Performance analysis of elliptic curve cryptography for SSL

3 authors:

Vipul Gupta
Chikka Muniappa Reddy Institute of Technology
24 PUBLICATIONS   **1,799** CITATIONS

Sumit Gupta
Indian School of Business
28 PUBLICATIONS   **1,114** CITATIONS

Sheueling Chang Shantz
Mountain View College
34 PUBLICATIONS   **3,093** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project   Ai Health Website View project

Performance Analysis of Elliptic Curve Cryptography for SSL

Vipul Gupta
Sun Microsystems, Inc.
2600 Casey Avenue
Mountain View, CA 94303
vipul.gupta@sun.com

Sumit Gupta
Sun Microsystems, Inc.
2600 Casey Avenue
Mountain View, CA 94303
gupta.sumit@sun.com

Sheueling Chang
Sun Microsystems, Inc.
2600 Casey Avenue
Mountain View, CA 94303
sheueling.chang@sun.com

ABSTRACT

Elliptic Curve Cryptography (ECC) is emerging as an attractive public-key cryptosystem for mobile/wireless environments. Compared to traditional cryptosystems like RSA, ECC offers equivalent security with smaller key sizes, which results in faster computations, lower power consumption, as well as memory and bandwidth savings. This is especially useful for mobile devices which are typically limited in terms of their CPU, power and network connectivity.

However, the true impact of any public-key cryptosystem can only be evaluated in the context of a security protocol. This paper presents a first estimate of the performance improvements that can be expected in SSL (Secure Socket Layer), the dominant security protocol on the Web today, by adding ECC support.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques; E.3 [**Data Encryption**]: Public key cryptosystems; C.2.2 [**Computer Communication Networks**]: Network Protocols—*applications*

General Terms

Security, Performance

Keywords

Internet Security, Wireless, Secure Socket Layer (SSL), Elliptic Curve Cryptography (ECC)

1. INTRODUCTION

With the rapid deployment of applications like online banking, stock trading and corporate remote access, recent years have seen an explosive growth in the amount of sensitive data exchanged over the Internet. These days, an increasing number of Internet hosts are battery-powered, wireless, handheld devices with strict memory, CPU, latency and bandwidth constraints [1]. Given these trends, there is a clear need for efficient, scalable security mechanisms and protocols that operate well in both wired and wireless environments.

Most Internet security protocols (e.g. SSL [2], IPsec) employ a public-key cryptosystem to derive symmetric-keys and then use fast symmetric-key algorithms to ensure confidentiality, integrity and source authentication of bulk data. RSA is the most commonly used public-key cryptosystem today. The security of a system is only as good as that of its weakest component; for this reason, the work factor needed to break a symmetric key must match that needed to break the public-key cryptosystem used for key establishment. Due to expected advances in cryptanalysis and increases in computing power available to an adversary, both symmetric and public-key sizes must grow over time to offer acceptable security for a fixed protection life span, and Table 1 [3] shows this expected key-size growth for various symmetric and public-key cryptosystems.

Table 1: Computationally equivalent key sizes.

Symmetric	ECC	RSA/DH/DSA
80	163	1024
128	283	3072
192	409	7680
256	571	15360

As shown in Table 1, the Elliptic Curve Cryptosystem (ECC), offers the highest strength per bit of any known public-key cryptosystem today. ECC not only uses smaller keys for equivalent strength compared to traditional public-key cryptosystems like RSA, the key size disparity grows as security needs increase. This makes it especially attractive for constrained wireless devices because smaller keys result in power, bandwidth and computational savings. ECC was first proposed by Victor Miller [5] and independently by Neal Koblitz [6] in the mid 1980s and today it has evolved into a mature public-key cryptosystem. It was also recently endorsed by the U.S. government [4].

However, the true benefit and performance impact of any cryptosystem is closely tied to how it is used within a security protocol. In particular, it is imperative that expected performance improvements at the protocol level be carefully weighed against the usual costs associated with deploying any new technology.

This paper describes our work on integrating ECC into the Secure Socket Layer Protocol and a preliminary evaluation of the performance impact. We chose SSL because it is the most popular and trusted security protocol on the Web. In the form of HTTPS (HTTP secured using SSL), SSL is single handedly responsible for the widespread adoption of e-commerce and many emerging wireless devices too now have SSL capabilities. Our estimates are based on using the cryptographic execution times on an SSL client and server as a first approximation for the total processing time in an SSL transaction. Such an analysis can be generalized to estimate the performance of a cryptosystem within any other security protocol as well.

This paper is structured as follows. Section 2 provides an overview of ECC technology. Section 3 describes the SSL protocol and its usage of RSA and ECC public-key cryptosystems. Section 4 summarizes the public-key cryptographic operations needed to establish an SSL connection. An analytical performance comparison of RSA and ECC based SSL connections is presented in Section 5. Finally, we summarize our conclusions and discuss future work in Section 6.

2. ECC OVERVIEW

At the foundation of every public key cryptosystem is a hard mathematical problem that is computationally infeasible to solve. For instance, RSA and Diffie-Hellman rely on the hardness of integer factorization and the discrete logarithm problem respectively. Unlike these cryptosystems which operate over integer fields, the Elliptic Curve Cryptosystems (ECC) operates over points on an elliptic curve.

The fundamental mathematical operation in RSA and Diffie-Hellman is modular integer exponentiation. However, the core of elliptic curve arithmetic is an operation called *scalar point multiplication*, which computes $Q = kP$ (a point P multiplied k times resulting in another point Q on the curve). Scalar multiplication is performed through a combination of point-additions (which add two distinct points together) and point-doublings (which add two copies of a point together). For example, $11P$ can be expressed as $11P = (2 * ((2 * (2 * P)) + P)) + P$.

The security of ECC relies on the hardness of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP), which states that given P and $Q = kP$, it is hard to find k. While a brute-force approach is to compute all multiples of P until Q is found, k would be so large in a real cryptographic application that it would be infeasible to determine k in this way.

Besides the curve equation, an important elliptic curve parameter is the *base point*, G, which is fixed for each curve. In the Elliptic Curve Cryptosystem, the large random integer k is kept private and forms the secret key, while the result Q of multiplying the the private key k with the curve's base point G serves as the corresponding public key.

Not every elliptic curve offers strong security properties and for some curves the ECDLP may be solved efficiently. Since a a poor choice of the curve can compromise security, standards organizations like NIST and SECG have published a set of recommended curves [4] with well understood security properties. The use of these curves is also recommended as a means of facilitating interoperability between different implementations of a security protocol.

Elliptic Curve Diffie Hellman (ECDH) [7] and Elliptic Curve Digital Signature Algorithm (ECDSA) [8] are the Elliptic Curve counterparts of the Diffie-Hellman key exchange and Digital Signature Algorithm, respectively.

In ECDH key agreement, two communicating parties A and B agree to use the same curve parameters. They generate their private keys, k_A and k_B and corresponding public keys $Q_A = k_A.G$ and $Q_B = k_B.G$. The parties exchange their public keys. Finally each multiplies its private key and the other's public key to arrive at a common shared secret $k_A.Q_B = k_B.Q_A = k_A.K_B.G$.

While a description of ECDSA is not provided here, it similarly parallels DSA.

3. SSL OPERATION

3.1 Overview

Secure Sockets Layer [2] is the most widely deployed and used security protocol on the Internet today. The protocol has withstood years of scrutiny by the security community and is now trusted to secure virtually all sensitive web-based applications ranging from online banking and stock trading to e-commerce.

SSL offers encryption, source authentication and integrity protection for data exchanged over insecure, public networks. It operates above a reliable transport service like TCP and has the flexibility to accommodate different cryptographic algorithms for key agreement, encryption and hashing. However, the specification does recommend particular combinations of these algorithms, called *cipher suites*, which have well-understood security properties. For example, a cipher suite such as *RSA-RC4-SHA* would indicate RSA as the key exchange mechanism, RC4 for bulk encryption, and SHA for hashing.

The two main components of SSL are the Handshake protocol and the Record Layer protocol. The Handshake protocol allows an SSL client and server to negotiate a common cipher suite, authenticate each other[1], and establish a shared *master secret* using public-key cryptographic algorithms. The Record Layer derives symmetric-keys from the master secret and uses them with faster symmetric-key algorithms for bulk encryption and authentication of application data.

Public-key cryptographic operations are the most computationally expensive portion of SSL processing. SSL allows the re-use of a previously established master secret resulting in an abbreviated handshake that does not involve any public-key cryptography, and requires fewer and shorter messages. However, a client and server must perform a full handshake on their first interaction. Moreover, practical issues such as server load, limited session cache and naive load balancers can adversely impact the ability to use an abbreviated handshake. Therefore, speeding up the public-key operations in SSL still remains a very active area for research and development.

---

[1]Client authentication is optional. Only the server is typically authenticated at the SSL layer and client authentication is is achieved at the application layer, *e.g.* through the use of passwords sent over an SSL-protected channel. However, some deployment scenarios do require stronger client authentication through certificates.
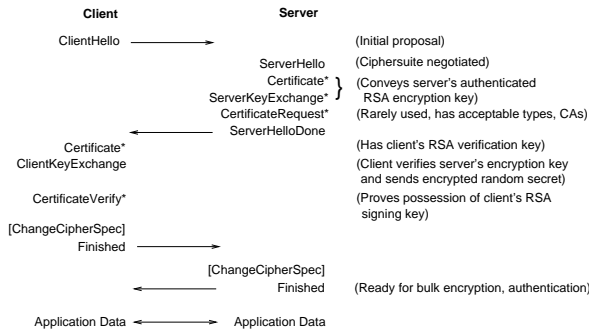
Figure 1: RSA-based SSL Handshake

Figure 2: ECC-based SSL Handshake

3.2 RSA-based Handshake

Today, the most commonly used public-key cryptosystem for master-key establishment is RSA. Figure 1 shows the operation of an RSA-based SSL handshake.

In this type of SSL handshake, the client and server first exchange random nonces (used for replay protection) and negotiate a cipher suite with *ClientHello* and *ServerHello* messages. The server then sends its signed RSA public-key either in the *ServerCertificate* message or the *ServerKeyExchange* message. To verify the server's RSA public key, the client performs an RSA public key operation. Then the client generates a 48-byte random number (the *premaster secret*), encrypts it with the server's public key (an RSA public-key operation), and sends it in the *ClientKeyExchange* message. The server uses its RSA private-key to decrypt the premaster secret. Both end-points then use the premaster secret to create a master secret which, along with previously exchanged nonces, is used to derive the cipher keys, initialization vectors and MAC (Message Authentication Code) keys for bulk encryption by the Record Layer.

The server can optionally request client authentication at the SSL layer by sending a *CertificateRequest* message listing acceptable certificate types and certificate authorities. In this case, besides performing the operations as described above, the client sends its RSA public-key in a *ClientCertificate* and proves possession of the corresponding private key by including a digital signature in the *CertificateVerify* message. Producing this signature requires the client to perform an RSA private-key operation.

3.3 ECC-based Handshake

A draft [3] describing the use of ECC with TLS [2] has been proposed at the IETF, and it forms the basis of our analysis in this paper.

Since ECC is a public-key cryptographic mechanism, only the handshake protocol is affected by incorporating ECC into SSL. Figure 2 shows the operation of an ECC-based SSL handshake, as specified in [3]. Through the first two messages (processed in the same way as for RSA) the client and server negotiate an ECC based cipher suite (for example, *ECDH-ECDSA-RC4-SHA*). However, the *ServerCertificate* message contains the server's Elliptic Curve Diffie-Hellman (ECDH) public key signed by a certificate authority using the Elliptic Curve Digital Signature Algorithm (ECDSA). After validating the ECDSA signature, the client conveys its ECDH public-key to the server in the *ClientKeyExchange*

_____

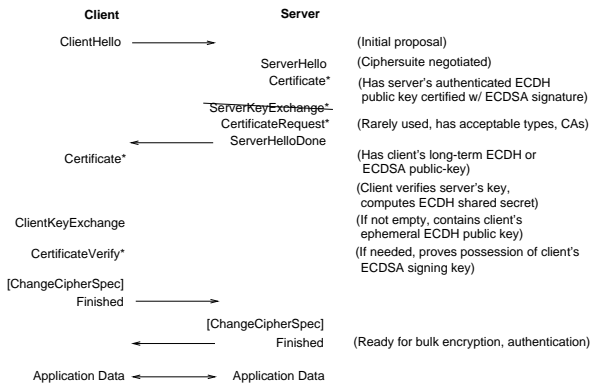[2]TLS[9] is another name for SSL version 3.1

message. Next, each entity uses its own ECDH private-key and the other's public-key to perform an ECDH operation and arrive at a shared premaster secret. The derivation of the master secret and symmetric keys is unchanged compared to RSA.

Client authentication is still optional but the actual message exchange depends on the type of authentication requested by the server and the kind of certificate a client possesses. If the client certificate has a long term ECDSA key, client authentication works similarly to the RSA case – the client sends its certificate in the *ClientCertificate*, a newly generated (ephemeral) ECDH public-key in the *ClientKeyExchange*, and signs the *CertificateVerify* message using its ECDSA private key.

However, if the client's certificate contains a long-term ECDH public-key, it is sent to the server in the *ClientCertificate* message. In this case, the *ClientKeyExchange* message is empty and the *CertificateVerify* message is not sent. The client's ability to generate a valid *Finished* message implicitly proves possession of the ECDH private-key.

The second form of client authentication is computationally cheaper for both sides but requires the client to have an ECDH certificate with the same curve parameters as those in the server certificate. If a client wishes to interact with multiple servers that use different parameters, it would need multiple certificates to use the second form of authentication.

4. PUBLIC-KEY CRYPTOGRAPHY IN SSL

Table 2 summarizes the various public-key cryptographic operations performed by a client and server in different modes of the of the SSL handshake.

4.1 Without Client Authentication

1. *RSA Handshake*
   The client performs two RSA public-key operations – one to verify the server's certificate and another to encrypt the premaster secret with the server's public key. The Server only performs one RSA private-key operation to decrypt the *ClientKeyExchange* message and recover the premaster secret.

2. *ECDH-ECDSA Handshake*
   The client performs an ECDSA verification to verify

Table 2: Cryptographic operations in an SSL Handshake.

	RSA	ECDH-ECDSA
Client	$\mathrm{RSA}_{verify} + \mathrm{RSA}_{encrypt}$	$\mathrm{ECDSA}_{verify} + \mathrm{ECDH}_{op}$
Server	$\mathrm{RSA}_{decrypt}$	ECDH_{op}

(a) Only the server is authenticated

	RSA	ECDH-ECDSA
Client	$\mathrm{RSA}_{verify} + \mathrm{RSA}_{encrypt} + \mathrm{RSA}_{sign}$	(i) $\mathrm{ECDSA}_{verify} + \mathrm{ECDH}_{op}$ *or*
		(ii) $\mathrm{ECDSA}_{verify} + \mathrm{ECDSA}_{sign} + \mathrm{ECDH}_{op}$
Server	$2 * \mathrm{RSA}_{verify} + \mathrm{RSA}_{decrypt}$	(i) $\mathrm{ECDSA}_{verify} + \mathrm{ECDH}_{op}$ *or*
		(ii) $2 * \mathrm{ECDSA}_{verify} + \mathrm{ECDH}_{op}$

(b) Both client and server are authenticated

the server's ECDSA certificate and then an ECDH operation using its private ECDH key and the server's public ECDH key to compute the shared premaster. All the server needs to do is perform an ECDH operation to arrive at the same secret.

4.2 With Client Authentication

1. *RSA Handshake*
   The client performs two RSA public-key operations (same as without client authentication) but additionally performs an RSA private-key operation to generate the *CertificateVerify* message The server performs two RSA public-key operations (one to verify the client's certificate and another to verify the client's signature in the *CertificateVerify* message) and a private-key operation to decrypt the premaster secret

2. *ECDH-ECDSA Handshake*
   (i) When the client uses an ECDH certificate, both sides perform an ECDSA verification operation on the other's certificate followed by an ECDH operation to compute the premaster secret.

   (ii) When the client uses an ECDSA certificate, the operations required on the two sides are asymmetric. The client performs an ECDSA verification of the server's certificate, an ECDH operation to compute the premaster secret and an ECDSA signature to generate the *CertificateVerify* message. The server performs an ECDH operation to compute the premaster secret and two ECDSA verifications – one to verify the client's certificate and another to verify the *CertificateVerify* message.

5. PERFORMANCE EVALUATION

5.1 Performance Metrics

While Table 2 identifies the individual cryptographic operations for the two sides in an SSL connection, the relevant performance metrics for the client and server may be quite different. An appropriate choice of such metrics is key to evaluate "perceived" performance accurately. In the case of a server that aggregates thousands of SSL requests, connection throughput (number of connections handled per second) is important. On the other hand, for a typical client which sequentially establishes one SSL connection at a time, connection latency is more important.

As a first approximation, we use the following two metrics for performance comparisons:

Handshake Crypto Latency This is the total time spent on performing cryptographic operations on the client and server. For instance, in the case of an RSA handshake without client authentication, this is the sum of times spent by the client doing two RSA public key operations and the server doing one RSA private key operation.

Server Crypto Throughput This is the rate at which the server can perform the cryptographic operations needed in the handshake. The client's performance is not a factor because the server can interleave multiple connections from different clients. For instance, in the case of an ECDH-ECDSA handshake without client authentication, the Server Crypto Throughput measured in connections per second is $1000/(\mathrm{ECDH}_{op}$ time in milliseconds).

Besides the public-key cryptographic operations, a full handshake also involves other delays due to message parsing, hashing and network latency. Therefore, the handshake crypto latency serves as a lower bound on the total handshake latency. Similarly, the Server Crypto Throughput figure only provides an upper bound on the actual SSL connection rate. Section 6 describes our plans regarding an empirical performance study involving actual measurements of SSL-level performance.

Network round trip delays can be a significant component of the handshake latency especially in the case of a slow wireless network such as CDPD. Latencies in the TCP/IP stack and web server implementation can further add to user perceived delay.

5.2 Measured algorithm-level performance

As part of adding ECC support to OpenSSL [10], the most widely used open source implementation of SSL, we have enhanced the OpenSSL cryptographic library to support ECDH and ECDSA. We have also added the ability to generate and process X.509 certificates containing ECC keys.

Table 3: Measured performance of public-key algorithms (in milliseconds).

	$RSA_{encrypt,verify}$	$RSA_{decrypt,sign}$	$ECDSA_{verify}$	$ECDSA_{sign}$	$ECDH_{op}$
Ultra-80	1.7	32.1	13.0	6.8	6.1
	6.1	205.5	18.1	9.2	8.7
Yopy	10.8	188.7	46.5	24.5	22.9
	39.1	1273.8	76.6	39.0	37.7

Table 3 shows the measured performance of primitive RSA, ECDH, and ECDSA operations using the OpenSSL0.9.6b speed program (enhanced to include ECC) on two platforms: (i) Yopy, a Linux PDA equipped with a 200MHz StrongARM processor, and (ii) an Ultra-80™, a Sun server equipped with a 450MHz Ultra-SPARC II processor. There are two rows of numbers for each platform. The top row is for 1024-bit RSA and 163-bit ECC whereas the bottom row is for 2048-bit RSA and 193-bit ECC.

The next section uses these measured numbers for the various plots.

5.3 Analytical SSL-level performance

To simulate various real world usage scenarios, we compare RSA and ECDH-ECDSA handshakes in each of the following cases:

Case I A Yopy to another Yopy (to model a Peer-to-Peer scenario of two small wireless handhelds communicating),

Case II A Yopy client talking to an Ultra 80 server (to simulate a wireless web scenario of a wireless handheld requesting a secure page from a webserver), and

Case III An Ultra 80 talking to another Ultra-80 (to model a usual desktop to webserver interaction).

The comparison in Figure 3(a) uses 1024-bit RSA and 163-bit ECC keys and does not involve client authentication. In terms of Server Crypto Throughput, ECC is more than five times better than RSA on the two platforms we consider. In terms of Handshake Crypto Latency, the comparison is more interesting. When both the SSL client and the server are on the same platform, we notice that ECC is nearly twice as fast as RSA. However, in the case of the Yopy client communicating with the Ultra-80 server, RSA beats out ECC. Compared to Case I, the SSL server is running on a faster platform in Case II and as shown in Table 3, this change speeds up the RSA decryption to a greater extent (from 188.7 ms to 32.1 ms) than it speeds up the ECDH operation (from 22.9 ms to 6.1 ms). The gain this provides to RSA puts it ahead of ECC in this situation.

Figure 3(b) compares the same scenarios as Figure 3(a) but with the addition of client-side authentication. As expected, client authentication using ECDH certificates performs better than client authentication using ECDSA certificates. More importantly, ECC beats out RSA on all the criteria we consider. While ECC's advantage over RSA in terms of Server Crypto Throughput is not as spectacular as before, it is still considerable especially when ECDH certificates are used.

We repeated these experiments using 2048-bit RSA keys and 193-bit ECC keys. We found ECC to perform better than RSA without any exceptions, even for Case II without

client authentication. Figure 4 shows the impact of using higher key sizes for the Yopy client communicating with the Ultra-80 server with and without client authentication. It is clear from the figure that the performance advantage of ECC over RSA increases at higher key sizes.

6. CONCLUSIONS AND FUTURE WORK

The above analysis suggests that the use of ECC cipher suites can offer significant performance benefits to SSL clients and servers especially as security needs increase. Already, there is significant momentum behind widespread adoption of the Advanced Encryption Standard (AES) which specifies the use of 128-bit, 192-bit and 256-bit symmetric keys. As indicated in Table 1, key sizes for public key cryptosystems used to establish AES keys will also need to increase from current levels. We believe this trend bodes well for the future of Elliptic Curve Cryptography and not just for wireless environments.

We have completed implementing ECC cipher suites in OpenSSL0.9.6b and currently migrating those changes to the latest OpenSSL version. Webservers and browsers that use OpenSSL for SSL processing can now use our variant to communicate securely via ECC cipher suites. We have validated this claim for Dillo and Lynx, two open-source browsers, and the Apache web server.

We are now in the process of setting up a testbed that would allow us to empirically study the impact of using ECC-based cipher suites. We intend to discuss those results in a more detailed follow-on publication.

7. ACKNOWLEDGMENTS

8. ADDITIONAL AUTHORS

Additional authors: Douglas Stebila (Department of Combinatorics and Optimization, University of Waterloo, email: `dstebila@uwaterloo.ca`).

9. REFERENCES

[1] IDC, "IDC envisions a time when majority of Internet access will be through wireless devices", see http://www.idc.com:8080/communications/press/pr/ CM041000pr.stm

[2] A. Frier, P. Karlton and P. Kocher, "The SSL3.0 Protocol Version 3.0", see http://home.netscape.com /eng/ssl3/.

[3] S. Blake-Wilson and T. Dierks, "ECC Cipher Suites for TLS", Internet draft <draft-ietf-tls-ecc-01.txt>, work in progress, Mar. 2001.
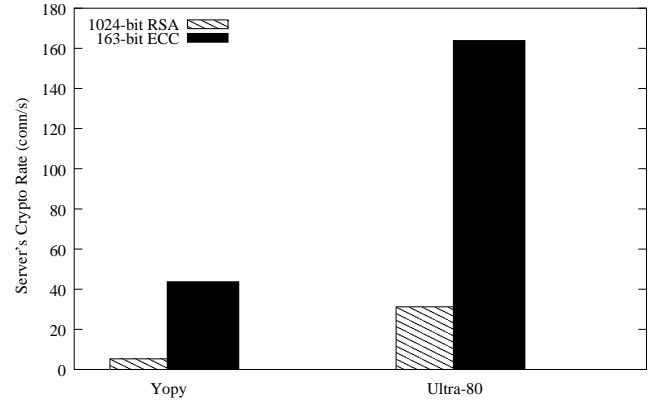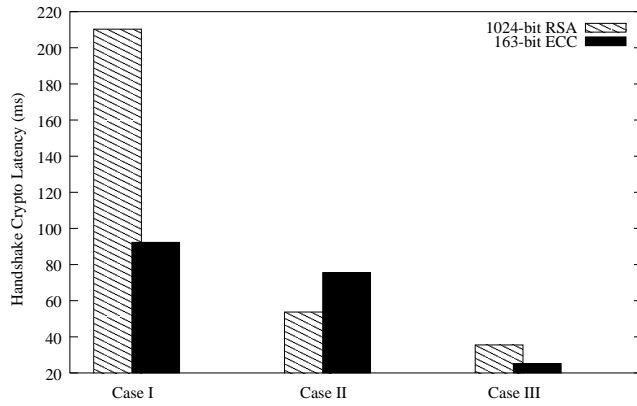
(a) Without Client Authentication
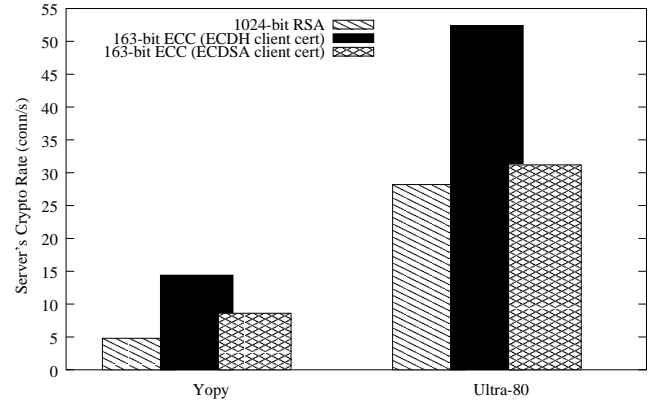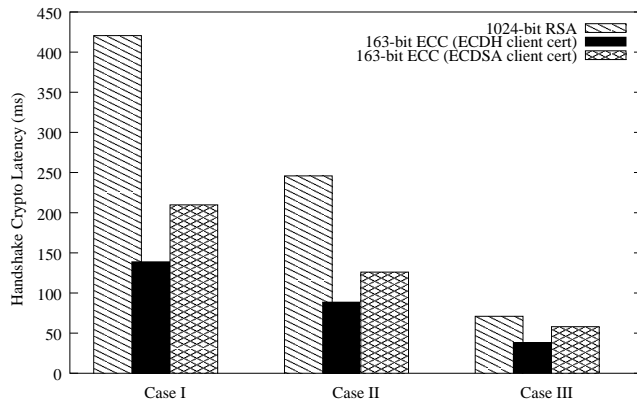
(b) With Client Authentication

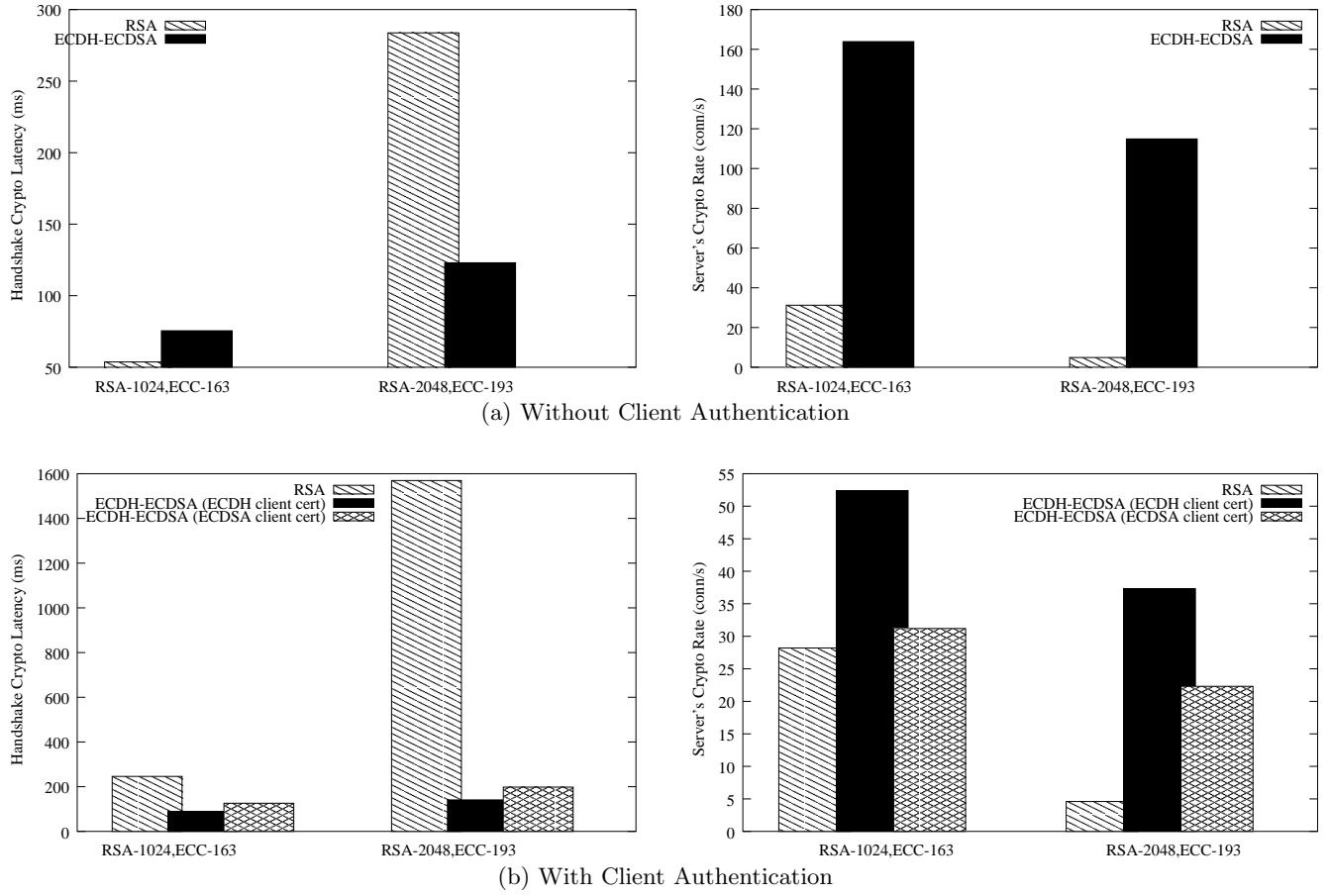Figure 3: RSA v/s ECC based handshake on different platforms.

(a) Without Client Authentication

(b) With Client Authentication

Figure 4: Impact of using higher key sizes in Case II (Case I and Case III show similar trends).

[4] NIST, "Recommended Elliptic Curves for Federal Government Use", July 1999, see http://csrc.nist.gov/csrc/fedstandards.html.

[5] V. Miller, "Uses of elliptic curves in cryptography", Lecture Notes in Computer Science 218: Advances in Cryptology - CRYPTO '85, pages 417-426, Springer-Verlag, Berlin, 1986.

[6] N. Koblitz, "Elliptic curve cryptosystems", Mathematics of Computation, 48:203-209, 1987.

[7] ANSI X9.62, "The Elliptic Curve Digital Signature Algorithm (ECDSA)", American Bankers Association, 1999.

[8] ANSI X9.63, "Elliptic Curve Key Agreement and Key Transport Protocols", American Bankers Association, 1999.

[9] T. Dierks and C. Allen, January 1999. "The TLS Protocol - Version 1.0.", IETF RFC 2246, see http://www.ietf.org/rfc/rfc2246.txt

[10] The OpenSSL Project, see http://www.openssl.org/.