark.net/cpu.php?cpu=AMD+Ryzen+5+5625U&id=4760
[35] PassMark Software, AMD EPYC 7513 Benchmark. https://www.cpubenchmark.net/cpu.php?cpu=AMD+EPYC+7513&id=4383