Predicting TLS performance from key exchange performance (short paper)

Farhad Moghimifar
Queensland University of Technology
Brisbane, Australia
farhad.moghimifar@connect.qut.edu.au

Douglas Stebila
Queensland University of Technology
Brisbane, Australia
stebila@qut.edu.au

ABSTRACT

Most benchmarking of cryptographic systems focuses on the performance of individual algorithms in a standalone setting. However, real-world applications such as the Transport Layer Security (TLS) protocol use a variety of cryptographic algorithms together. Benchmarking the performance of a web server using TLS is a more complex task, so fewer works include performance characteristics of full systems. In this work, we develop a model for the number of connections per second of a TLS-protected web server based on the runtime of individual cryptographic operations. Our model allows us to predict how performance scales with file size. Our model also allows us to predict the impact of improved key exchange algorithms: for example, on an HTTPS server with 1 KiB files running ECDSA-nistp256 with AES-128-GCM and HMAC-SHA-256, a 2× improvement in ephemeral Diffie–Hellman key exchange performance only leads to a 10% improvement in connections per second, as signatures become the dominant cost.

CCS Concepts

•**Security and privacy** → **Security protocols;** *Cryptography;* •**General and reference** → Performance;

Keywords

Transport Layer Security (TLS) protocol; key exchange; performance

1. INTRODUCTION

Improving the performance of cryptographic algorithms is an active area of research, as the runtime characteristics of cryptographic algorithms directly impact on the number of secure connections a given server can simultaneously support.

Since public key operations are generally more expensive than symmetric key operations, a particular area of active research is on fast Diffie–Hellman key exchange using elliptic curves. Internet standards today typically rely on the NIST family of curves, specifically the nistp256 prime field curve [11]. However, there are several alternative curves proposed that are more amenable to high-speed implementations, such as curve25519 [2], FourQ [5], and others.

Researchers can and typically do give runtime measurements of their curve in software. However, it is often difficult to compare results across papers to due to different hardware architectures, different measurement frameworks, and even whether results reported are time or cycles. In order to provide a framework for the systematic comparison of the performance of cryptographic algorithms, Bernstein and Lange developed the ECRYPT Benchmarking of Cryptographic Systems (eBACS) framework [3]. Implementation developers can submit their implementation to eBACS. If the implementation follows a standard API, the eBACS benchmarking tool "SUPERCOP" will run the implementation on a variety of machines and report the results. This allows for the fair comparison of implementations across a range of hardware systems. eBACS includes 31 different elliptic curve Diffie–Hellman primitives.

Even with this framework for measuring the standalone performance of an individual cryptographic primitive, there is still some way to go to be able to characterize the performance of an application using this primitive, such as a web server using the Transport Layer Security (TLS) protocol [6, 7] for secure communication. Performance of a ciphersuite in TLS relies not just on the speed of the key exchange algorithm but also on the signature scheme, key derivation function, and bulk encryption performance; the latter depending on the size of the transmitted data as well. eBACS only measures the performance of individual operations, for example Diffie–Hellman operations in isolation. Measuring the performance of a ciphersuite in a web server requires significantly more effort: one must integrate the cryptographic primitive into an SSL/TLS library (such as the OpenSSL library) which can require changing hundreds of lines of code across dozens of files; then set up a testing network with a server machine and one or more client machines; and then generate sufficient load to saturate the web server and measure performance. As a result of this added complexity, there have been fewer works carrying out a full TLS-level performance analysis. Some works have done so to demonstrate the viability of a particular cryptographic primitive over another (Gupta et al. [8] for elliptic curves over RSA; Bos et al. [4] for ring-learning-with-errors compared to elliptic curves). Bernat [1] reports on the performance of TLS handshakes without forward secrecy versus with forward secrecy.

ACSW '16 Multiconference February 2–5, 2016, Canberra, Australia

ISBN 978-1-4503-4042-7/16/02…$15.00

DOI: http://dx.doi.org/10.1145/2843043.2843360

Our contributions.

In this work, we aim to give a predictive model for the performance of an HTTPS web server based on the runtime of the underlying key exchange. Our method is as follows:

- Extend OpenSSL to quickly support plugging in Diffie–Hellman key exchange algorithms from the eBACS project using the SUPERCOP API.
- Measure the standalone performance of signatures and key exchange to determine the runtime of basic cryptographic operations.
- Measure the performance (in connections per second) of an HTTPS web server using fixed signature and bulk encryption/authentication algorithms, but different key exchange algorithms and application data payload sizes.
- Using linear regression, develop a model that predicts the performance of the web server at different file sizes and with different key exchange algorithms.

Our model describes the runtime of a TLS connection as

$$t_{\text{conn}}(x) = t_{\text{sig}} + t_{\text{dhkg}} + t_{\text{dhss}} + t_{\text{fix}} + c_{\text{bulk}} \cdot x \qquad (1)$$

where t_{sig} is the runtime of the signing algorithm, t_{dhkg} and t_{dhss} are the runtime of the Diffie–Hellman key pair generation and shared secret generation, t_{fix} is a fixed-cost overhead, c_{bulk} denotes a coefficient related to the bulk encryption and authentication of x bytes of application data.

For a fixed ciphersuite, linear regression on file sizes yields a high coefficient of determination (R^2), leading to a highly predictive model of TLS performance in terms of file size.

Fixing the signature algorithm, bulk cipher, and file size, we can also predict the impact of key exchange performance improvements. For example, with ECDSA-nistp256 signatures and AES128-GCM encryption and HMAC-SHA-256 and a file size of x bytes, our model estimates the number of HTTPS connections per second per core as $\widetilde{n_{\text{conn}}} \approx$

$$\frac{1}{0.0005971 + t_{\text{dhkg}} + t_{\text{dhss}} + 0.0005662 + 6.834 \times 10^{-9} \cdot x}. \qquad (2)$$

Substituting in t_{dhkg} and t_{dhss} for ECDH-curve25519 and assuming a file size of $x = 1024$ bytes, we get a predicted performance of 694.31 HTTPS connections per second (within 1.3% of the measured performance of 703.35 HTTPS connections per second). If we replace ECDH-curve25519 with a curve that is twice as fast (as the FourQ curve is claimed to be, for example [5]), then we predict 766.1 HTTPS connections per second, a 10.4% improvement.With fast ECDH, ECDSA then becomes the dominant cost. Replacing ECDSA-nistp256 with a faster signature scheme would lead to further improvements; for example, with a signature scheme 1/8 the cost of ECDSA-nistp256 and key agreement 1/2 the cost of ECDH-curve25519, we estimate 1277.4 HTTPS connections per second.

Related work.

The work most closely related to ours is that of Zhao et al. [12], which gives a detailed decomposition of the cycle count of server operations in the OpenSSL library for a TLS connection with a variety of built-in ciphersuites (AES/DES/3DES/RC4, RSA, MD5/SHA-1). Zhao et al.'s work measures cycle counts for each operation on the server side that contributes to an SSL connection by instrumenting the SSL stack. Our work views TLS performance from the "external" perspective of connections realized per second.

2. BACKGROUND ON TLS

The Transport Layer Security (TLS) protocol [6, 7] provides security services to a variety of Internet applications, including the Hypertext Transport Protocol (HTTP) in the form of "HTTPS". TLS is composed of a *handshake* protocol (which negotiates parameters, performs server-to-client and optionally client-to-server authentication based on public key certificates, and establishes a shared session key), and a *record layer* protocol (which uses bulk encryption and authentication to provide confidentiality and integrity protection to application data). A modern ciphersuite in TLS 1.2 would use RSA with 2048-bit keys or ECDSA using the nistp256 curve for digital signatures, elliptic curve Diffie–Hellman over the nistp256 curve for key exchange (with forward secrecy), HMAC-SHA-256 for key derivation, and AES-128 in Galois counter mode (GCM) for authenticated encryption of application data. Figure 1 shows the message flow in the TLS protocol as well as the cryptographic operations performed by the server at each stage.

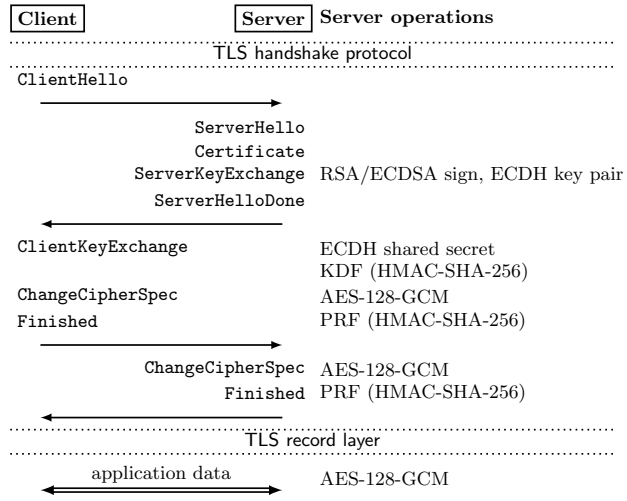

Figure 1: Message flow and server cryptographic operations for TLS 1.2 ciphersuite `RSA/ECDSA-ECDHE-AES128-GCM-SHA256` (server authentication only)

3. METHODOLOGY

Our methodology is based on that of Bos et al. [4]. Our experiment was carried out between two computers – a "client" and a "server". Throughout, OpenSSL v1.0.1g was used.

Standalone performance. The standalone performance measurements were carried out on the server computer using the `openssl speed` command, which uses a single core, and runs each performance test for 10 seconds. Results reported are the average of 5 runs.

TLS performance. For TLS performance measurements, we used the `http_load` tool (version 09jul2014)[1] to make many simultaneous HTTP requests from the client to the server; it used our modified OpenSSL for TLS connections.

[1]http://www.acme.com/software/http_load/

The server was running Apache `httpd` 2.4 with the prefork module for multi-threading. The client and server computers were connected over a local area network with sub-millisecond ping time and sufficient bandwidth to ensure that the server's 2 cores had at least 95% utilization during all tests. Session resumption was disabled.

We collected TLS performance measurements for several different web page sizes, namely HTTP payloads of 1 byte, 1 KiB = 1024 bytes, 10 KiB, and 100 KiB. Each test was run for 100 seconds; results reported are the average of 5 runs.

4. PREDICTIVE MODEL

Let n_{conn} denote the number of HTTPS connections per second per core, as in Table 2. Let t_{conn} represent the total runtime the server spends in processing an HTTPS connection. We model t_{conn} as the sum of the runtimes of various components:

$$t_{\text{conn}} = t_{\text{sig}} + t_{\text{dhkg}} + t_{\text{dhss}} + t_{\text{rest}} \tag{3}$$

where t_{sig} is the runtime of the signing algorithm, t_{dhkg} is the runtime of the Diffie–Hellman key pair generation, t_{dhss} is the runtime of the Diffie–Hellman shared secret generation, and t_{rest} is the runtime of the bulk encryption of the payload data as well as all other operations such as the key derivation function and non-cryptographic message processing.

Stage 1: Fixed key exchange, regression on file size.

In equation (3), t_{conn} and t_{rest} depend on the size of the payload, but the remaining terms $(t_{\text{sig}}, t_{\text{dhkg}}, t_{\text{dhss}})$ do not. As a result, we aim to derive a parameterized form of equation (3) based on the number of bytes x in the payload:

$$t_{\text{conn}}(x) = t_{\text{sig}} + t_{\text{dhkg}} + t_{\text{dhss}} + t_{\text{rest}}(x) \tag{4}$$

Since t_{rest} includes both the time for bulk encryption as well as other processing, we can further divide $t_{\text{rest}}(x)$ as the sum of a fixed component plus a component that scales with the number of bytes:

$$t_{\text{rest}}(x) = t_{\text{fix}} + c_{\text{bulk}} \cdot x \ . \tag{5}$$

Once we fix a ciphersuite, we can compute t_{sig}, t_{dhkg}, and t_{dhss} using the standalone performance measurements from Section 5. We estimate $t_{\text{conn}}(x)$ as $1/n_{\text{conn}}(x)$, and then we can compute $t_{\text{rest}}(1)$, $t_{\text{rest}}(1024)$, $t_{\text{rest}}(10240)$, and $t_{\text{rest}}(102400)$ by solving from the corresponding $t_{\text{conn}}(x)$ values. We then perform linear regression on $t_{\text{rest}}(x)$ to find coefficients t_{fix} and c_{bulk}.

Stage 2: Prediction over key exchange algorithms.

In the second stage of our predictive model, we fix all ciphersuite components except the key exchange algorithm, in order to predict the impact of improvements in key exchange performance. Fix t_{sig} as the runtime of the signing algorithm, and set $\overline{t_{\text{fix}}}$ and $\overline{c_{\text{bulk}}}$ to be the average of t_{fix} and c_{bulk}. For file size x bytes, the predicted number of connections per second is $\widetilde{n_{\text{conn}}} \approx 1/(t_{\text{sig}} + t_{\text{dhkg}} + t_{\text{dhss}} + \overline{t_{\text{fix}}} + \overline{c_{\text{bulk}}} \cdot x)$.

5. EXPERIMENTAL RESULTS

We focused on 4 Diffie–Hellman implementations from the eBACS project's SUPERCOP toolkit (version 20141124): curve2251 (version mpfq-x86_64), curve25519 (version mpfq-x86_64), ecfp256e (version v01-w8s1), and surf2113 (version mpfq-x64_64). We chose these specific primitives because they had portable C implementations that could be easily integrated into OpenSSL's build process and had a range of performance characteristics. We also included OpenSSL's own nistp256 implementation.[2]

Primitive	Key pair gen.	Shared secret gen.
ECDH-nistp256	1871.5	1803.9
ECDH-curve2251	3001.3	3182.9
ECDH-curve25519	7274.6	7579.3
ECDH-ecfp256e	28474.7	8296.6
ECDH-surf2113	4097.6	4180.9
RSA-2048	**signature** 468.3	
ECDSA-nistp256	**signature** 1674.6	

Table 1: Standalone cryptographic performance, in operations per second

We used the same symmetric algorithms in all TLS experiments: AES128 in GCM mode with SHA256 for hashing and key derivation as specified in TLS 1.2. We used two different signature schemes: ECDSA with the nistp256 curve and RSA with 2048-bit keys.

In our experiment, the "client" computer had an Intel Core i7 (4770) processor with 4 cores each running at 3.4 GHz. The "server" computer had an Intel Core 2 Duo (E8400) with 2 cores each at 3 GHz. (We used a more powerful client computer to ensure that the client would saturate the server.)

Standalone algorithm performance results.

Table 1 shows the results of the standalone performance results of the various cryptographic primitives.[3] Results shown are number of operations per second on the server computer. Note that key pair generation for nistp256 and ecfp256e is faster than shared secret generation due to fixed base-point scalar multiplication optimizations. For ecfp256e, this is approximately 3.39× faster than variable base-point scalar multiplication [10] [9, Table 6.4, 6.5].

HTTPS performance results.

Table 2 shows the results of the HTTPS performance of the various cryptographic primitives integrated into a TLS ciphersuite, with either ECDSA-nistp256 or RSA-2048

[2]In OpenSSL v1.0.1g, there are two nistp256 implementations: the default using wNAF point multiplication, and a 64-bit optimized constant-time implementation that must be selected at compile time; we used the default.

[3]Interestingly, the `openssl speed` benchmarking tool includes a performance optimization for ECDSA signatures that Apache's `mod_ssl` module does not use. `openssl speed` precomputes multiples of the base-point for optimization purposes using the `EC_KEY_precompute_mult` function. Apache's `mod_ssl` does not use this function. We observed that this precomputation can speed ECDSA signature generation by a factor of approximately 3.8× on our server machine ($t_{\text{ecdsa}} = 0.0001577$ with the optimization versus $t_{\text{ecdsa}} = 0.0005971$ without), suggesting that Apache's performance could be improved by adopting this optimization. Using equation (1) and fixing ECDH with nistp256 and the same bulk encryption parameters, our model predicts that this optimization would improve server performance to 549.6 connections per second, a 1.27× improvement. Since most TLS servers use the same curve for most ephemeral Diffie–Hellman connections, a further improvement may be possible by carefully switching ECDH key pair generation to use this optimization as well.

HTTP payload	nistp256		curve2251		curve25519		ecfp256e		surf2113	
	ECDSA	RSA	ECDSA	RSA	ECDSA	RSA	ECDSA	RSA	ECDSA	RSA
1 B	434.19	259.30	551.42	296.05	709.43	338.16	779.36	354.69	620.04	311.55
1 KiB	433.97	258.18	548.83	293.90	703.35	336.09	774.88	352.75	617.34	311.02
10 KiB	423.20	255.33	534.76	290.31	685.10	331.67	750.21	347.87	601.26	306.69
100 KiB	334.14	219.73	397.76	245.28	472.01	274.19	500.55	285.01	432.54	256.15

Table 2: HTTPS performance (HTTPS connections per server core per second); ECDSA or RSA signatures

DH primitive	ECDSA signatures		RSA signatures	
	t_{fix}	c_{bulk}	t_{fix}	c_{bulk}
nistp256	0.6123×10^{-3}	6.778×10^{-9}	0.6329×10^{-3}	6.769×10^{-9}
curve2251	0.5652×10^{-3}	6.869×10^{-9}	0.6003×10^{-3}	6.770×10^{-9}
curve25519	0.5377×10^{-3}	6.957×10^{-9}	0.5523×10^{-3}	6.724×10^{-9}
ecfp256e	0.5235×10^{-3}	7.032×10^{-9}	0.5270×10^{-3}	6.733×10^{-9}
surf2113	0.5262×10^{-3}	6.875×10^{-9}	0.5847×10^{-3}	6.831×10^{-9}

Table 3: Linear regression of $t_{\text{rest}}(x) = t_{\text{fix}} + c_{\text{bulk}} \cdot x$ for an x-byte HTTP payload

signatures. Results shown are number of HTTPS connections per server core per second.

5.1 Evaluation of predictive model

Consider for example the case of curve2251 with ECDSA signatures. Here, we have $t_{\text{sig}} = 0.0005971$, $t_{\text{dhkg}} = 0.0003332$ and $t_{\text{dhss}} = 0.0003142$. We then have

$$\begin{aligned} t_{\text{conn}}(1) = 0.0018135 &\implies t_{\text{rest}}(1) = 0.0005690 \\ t_{\text{conn}}(1024) = 0.0018221 &\implies t_{\text{rest}}(1024) = 0.0005775 \\ t_{\text{conn}}(10240) = 0.0018700 &\implies t_{\text{rest}}(10240) = 0.0006255 \\ t_{\text{conn}}(102400) = 0.0025141 &\implies t_{\text{rest}}(102400) = 0.0012696 \end{aligned}$$

Carrying out linear regression, we derive the equation

$$t_{\text{conn}}(x) = 0.5652 \times 10^{-3} + 6.869 \times 10^{-9} x$$

with a coefficient of determination $R^2 = 0.9996$.

Table 3 shows the results of this linear regression on all five DH primitives and 2 signature schemes. In all cases, the coefficient of determination of R^2 is at least 0.9989.

We can now see the justification for equation (2) in the Introduction: for ECDSA-nistp256 signatures (with $t_{\text{sig}} = 0.0005971$), and with $\overline{t_{\text{fix}}} = 0.0005662$ and $\overline{c_{\text{bulk}}} = 6.834 \times 10^{-9}$ as the average of the t_{fix} and c_{bulk} values in Table 3, when substituted into equation (1), we obtain equation (2).

Acknowledgments

This research is supported in part by Australian Research Council (ARC) Discovery Project grant DP130104304.

6. REFERENCES

[1] V. Bernat. SSL/TLS and perfect forward secrecy, 2011. http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html.

[2] D. J. Bernstein. Curve25519: New Diffie-Hellman speed records. In M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, Heidelberg, Apr. 2006.

[3] D. J. Bernstein and T. Lange. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to.

[4] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570. IEEE Computer Society Press, May 2015.

[5] C. Costello and P. Longa. FourQ: four-dimensional decompositions on a Q-curve over the Mersenne prime. Cryptology ePrint Archive, Report 2015/565, 2015. http://eprint.iacr.org/2015/565.

[6] T. Dierks and C. Allen. The Transport Layer Security (TLS) protocol version 1.0, January 1999. RFC 2246.

[7] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.2, August 2008. RFC 5246.

[8] V. Gupta, D. Stebila, S. Fung, S. Chang, N. Gura, and H. Eberle. Speeding up secure web transactions using elliptic curve cryptography. In *Proc. Network and Distributed System Security Symposium (NDSS) 2004*. Internet Society, 2004.

[9] H. Hışıl. *Elliptic Curves, Group Law, and Efficient Computation*. PhD thesis, Queensland University of Technology, April 2010.

[10] H. Hisil, K. K.-H. Wong, G. Carter, and E. Dawson. Jacobi quartic curves revisited. In C. Boyd and J. M. G. Nieto, editors, *ACISP 09*, volume 5594 of *LNCS*, pages 452–468. Springer, Heidelberg, July 2009.

[11] National Institute of Standards and Technology. Recommended elliptic curves for Federal government use, July 1999. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf.

[12] L. Zhao, R. Iyer, S. Makineni, and L. Bhuyan. Anatomy and performance of SSL processing. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) 2005*, pages 197–206. IEEE, 2005.