s2n-tls Benchmarking and Comparative Analysis

CS4991 Capstone Report, 2023

Justin Zhang
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
jmz8rm@virginia.edu

ABSTRACT
As an intern for Amazon Web Services (AWS), I designed previously nonexistent simple and reliable comparative benchmarks for AWS's s2n-tls and other common Transport Layer Security (TLS) libraries, identifying areas for optimization and ensuring that s2n-tls is performant. s2n-tls handles hundreds of millions of connections per second, making any small optimization result in massive cost savings. The benchmarking harness adapts each library (s2n-tls, OpenSSL, and Rustls) to a common interface and measures handshake latency, throughput, and memory usage. s2n-tls was found to be more performant than Rustls and OpenSSL but at the cost of a higher memory usage than Rustls, making memory a possible target for optimization. Future work includes incorporating benchmarks in testing to prevent performance regressions before deployment, more granular testing to get more specific insights, and testing with more parameters.

1. INTRODUCTION

TLS is a network protocol that ensures two endpoints (ex. your computer and a web server) communicate securely. TLS has two main goals: authentication and encryption. Authentication is the verification of an endpoint's identity, which prevents a bad actor from pretending to be the server a client might want to talk to. Encryption protects the security of data in transit, which prevents man-in-the-middle attacks, where a bad actor would read but not necessarily alter data transferred between a client and a server.

Computers use implementations of TLS to establish connections using this protocol. Each implementation might do this a little bit differently and thus have different performances, security risks, etc. For example, an implementation could decide to handle incoming data in series or in parallel. The terms "TLS implementation" and "TLS library" are generally interchangeable and used as such in this report. One of the most common TLS implementations is OpenSSL, which is the de-facto open source TLS implementation that comes with almost all distributions of Linux. AWS also has their own open source implementation of TLS, s2n-tls, in part due to past security vulnerabilities in OpenSSL.

TLS powers most of the Internet, encrypting around 80% of general Internet traffic and more than 95% of web traffic (Zvik & Null, 2023). Cloud services, such as AWS, especially rely on TLS, since their services must be accessed via networking through the Internet. s2n-tls itself powers hundreds of millions of connections per second. Any change in performance to a TLS library such as s2n-tls thus has a broad impact on cloud performance in general.

My internship project was to benchmark s2n-tls against two other common TLS implementations, namely OpenSSL and Rustls, to ensure s2n-tls was performant and to identify possible areas of optimization where other libraries might be doing better.

2. RELATED WORKS

There are benchmarks that show that Rustls is more performant than OpenSSL on essentially all metrics (Biff-Pixton, 2019). These benchmarks used a similar methodology to mine but are not as well-documented or extensible. They also do not benchmark s2n-tls, which was important to my team.

My team also has custom benchmarks that simulate real network conditions, changing latency, packet loss, and other parameters (AWS, 2023). It uses the whole networking stack with OS sockets, etc., so the numbers from those benchmarks are representative of what a user of the library might see. These benchmarks are thorough, but they are relatively hard to configure, slow to iterate on, and have a lot of external factors. Because of this, there was a gap in benchmarks that were simple, fast, and easy to use, which this project aimed to fill.

A sister team at AWS has benchmarks comparing the performance of different cryptography libraries, which all TLS libraries call and use. Since most TLS computation occurs during the calls to underlying cryptography libraries, I expected the performance of each TLS library to correlate with the performance of their respective underlying cryptography libraries.

3. PROJECT DESIGN

I was tasked with benchmarking s2n-tls against OpenSSL and Rustls specifically. OpenSSL is the main alternative to s2n-tls internally at AWS; Rustls, on the other hand, is not as widely used (ex. it has <25% of GitHub stars that OpenSSL has), but as mentioned above, there are benchmarks that show it is more performant than OpenSSL in almost every way.

The three main metrics benchmarked were handshake latency, how fast a TLS connection could be established; throughput, how much data can be encrypted and decrypted over a TLS connection in a given amount of time; and memory, how much space the data associated with a connection takes in RAM. These are all important, since a reduction in any one of these could translate to massive cost savings for the company.

The application programming interfaces (APIs) for s2n-tls, OpenSSL, and Rustls are all very different. My first task was to create a wrapper around each of the libraries that has a common API for ease of use and testing. s2n-tls and OpenSSL have mainly APIs in C, a low-level programming language, while Rustls only has an API in Rust, another low-level programming language. Since s2n-tls and OpenSSL also have Rust APIs (albeit much less frequently used), the benchmarking suite also used Rust.

The benchmarking framework used for measuring handshake latency and throughput was Criterion.rs. Criterion.rs is the de-facto standard for benchmarking in Rust and provides useful functionality for collecting, analyzing, and visualizing data. For memory benchmarking, I used Valgrind and its subtool Massif, which profiles memory usage throughout the runtime of a program, and visualized the data myself with Plotters, a plotting library written in Rust.

All benchmarks were run in a single CPU thread with no concurrency, which is very different from how a typical user would use the s2n-tls library (which would be asynchronously with concurrency). All data was also sent between a mock client and server over local buffers (not over an actual connection) and thus sent much faster than it would typically be. Because of this, the absolute numbers from the benchmarks are not accurate, but external factors were reduced, and relative performance between libraries was more accurate.

During preliminary testing, I found that a large factor of TLS performance is which cryptography library is being used. For example, s2n-tls could use either OpenSSL's cryptography library or AWS's own library, AWS-LC, which was 3-4x faster for some sets of parameters. Cryptography library performance optimizations were out of the scope of my project, so I decided to mainly analyze the performance of each TLS library as most commonly used: OpenSSL with its native cryptography library, s2n-tls with AWS-LC, and Rustls with *ring*, its default.

OpenSSL also has many different versions that are still widely used. OpenSSL 1.1.1 is the main alternative to s2n-tls used internally in AWS; OpenSSL 3.0.0 is the default on Ubuntu, which is the operating system I was using; and OpenSSL 3.1 is the newest (and fastest) version. To align with the goal of comparing these libraries as most commonly used, I decided to focus on OpenSSL 1.1.1, since it was the most common version at AWS.

Finally, I also created historical benchmarks, where older versions of s2n-tls are benchmarked against the newer versions to identify performance optimizations or regressions. Due to design limitations, I only made historical benchmarks for handshake latency and throughput (and not memory). I also could only benchmark versions after about a year ago, since any versions before that had a different API that was incompatible with the benchmarking suite I made.

4. RESULTS

All benchmarking results were consistent, reproducible, and easy to run, needing only a few commands (if not only one command) to generate effective visualizations in minutes. I tested many different CPU architectures and cloud instance types, with the benchmarks working successfully on all of them.

I found that the signature algorithm (ECDSA or RSA) used for authentication was the most important factor in performance. s2n-tls was 3-4x faster than both Rustls and OpenSSL with ECDSA, while performance was similar between the libraries with RSA. ECDSA was also faster than RSA for all libraries when similar parameters were chosen for both. Other factors (like handshake type and key share algorithm) had no interesting impacts on handshake performance or differences in performance between the libraries.

For throughput, the results were a little less drastic but still a win for s2n-tls. s2n-tls was measured to have 15-20% higher throughput than either Rustls or OpenSSL. This was true for all parameters tested.

Memory was the only metric where s2n-tls was not dominant. OpenSSL used over twice the memory of s2n-tls, which in turn used over twice the memory of Rustls. This means that memory usage is a prime target for optimization. However, after analyzing which parts of s2n-tls and Rustls were using the most memory and talking with the rest of the team, we concluded that there were no low-hanging fruit for memory optimizations.

Finally, with historical benchmarking, I only found one significant change in performance over the past year, which was an around 15% speedup in throughput; otherwise, handshake and throughput performance was relatively consistent. The version where this speedup occurred had enabled a compile-time optimization for only the Rust API of s2n-tls and left the core C library untouched, meaning that for most users of s2n-tls, there was no speedup from that version. However, it shows us that the historical benchmarks can successfully catch performance changes.

5. CONCLUSION

Overall, my project filled a gap in s2n-tls benchmarks, being simple, fast, and reliable. The API I designed for the s2n-tls library for the benchmarks was elegant, with my team planning to replace all existing Rust tests with ones that use the API I designed. I also produced useful and actionable results that my team did not have before: s2n-tls generally outperforms Rustls and OpenSSL, with only memory not being so, where Rustls uses less memory than s2n-tls.

I also personally learned a lot over the course of working on this project. This being my first time in industry, I experienced the software design process, working with a team, and writing code at a very high standard. I realized why I enjoyed software engineering: its fast pace, continual learning, and immediate results.

6. FUTURE WORK

As with a lot of benchmarking projects, there are many things that could be extended to improve this project. The most impactful few things would be to incorporate the benchmarks into testing to catch performance regressions earlier, to test more libraries, and to test more parameters.

Some smaller but still meaningful extensions to the project could include: separating the client and server halves of a TLS connection and benchmarking each separately; reducing external factors and noise in the results; testing the performance of a resumed TLS connection, where a lot of the cryptographic computation has already happened in a prior terminated connection; varying the amount of data sent during throughput testing; and varying the certificate chain length used for TLS authentication. These possible improvements and more are all tracked in GitHub issue #4157 in the s2n-tls GitHub repository (Zhang, 2023).

7. ACKNOWLEDGMENTS

First and foremost, I would like to thank my mentor during my internship, James Mayclin, for thoughtfully and patiently answering every question I had. I attribute most of deep understanding of and passion for my project to him.

Next, I would like to thank my manager, Wesley Rosenblum, for always being understanding and flexible, as well as being a great resource and role model. He handled most of the logistics and paperwork during my internship and enabled me to have such a fulfilling summer experience.

Finally, I would like to thank the whole Transport Libraries team that I had the honor of being a part of. I had the pleasure of collaborating with everyone on the team at some point. They were always excited to see my new results, and I felt as if what I was doing mattered.

Since s2n-tls is open source, we can see all the code online; it is fulfilling to see my team still building on my project to this day! I could not have asked for a better internship, and I hope our paths cross in the future.

REFERENCES

AWS. (2023, March 3). *s2n-quic/netbench at main · aws/s2n-quic*. GitHub. https://github.com/aws/s2n-quic/tree/main/netbench

Birr-Pixton, J. (2019, July 1). *TLS performance: Rustls versus OpenSSL*. jbp.io. https://jbp.io/2019/07/01/rustls-vs-openssl-performance.html

Synopsys. (2020, June 3). *The heartbleed bug*. The Heartbleed Bug. https://heartbleed.com/

Zhang, J. (2023, August 18). *Possible improvements to the benchmarking suite · Issue #4157 · aws/s2n-tls*. GitHub. https://github.com/aws/s2n-tls/issues/4157

Zvik, E. W., & Null. (2023). Traditional Firewalls Can't Keep Up with the Growth of Encrypted Traffic. *Cato Networks*. https://www.catonetworks.com/blog/traditional-firewalls-cant-keep-up-with-the-growth-of-encrypted-traffic/