Comparing the Performance of Open-Source Cryptographic Libraries

A Technical Report Submitted to the Department of Computer Science

Department of Computer Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements of the Degree
Bachelor of Science, School of Engineering

Zachary Goldstein

Spring, 2021

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed: _____ Date _____

Zachary Goldstein

Approved: _____ Date _____

John A. Stankovic, Department of Computer Science

Comparing the Performance of Open-Source Cryptographic Libraries

Zachary Goldstein
Department of Computer Science
University of Virginia
Charlottesville, VA United States
of America
zng8tb@virginia.edu

ABSTRACT

This paper is a survey and comparative analysis of various open-source cryptographic libraries. This paper introduces popular open-source cryptographic libraries for the C programming language and the algorithms and functionality that they make available. Runtime analysis of memory usage and execution time on commonly-used algorithms highlights the comparative performance differences across multiple cryptography APIs. Comparative performance analysis between algorithms is well-researched, but performance between cryptographic libraries' implementations is relatively unexplored. These implementation differences, especially when a library is developed to target a particular use case, have significant effects on the runtime performance of applications. This research provides application developers with the insight to better understand the tradeoffs between open-source libraries in their specific applications.

1 Introduction

Modern communication relies on the use of the Internet to transmit information between users. To ensure that information is forwarded between peers securely, cryptography is used to encrypt information before it is sent across the Internet. Failing to properly secure data both in storage and in transit risks leaking information to bad actors and can jeopardize people's information and the digital economy as a whole. Due to the risks associated with writing flawed cryptographic implementations, many organizations and projects rely on existing cryptography APIs.

Multiple open-source cryptography libraries are available to encrypt and decrypt data and establish secure connections between parties. There are at least 19 different cryptography-related libraries for the Python programming language alone. Some libraries are focused on specific algorithms while others are full-featured and have support for SSL and TLS connections. Additionally, various cryptographic libraries have different API styles and algorithm implementations that affect runtime performance and take different approaches to improving security and usability of their APIs. Developers are tasked with the challenge of choosing the appropriate cryptographic library for their needs. For example, OpenSSL, a popular, full-featured cryptographic library, might not have enough resources to run properly on an embedded device with memory constraints.

To understand the effect of these implementations for different use cases, it is necessary to analyze what the libraries offer and what results they can achieve. By comparing runtime performance of commonly-used algorithms by four popular cryptography libraries, this research provides the groundwork to help implementers better understand the general strengths and weakness of the libraries and provide directions for further research.

2 Related Work

There is past research dedicated to analyzing the utility and performance of cryptography APIs. A comparative analysis of cryptographic libraries looked into the algorithms, features, and platforms that popular libraries supported. The study analyzed 22 different cryptography libraries, including 5 targeted for IoT. It found the non-existence of a universal library for IoT devices due the various features and optimizations that different libraries make for specific platforms [1].

The APIs of Python cryptographic libraries has also been studied to determine how usability (e.g., documentation, functionality) affects the security of developers' implementations of those libraries for various use cases. Acar et al. found that clear APIs and feature availability had significant impacts on developers' ability write cryptographically secure programs. Libraries that had more documentation available and simplified the encryption process had better security overall [2].

Finally, there has been limited research into the speed performance of different cryptography libraries across serveral asymmetric and symmetric ciphers, Linux distributions, compiler flags, and CPU architectures [3]. This research provides an understanding of how fast libraries' implementations can process data as the size to encrypt and decrypt increases. The study found that OpenSSL's implementations were the fastest, followed by Beecrypt and Tomcrypt [3]. However, the study was conducted in 2008 and does not contain results for key generation algorithms nor newer libraries or forks thereof.

3 Study Design

This study uses a suite of programs to compare how effectively popular open-source libraries perform in common cryptographic tasks – for example, including key generation, hashing, encryption, and certificate generation. This study intends to adhere to how developers might use each of these libraries in their own implementations. Appropriate initialization and cleanup functions were called to mimic practical real-world usage – for example, ensuring that keys erased from memory. Recommendations to use higher-level APIs described in the documentation were followed if applicable.

3.1 Library Selection

To determine which open-source cryptography libraries to analyze, a series of searches were performed to identify a set of libraries that were in popular use. Although cryptographic libraries have been written in multiple programming languages, such as C/C++, Java, C#, etc., the libraries were chosen primarily for their native support for the C programming language. C is low-level, efficient, and can be retargeted to multiple platforms, which makes it valuable for developers looking to add encryption to their applications.

The study selected four open-source libraries that are actively being developed. The analysis includes OpenSSL, BoringSSL, Libgcrypt, and wolfSSL. Although wrappers for some of these libraries exist, such as pyOpenSSL binding to OpenSSL, the analysis is on the original programming language implementation.

OpenSSL. OpenSSL is one of the most popular open-source cryptographic libraries. It is a multi-platform library that provides a secure sockets layer (SSL) and transport layer security (TLS) protocols in addition to numerous encryption protocols. OpenSSL is considered the de facto industry standard cryptography library and used in two-thirds of web servers [4]. It is licensed under Apache License 1.0 and BSD License. The popularity of OpenSSL led to the development of multiple forks, notably LibreSSL and BoringSSL.

BoringSSL. BoringSSL is a fork of OpenSSL by Google due to the increasingly large number of upstream patches that Google developers had made to OpenSSL. BoringSSL was chosen because its development team has made significant changes to OpenSSL, such as removing dead/legacy code and insecure algorithms, to tailor it for Chrome/Chromium and Android. BoringSSL shares many of the same API calls as OpenSSL, but Google does not recommend general usage. However, other libraries, such as the Envoy Proxy, have developed their code bases around BoringSSL to take advantage of Google's improvements [5]. BoringSSL is licensed under Apache License 1.0 and BSD License, the same as OpenSSL.

Libgcrypt. Like OpenSSL, Libgcrypt is a popular open-source and multi-platform cryptographic library. It was chosen for its reputation for being secure and its entropy gathering utility. However, Libgcrypt entirely focuses on cryptographic algorithms and does not have implementations for SSL or TLS and cannot be used to generate certificates. Libgcrypt is licensed under the GNU Lesser General Public License (LGPLv2.1+) and GNU General Public License (GPLv2+).

WolfSSL. WolfSSL ("wolfSSL"), formerly known as CyaSSL, is a cryptographic library for applications that are targeted for environments with where library footprint size or memory usage is important. These include embedded systems or cloud environments where memory usage per connection is a concern. WolfSSL claims to have a build size between 30-100 kB with support for a full-featured SSL library and TLS client and server support [6]. Other IoT-focused cryptography libraries exist, such as TinyECC and AvrCryptoLib, but wolfSSL is the only one known to serve as a general-purpose cryptography library similar to OpenSSL. WolfSSL is licensed under the GNU General Public License (GNUv2) and has a standard commercial license.

3.2 Algorithm Selection and Program Design

To understand how each library performs in terms runtime speed and memory efficiency, each library was tested under the following algorithms: RSA key generation, DSA key generation, elliptic curve P-256 key generation, SHA-256 hashing, AES encryption using cipher block chaining (CBC), and X.509 self-signed certificate creation. These were chosen because of their pervasiveness and popular usage in data encryption or Internet security. All of the chosen libraries support these algorithms with the exception that Libgcrypt does not support X.509 certificate creation.

RSA. The RSA algorithm was tested across all platforms using 2048-bit key generation. NIST considers 2048-bit keys for RSA as the minimum acceptable length to be cryptographically secure [7]. All of the key generation programs generate keys through random number generation. The programs do not write the keys to the terminal or a file in order to eliminate the time needed for I/O operations. OpenSSL and BoringSSL use the same function calls through the RSA low-level interfaces. The EVP wrapper is available for these libraries, but the OpenSSL documentation does not explicitly recommend that it is used unlike other key generation algorithms. RSA in Libgcrypt was computed using s-expressions. WolfSSL's implementation was done through its RSA API function calls. The wolfSSL source code was configured and built with key generation enabled (--enable-keygen).

DSA. Similar to RSA, the DSA algorithm was tested across all libraries using 2048-bit key generation. This is the minimum acceptable key size by NIST [7]. The programs only tested key generation and did not include signature or verification functions. All of the key generation programs used random number generation and did not write the keys to the terminal or a file. OpenSSL and BoringSSL followed the same structure as RSA and did not generate the keys through the EVP interface. Libgcrypt's implementation was done through s-expressions. Finally, WolfSSL used its default DSA library with key generation (--enable-keygen) and DSA (--enable-dsa) enabled when building and configuring the library.

Elliptic Curve. The elliptic curve algorithm was tested using the NIST Curve P-256 (or secp256k1) for key generation. All of the key generation programs used random number generation and did not write the keys to the terminal or a file. OpenSSL and

BoringSSL elliptic curve keys were generated through the EVP wrapper rather than the low-level elliptic curve library. Libgcrypt's implementation was done through s-expressions. WolfSSL used its default elliptic curve library with key generation enabled (--enable-keygen) when building and configuring the library.

AES. The advanced encryption standard (AES) was tested with a 128-bit block size and cipher block chaining (CBC) mode. All of the encryption programs used the same key and initialization vector, which was stored in a static variable rather than loaded from a file or the command line at runtime. Each of the programs encrypted a 4 KB file filled with randomized bytes and was read into a memory during runtime. This file size ensures that no padding is necessary for proper encryption. The file remained constant for all iterations and libraries. OpenSSL and BoringSSL both used the high-level EVP wrapper to encrypt the file. OpenSSL has low-level API functions for AES, but the documentation recommends that that application programs use the high-level EVP interface. AES encryption in Libgcrypt was done through the cipher API. WolfSSL's implementation was done through the default AES API.

SHA-256. The SHA-256 hash function was tested similarly to AES. A 4 KB file of randomized bytes was loaded into memory during runtime and hashed by each program. The file is the same as used in the AES test. OpenSSL and Boring SSL both used the high-level EVP wrapper to hash the file. Like AES, OpenSSL allows access to the low-level functions, but recommends that applications use the higher-level EVP digest functions. In libgcrypt, the message digest API was used to access the hash function. Finally, the default SHA-256 functions were used for wolfSSL.

X.509. X.509 certificate functionality is an important part of SSL implementation. Programs were developed for all libraries except libgcrypt since it has no certificate generation functions. For each of the programs, a 2048-bit PEM-encoded RSA private key was loaded from a file into memory to self-sign each of the certificates. Only the country, organization, and common name fields were set. Finally, each of the tests wrote the PEM-encoded certificate to a file. WolfSSL uses the DER format to load RSA keys, and the key was converted during execution. Moreover, the wolfSSL library was built with certificate generation enabled (--enable-certgen).

3.3 System and Compilation

Each test program was executed on an Ubuntu 18.04 LTS virtual machine environment inside a native Windows 10 environment. All tests were conducted using an Intel 6700K Skylake CPU @ 4.0 GHz. The task-clock time was measured using the Performance Counters for Linux (perf) tool and the maximum resident size was measured using the bash time command. Each algorithm program was compiled using gcc and level 2 optimization (-O2 flag).

3.4 Limitations

Results should be interpreted in context. There is no 1-to-1 method to compare timing and memory usage of these programs. The focus of the programs is on the algorithms themselves, but it is possible that there is significant time or memory usage during the setup or teardown steps of the programs. Additionally, OpenSSL, BoringSSL, and Libgcrypt have multiple options to access the cryptographic functions. The choice of implementation may have effects on performance. For some of the programs, the documentation-recommended implementation was used, which may reduce performance compared to the low-level interfaces. X.509 default certificate generation is also different. OpenSSL and BoringSSL generate a version 1 certificate by default while wolfSSL generates a version 3 certificate. No certificate extensions were used in wolfSSL, but there may be slight performance differences.

4 Results

Cryptographic libraries performed at notably different rates of execution time and maximum resident size. Each program was executed and analyzed 30 times to ensure a proper sample size for comparative analysis. Overall, either OpenSSL and BoringSSL performed the best in execution time and BoringSSL performed the best in terms of maximum resident size.

RSA. As seen in Table I below, OpenSSL outperforms the other libraries in execution time. However, this comes at a steep time-memory tradeoff, as OpenSSL had the largest maximum resident size of 3845.7 KB. It is surprising that BoringSSL, being a fork of OpenSSL, reduces the maximum resident size significantly in exchange for a longer average runtime. It is interesting that wolfSSL and BoringSSL have similar performance metrics in both runtime and memory usage.

Library	Benchmark	Mean	Std Dev
OpenSSL	time (ms)	78.3	36.6
	mem. (kB)	3845.7	68.1
BoringSSL	time	191.4	150.3
	mem.	1975.7	62.7
WolfSSL	time	190.8	87.9
	mem.	2278.0	95.5
Libgcrypt	time	209.5	105.3
	mem.	2479.6	54.9

Table I. Benchmarks for 2048-bit RSA key generation

DSA. DSA 2048-bit key generation shows a similar pattern to RSA key generation. As seen in the table below, OpenSSL has a significantly better average runtime performance at the cost of a larger maximum resident size. BoringSSL had the minimum average size, also as in RSA key generation.

Library	Benchmark	Mean	Std Dev
OpenSSL	time (ms)	359.7	259.2
	mem. (kB)	3470.1	67.0
BoringSSL	time	615.0	287.9
	mem.	2165.3	56.8
WolfSSL	time	929.9	920.7
	mem.	2287.5	95.8

Libgcrypt	time	806.3	522.0
	mem.	2493.3	75.2

Table II. Benchmarks for 2048-bit DSA key generation

Elliptic Curve. Computing for the P-256 (secp256k1) curve, BoringSSL performed much better than OpenSSL in both time and size. This may indicate some optimizations to the library through the EVP interface or curve computations. It is also very surprising that Libgcrypt had a very slow task time and it is unclear why this occurred.

Library	Benchmark	Mean	Std Dev
OpenSSL	time (ms)	0.79	0.04
	mem. (kB)	3619.87	188.93
BoringSSL	time	0.39	0.03
	mem.	1883.47	48.24
WolfSSL	time	1.33	0.07
	mem.	2297.87	83.65
Libgcrypt	time	12.31	0.33
	mem.	2420.53	74.15

Table III. Benchmarks for Elliptic Curve P-256 key generation

AES. BoringSSL once again outperformed OpenSSL in both time and memory metrics. Surprisingly, OpenSSL showed significantly worse performance compared to all other libraries both in time and memory usage.

Library	Benchmark	Mean	Std Dev
OpenSSL	time (ms)	0.65	0.07
	mem. (kB)	2686.27	77.43
BoringSSL	time	0.37	0.03
	mem.	1771.60	71.96
WolfSSL	time	0.44	0.04
	mem.	2053.53	185.00
Libgcrypt	time	0.49	0.05
	mem.	2302.80	86.48

Table IV. Benchmarks for encrypting a 4 KB file with AES 128-bit blocks and CBC mode

SHA-256. BoringSSL, wolfSSL, and Libgcrypt all achieve similar runtime performance in SHA-256 hasing. Similar to the AES results, OpenSSL resulted in a higher time and memory usage compared to all other libraries. Moreover, Libgcrypt achieved had the smallest average maximum resident size with 1605 KB.

Library	Benchmark	Mean	Std Dev
OpenSSL	time (ms)	0.62	0.04
	mem. (kB)	2581.33	76.27
BoringSSL	time	0.35	0.04
	mem.	1708.80	64.12
WolfSSL	time	0.40	0.06
	mem.	1929.33	56.91
Libgcrypt	time	0.37	0.03
	mem.	1605.33	49.11

Table V. Benchmarks for hashing a 4 KB file with SHA-256

X.509. X.509 certificate is not available on Libgcrypt. Surprisingly, the results do not show a space-time tradeoff for certificate generation. As seen in Table VI, OpenSSL achieved the slowest time and had the highest memory usage, while BoringSSL performed with the fastest time and lowest memory usage.

Library	Benchmark	Mean	Std Dev
OpenSSL	time (ms)	0.65	0.07
	mem. (kB)	2686.27	77.43
BoringSSL	time	0.37	0.03
	mem.	1771.60	71.96
WolfSSL	time	0.44	0.04
	mem.	2053.53	185.00

Table VI. Benchmarks for X.509 self-signed certificate creation

The metrics reveal that OpenSSL is not the most optimal cryptography library in every algorithm. OpenSSL exceled in RSA and DSA key generation, but was significantly slower than the other libraries in AES and SHA-256 hashing. While OpenSSL does have support for a wide variety of algorithms and backwards compatibility with other versions, that comes at the of higher memory usage. As illustrated by Figure I, OpenSSL's maximum resident set size was consistently higher than the other libraries.

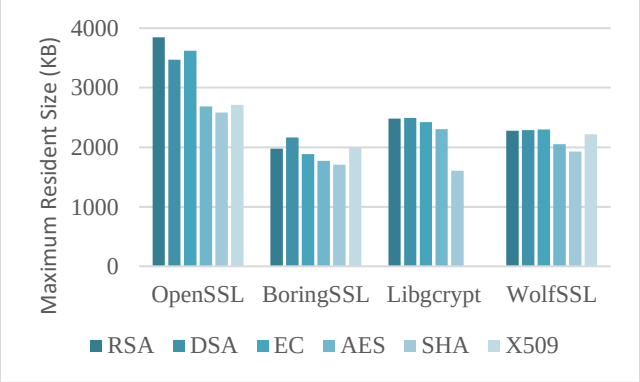

Figure 1. Comparison of maximum resident set size across all libraries and algorithms

In comparison, it is very surprising that BoringSSL performed significantly differently from OpenSSL despite being forked from the project. BoringSSL outperformed OpenSSL in numerous tests and had the minimum resident set size in all but SHA-256 hashing. This indicates that Google has significantly optimized

BoringSSL – for example, removing insecure and deprecated functionality and unnecessary legacy code.

Finally, the metrics show that wolfSSL and Libgcrypt performed fairly similarly across all of the tests with the exception of EC P-256 key generation. Given that wolfSSL is designed with the embedded devices in mind, the results were surprising as wolfSSL performed worse than BoringSSL in terms of maximum resident set size.

5 Conclusions

This research shows how OpenSSL, BoringSSL, Libgcrypt, and wolfSSL have significantly different performance results despite completing the same tasks. Although OpenSSL is commonly referred to as the de facto industry standard, the results reveal that OpenSSL is far from a universal library.

Compared to OpenSSL, BoringSSL had significantly better memory performance across the board (Figure 1) and better speed performance in 4 of the 6 tasks. Despite forking from the same project, BoringSSL has a much smaller codebase of 349 K lines compared to OpenSSL's 675 K lines [8, 9]. This cleanup may be part of the reason the maximum resident set size is smaller for BoringSSL. Google recommends that projects do not target BoringSSL because it is not intended for general use and may have API or ABI instability. However, projects that do not intend on supporting older standards and are willing to update their projects to match BoringSSL's development should consider this lightweight fork of OpenSSL as their library of choice.

This research also reveals that Libgcrypt and wolfSSL had decent memory usage advantages over OpenSSL, but were generally slower than either OpenSSL or BoringSSL. It is very surprising that wolfSSL did not achieve the smallest average maximum resident set size given its emphasis on targeting for embedded devices and cloud environments. WolfSSL has build options to create a smaller static binary for these applications, but the test results revealed that the runtime memory performance was consistently worse than BoringSSL.

Taken together, these results illustrate the need for thorough assessment by developers before choosing the best cryptographic library for their applications if runtime performance is a key priority. However, developers should also consider factors other than performance – for example, security history, usability, and functionality.

6 Future Work

Although this study provides insight into the runtime performance of these open-source cryptography libraries, more research should be conducted to extend the analysis holistically. Looking forward, further research is needed to evaluate libraries more extensively and include more libraries overall.

First, libraries should also be further tested on a wider variety of CPUs and hardware devices. The testing system for this research had desktop-level resources, which is not the intended platform for wolfSSL. Additional benchmarks on ARM and other x86 CPUs should be conducted.

Second, test programs should be extended to cover more algorithms and parameters (e.g., key size). Analysis of DSA can be extended to cover signatures and verification. For algorithms that act on variable data sizes – for example, SHA-256 and AES – more research is necessary to understand the impact of startup time as the file size increases. Testing configuration flags when building the libraries from source is needed to understand the binary size needed for certain applications.

Finally, the analysis should be extended to cover more cryptography libraries. Specifically, LibreSSL, another fork of OpenSSL, and libsodium should be pursued to compare with existing results. Cryptography libraries in other languages may need separate evaluations due to differences in how the code is executed.

REFERENCES

[1] Tuhin Borgohain, Uday Kumar, and Sugata Sanyal. 2015. Comparative Analysis of Cryptography Library in IoT. In *International Journal of Computer Applications* 118, 10 (May 2015), 5-10. DOI:https://doi.org/10.5120/20779-3338

[2] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Strankey. 2017. Comparing the Usability of Cryptographic APIs. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017. DOI:https://doi.org/10.1109/SP.2017.52

[3] Timo Bingmann. 2008. Speedtest and Comparison of Open-Source Cryptography Libraries and Compiler Flags, 2008. Accessed on April 20, 2021. Available: https://panthema.net/2008/0714-cryptography-speedtest-comparison/#c05-s03-distro

[4] Imran Ghafoor, Imran Jattala, Shakeel Durrani, and Mhuammad T Ch. 2014. Analysis of OpenSSL Heartbleed vulnerability for embedded systems. In *17th IEEE International Multi Topic Conference 2014.* DOI:https://doi.org/10.1109/INMIC.2014.7097358

[5] Why does Envoy use BoringSSL? Accessed April 20, 2021. Available: https://www.envoyproxy.io/docs/envoy/latest/faq/build/boringssl

[6] How Does wolfSSL Compare to OpenSSL? 2011. Accessed April 20, 2021. Available: https://www.wolfssl.com/how-does-wolfssl-compare-to-openssl/

[7] Elaine Barker and Allen Roginsky, 2019. Transitioning the Use of Cryptographic Algorithms and Key Lengths. (March 2019). DOI: https://doi.org/10.6028/NIST.SP.800-131Ar2

[8] "OpenSSL Languages Summary" 2021. [Online]. Available: https://www.openhub.net/p/openssl/analyses/latest/languages_summary

[9] "BoringSSL Languages Summary" 2021. [Online]. Available: https://www.openhub.net/p/boringssl/analyses/latest/languages_summary