March 17 2021

Surya Keswani
Donald Stewart

Analyzing Programming Language Performance on Simon - a Lightweight Block Cipher

I. Abstract

It has been very interesting learning about programming languages, and we were interested in measuring the effects of different compilers and interpreters in a program performance setting. In this paper we measure the performance of a cryptographic algorithm (Simon) by implementing Simon in 3 different programming languages: Go, Python3, and JavaScript. We then run a series of performance tests and measure performance metrics for each of the 3 Simon implementations. The performance metrics measured include: execution time, CPU user usage, CPU system usage, memory usage, raw compression, disk compression, lines of code. This paper outlines our requirements for each implementation to maintain consistency as well as the results of our tests, as described in the Results section.

II. An Overview of Simon

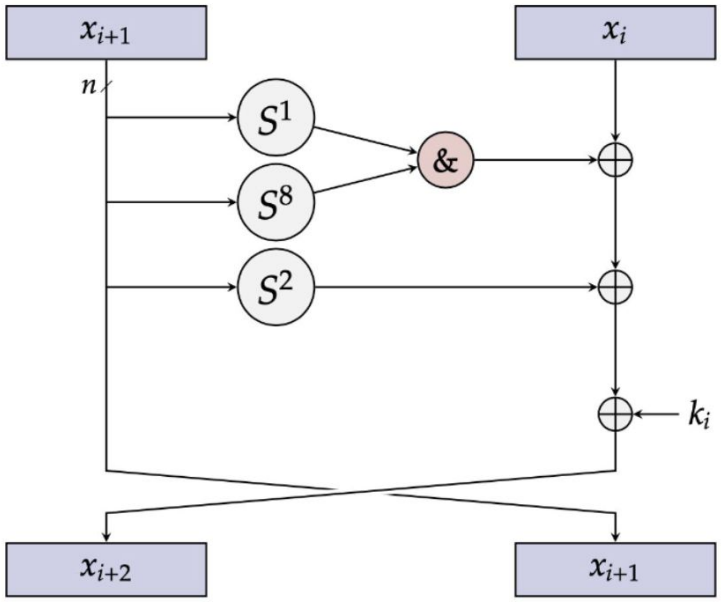

Figure 2.1

Simon is a Lightweight Block Cipher that was originally created by the NSA in June 2013. The cryptographic family of Lightweight Block Ciphers includes encryption schemes that have the properties of utilizing a symmetric key, operate on a fixed length

of bits (called blocks), while their implementation is relatively simplistic and needs little memory to operate. When the NSA began working on this cryptosystem, they saw the need for creating secure measures in the increasing collection of the *Internet of Things*. Noticing the potential of Wifi enabling an array of physical objects provokes the need for a secure design that could be implemented in circumstances where little memory and processing power is available. Ultimately, this led to the creation of Simon 128/256, which could provide similar security guarantees to that of AES 128/256 (a well-known encryption standard), and can be employed in scenarios where AES isn't possible.

$$k_{i+m} = \begin{cases} c \oplus (z_j)_i \oplus k_i \oplus \left(I \oplus S^{-1}\right)\left(S^{-3}k_{i+1}\right), & m = 2 \\ c \oplus (z_j)_i \oplus k_i \oplus \left(I \oplus S^{-1}\right)\left(S^{-3}k_{i+2}\right), & m = 3 \\ c \oplus (z_j)_i \oplus k_i \oplus \left(I \oplus S^{-1}\right)\left(S^{-3}k_{i+3} \oplus k_{i+1}\right), & m = 4 \end{cases}$$

Figure 2.2

The specification for Simon 128/256 is comparatively simple and can be summarized by Figure 2.1 and Figure 2.2. The second figure describes the calculations involved in creating the key schedule. We elected to generate the keys using the formula where *m = 4* in Figure 2.2. This key schedule ensures the greatest security possible for the Simon Block Cipher and is comparable to the best encryption schemes known to date. There will be 72 keys generated to correspond with the 72 rounds of encryption in Simon 128/256. A single round is depicted in Figure 2.1. The Simon specification we chose to implement requires 128b input which is known as the plaintext. In the first figure note that this 128b input is divided into two 64b words (X_i, X_{i+1} corresponds to the least significant and most significant 64b respectively) to perform round operations on. These words then enter the Feistel part of the network where they go under a series of bit Xors, bits shifts, and a single bit And operation to guarantee the result cannot be reversed by a system of linear equations. The round result output, X_{i+2}, and the previous round input, X_{i+1}, then become the new inputs for the following round. In short, a plaintext enters this system round 1, and 72 rounds later the cipher text we want is the output.

A major advantage of using Block Ciphers is that decryption relies on the same infrastructure as encryption. When given a ciphertext, the computation to uncover the plaintext follows the same structure as before. First, the ciphertext will be given as input to the system described in Figure 2.1. Then, the only difference between creating the ciphertext and recovering the plaintext is that now the key schedule will be reversed in the computations (round one will use the 72nd key and round 72 will use the first key).

III. Implementation

In order to keep our performance measurements as fair and unbiased as possible, we followed a few rules to maximize the similarity in each implementation. Our specification for each implementation is listed below.

1. Each implementation is run 10 times and the performance metrics include both the raw data from each of the 10 runs as well as the average metric across all 10 runs.
2. Though Simon supports multiple sizes for keys and blocks, each Simon implementation only supports 128-bit block sizes and 256-bit key sizes.
3. Every implementation encrypts and then decrypts the same test vector (the one provided in the original paper, under the Sources section). Each encrypted / decrypted result is then checked against the hardcoded test vector values
4. Every implementation creates the same functions that do the same tasks.
5. Each implementation checks the performance metrics in the same exact order in a main function to minimize ordering discrepancies. The order of performance measuring is as follows: execution time, memory usage, CPU user usage, and CPU system usage.
6. Every implementation starts with the test vector as a hexadecimal value. The hex value is then encrypted to a bit list for encryption and decryption. After encryption/decryption, the bit list is then converted back into a hexadecimal value to check against the test vector. Note: we know a bit list is an inefficient way to implement Simon but we wanted to include list structures in the program as lists are a common tool used in programming. The goal of this project is to get a better understanding of how these languages perform in a general sense.

The hexadecimal test vector used for this project is:

```
Plaintext:  0x74206e69206d6f6f6d69732061207369
Ciphertext: 0x8d2b5579afc8a3a03bf72a87efe7b868
Key:  0x1f1e1d1c1b1a19181716151413121110 0f0e0d0c0b0a0908 0706050403020100
```

All of our implementations can be found on GitHub. Instructions to run the code can be found in the `README.md` file.

IV. Results

For each implementation, we measured execution time, CPU user usage, CPU system usage, memory usage, raw compression, disk compression, lines of code. Every implementation was run on the same hardware and operating system, as can be seen in Figure 4.1 (we include this information because we expect our program to perform differently on different hardware and operating systems). The following subsections describe in detail our findings for each of these performance metrics.

Figure 4.1

A. Execution time

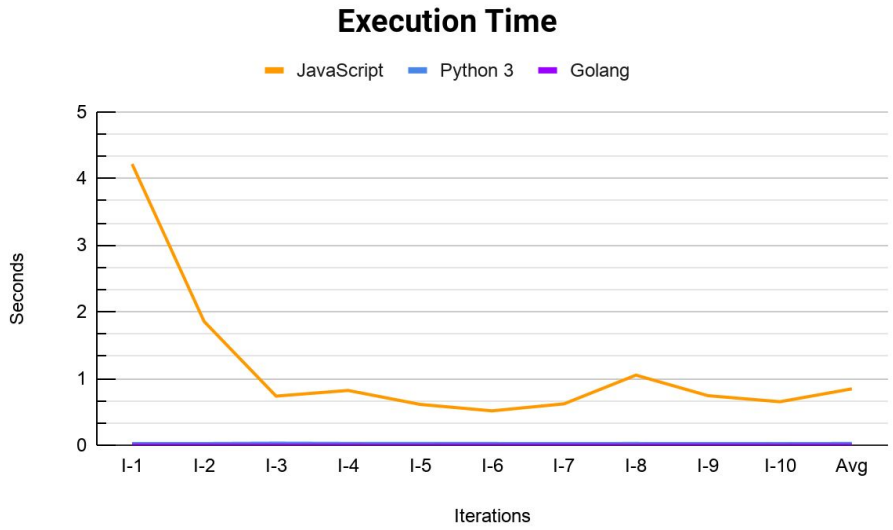

Figure 4.2

Figure 4.2 shows how long each Simon implementation took from start to finish. As the graph shows, JavaScript performs the absolute slowest and is the most inconsistent with its runtimes, with an outlying runtime of over 4 seconds for the first iteration. Python and Go are much more consistent with their runtimes. Though it may seem that Python3 and Go have similar runtimes, Go actually runs 50x faster than the Python3 implementation on average. Figure 4.3

is the execution times transformed using log base 2. The logarithmic transformation better shows how much faster Go is than Python3.

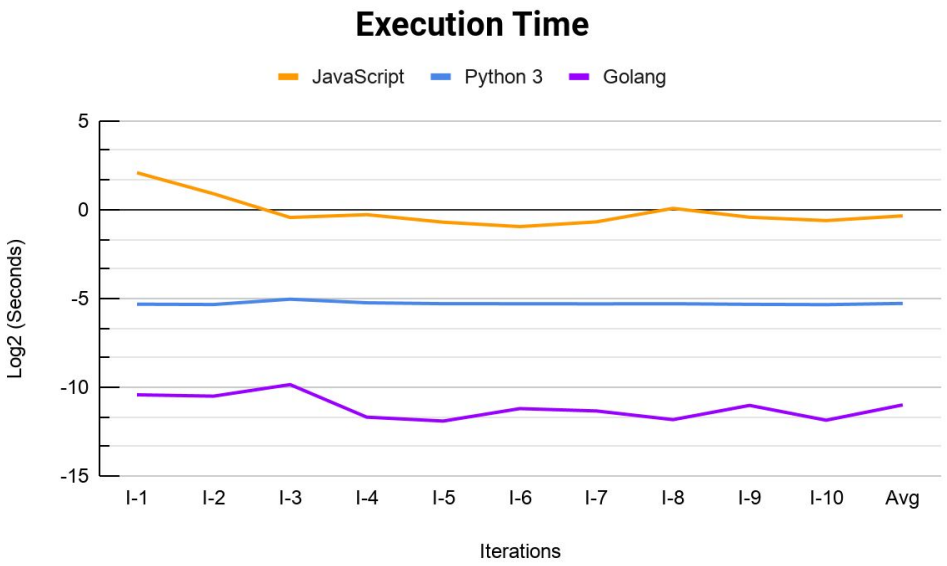

Figure 4.3

Winner: Go

B. Memory Usage

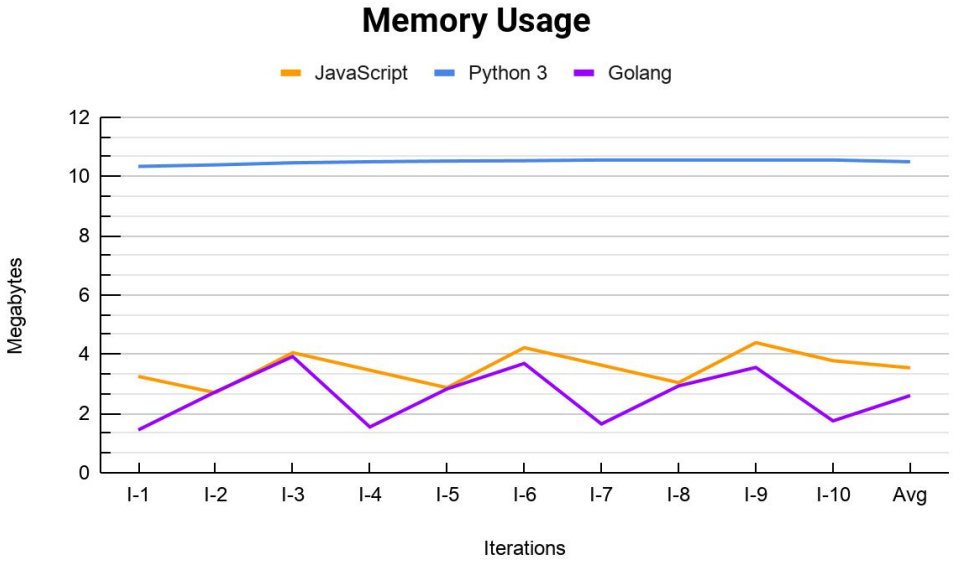

Figure 4.4

Figure 4.4 shows how many megabytes of memory each implementation uses when run. Like execution time, Python3 remains very consistent with its metrics. JavaScript and Go use similar amounts of memory and have an interesting cyclic pattern that repeats every 3 iterations.

Winner: Go

C. CPU User Usage

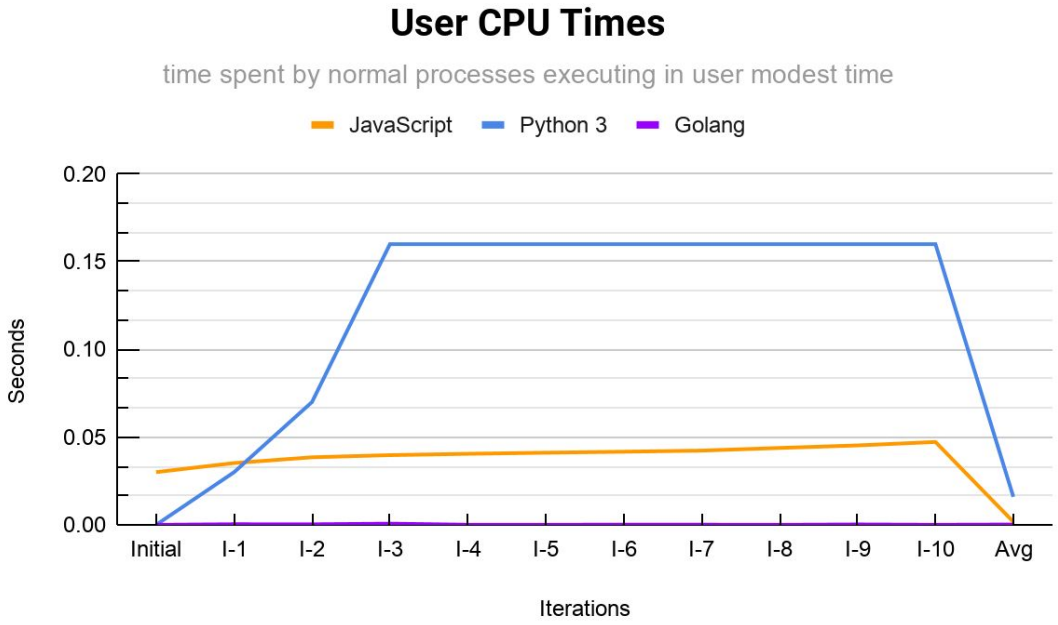

Figure 4.5

Figure 4.5 above shows how much time each Simon implementation spent while normal processes were executing in user modest time. Unlike the other graphs which show the metrics for a single iteration, this Figure 4.5 shows the cumulative buildup of time from the initial state to the final state. The average metric shows the average time taken by a program in a single iteration. It is interesting to see how Python3 ramps up exponentially over the first 3 iterations and then plateaus. Go outperformed the consistent JavaScript implementation by a wide margin.

Winner: Go

D. CPU System Usage

Figure 4.6 shows how much time was spent by processes executing in kernel mode with respect to the running implementation. Like the previous CPU figure, this figure shows the cumulative time buildup with the final data point being the average time spent by a single iteration.

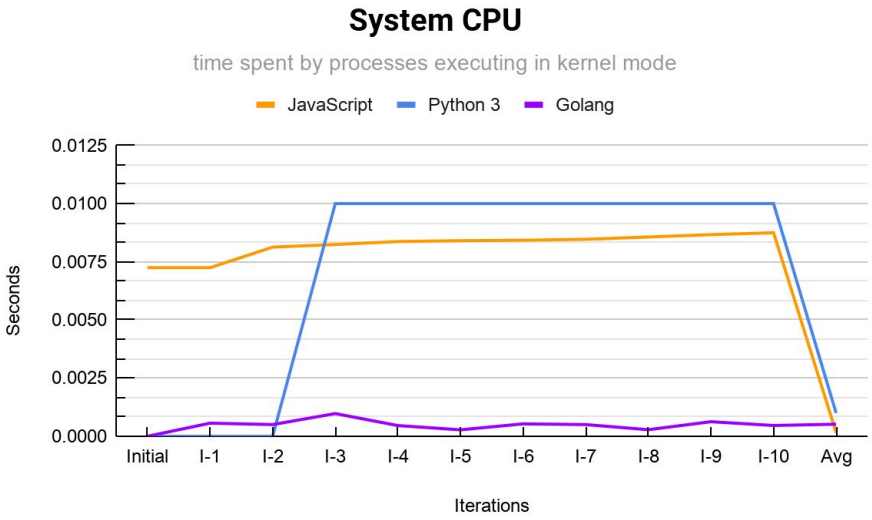

Figure 4.6

Python has a similar trend as its performance with user CPU, increasing exponentially for the first couple of iterations then plateauing off and becoming more consistent. Compared to the user CPU, Go is much more inconsistent but outperforms Python3 and JavaScript.

Winner: Go

E. Program Size

In addition to measuring time, space, and CPU measurements, we also calculated 3 metrics to evaluate the size of the program. These metrics include: lines of code, raw compression, and disk compression.

Lines of Code Ascending: Python3 (189), Go (245), and JavaScript (276)

Lines of code are heavily reliant on the programmer, we consider a statistical significance for this metric ± 15% of other implementations. Therefore, Python3 is statistically better for producing fewer lines of code (and more

readable in our opinion), and Go and JavaScript have a negligible difference in lines of code.

To measure a raw compression rate, we took the original byte size and divided it by the compressed byte size (as can be seen in Figure 4.7). A higher compression factor means the file compressed down more.

$$Compression\ rate\ =\ \frac{Original\ byte\ size}{Compressed\ byte\ size}$$

Figure 4.7

Raw Compression Rates Descending: Go (3.05x), Python3 (2.9x), JavaScript (2.8x).

Like lines of code, compression is highly biased by a programmer's coding and language style. We consider a statistical significance for this metric ± 15% of other implementations. Therefore, all of our implementations tie for this metric.

The last metric measured was disk compression. Every file had an original disk size of 8KB and compressed to 4KB.

Winners
Lines of Code: Python3
Raw Compression: 3-way tie (statistical insignificance)
Disk Compression: 3-way tie

V. Conclusion

Security is a vital component of any software system with any level of reliability in today's world. In this age of the *Internet of Things* where there is a continuous integration of software in everyday items, we are becoming surrounded by Wifi-enabled devices. In our opinion, the best cryptosystem is the one that goes unnoticed. For a device to have a cryptosystem that accomplishes this, it will be fast, use little memory, and in one word, be lightweight. The current best-known cryptosystem that can be successfully implemented given these metrics is the Simon Lightweight Block Cipher using specification 128/256. Additionally, the best language tested here to maximize its potential is Golang. The language Go bested the other languages, Javascript and Python3, in virtually every metric. It required several orders of magnitude less execution time, memory, and energy, to complete a full set of encryption and decryption using the Simon cryptosystem. These results are consistent with our intuition when starting the research due to Go serving as a statically typed, compiled programming language. Meanwhile,

Python is an interpreted, high-level language and similarly, Javascript is a high-level language. Thus, these high-level languages have an inherent performance disadvantage that the creators of Go were able to avoid in its creation.

There were several complications in implementing Simon across three languages. The first language that we created the cipher in was Python3 which we were most familiar with going in. The main challenge of implementing Simon in this language was that there was no reference to go off of. This is a common problem when implementing security systems in general, is that the desired outputs appear to be random; in debugging it is very difficult to discern why the current random output is different from the desired random output. Also, trying to replicate the cryptosystems across three languages posed other challenges. For example, while writing Python3 we leverage how easy it is to switch between types. We converted hex strings, to binary lists, back to hex strings seamlessly in Python3, yet the other two languages had less built-in type switching to allow this. Go especially caused severe issues when trying to accomplish this type switching but this could have been mitigated if we started with a different initial design. Furthermore, we took quite drastic measures to ensure the programs were as similar as possible, and if we used types that Go and Javascript were better at, like their binary representation, we could have also avoided some challenges in writing in these languages.

There are several reasons why Go is advantageous over the others in writing Simon 128/256. The high-level languages, Python and Javascript, abstract many facets of memory to the point it is hard to control how much you are using. Meanwhile, Go has a very aggressive garbage collection system to avoid any excess memory. This includes having any unused variable in the code is an error in Go. Additionally, Go was created in the multicore era of computing and was implemented in a very efficient way. It tries to keep the efficiency of C while providing the readability of Python. I believe that after some initial hurdles with the Go implementation, it was very easy to use. It's built-in data structures give it similar ease of use as Python, yet it retains all the performance advantages. In conclusion, Simon 128/256 is an important security standard in the modern digital world and a modern language like Go could highlight all of the best features of it.

VI. References

A. The Simon and Speck Families of Lightweight Block Ciphers. Click here
B. The official Go website. Click here
C. The official Python website. Click here
D. The official Javascript website. Click here