Digital Signature Algorithm Implementations

Leah Chatkeonopadol 6951651
Smruthi Manjunath 3952009

***Abstract*—In this project, we implement DSA and ECDSA in Python to compare their performances. Specifically, we examine timings for signature generation and verification. For ECDSA, we also perform and analyze point multiplication in four different coordinate systems: affine, projective Jacobian, projective standard, and modified Jacobian.**

I. Introduction

The Digital Signature Algorithm (DSA) is a commonly used variant of El Gamal, one of the cryptographic signature schemes. In addition to the standard DSA, there exists an elliptic curve version, ECDSA. We implemented both DSA and ECDSA to explore the differences between these two algorithms in terms of timing and complexity [1].

The most notable difference between DSA and ECDSA is the fact that DSA requires larger parameters to provide the same level of security as ECDSA. For example, an ECDSA signature based on a 192-bit elliptic curve is as secure as a 1024-bit DSA signature. In turn, this means that ECDSA needs less memory for computation and hence is preferable for environments with limited memory, such as embedded systems. ECDSA also scales more efficiently than DSA in terms of security level.

The rest of this paper covers some background details and our implementation approach. Then we conclude with the results of our work.

II. Background

DSA includes a specific hash function, SHA-1, which takes the message to be sent and computes a 160-bit message digest. As shown in Figure 1, this message digest is used along with the private key to produce the signature. At the other end, the message is also hashed and used with the received signature and the public key to verify the signature [2].

DSA and ECDSA are extremely similar except that their parameters and main operation differ. In particular, DSA uses modular exponentiation, whereas ECDSA uses scalar point multiplication on an elliptic curve.

III. Implementation

Our implementations were done in Python running on a 2.4 GHz Intel Core 2 Duo. First we implemented DSA (parameters from [3]) using the binary method for modular exponentiation. Then we implemented ECDSA (parameters from [4]) using an NAF recoding of the exponent into balanced ternary and the corresponding modified binary method. For the elliptic curve computations, we used the approved NIST curve P-224 [5]. The extended Euclidean algorithm for finding modular inverses was used in both cases.

Authors are with the Department of Computer Science, University of California, Santa Barbara, CA 93106. E-mail: {chatkeon,smanjunath}@cs.ucsb.edu

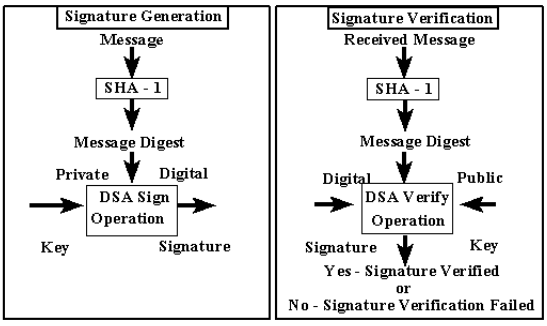

Fig. 1: Signature generation and verification

ECDSA was implemented in four different coordinate systems [6]. These are given in Table 1.

Coordinate System	Coordinates	Conversion to Affine
Affine	x, y	N/A
Projective Jacobian	X, Y, Z	$x = X/Z^2$, $y = Y/Z^3$
Projective Standard	X, Y, Z	$x = X/Z$, $y = Y/Z$
Modified Jacobian	X, Y, Z, T	$x = X/Z^2$, $y = Y/Z^3$, $(T = aZ^4)$

TABLE I: Coordinate Systems

IV. Results

The timings for signature generation and verification, averaged over 100 executions, are shown in Figure 2. These times do not include the time needed for message hashing. The time for signature verification is approximately twice as large as the time for signature generation. This is as expected, since DSA is known to be faster than RSA at signature generation, but slower at signature verification.

Affine coordinates clearly require more time than the other coordinate systems, as well as DSA, due to the significantly greater number of computations needed.

Figure 3 displays the average time per single operation for addition, subtraction, doubling, and inversion in each of the different coordinate systems. Naturally, addition and subtraction take roughly the same amount of time, while inversion is generally more expensive. Again, affine is much slower than the rest.

The number of operations these timings were averaged over are given in the table below (these are the numbers resulting

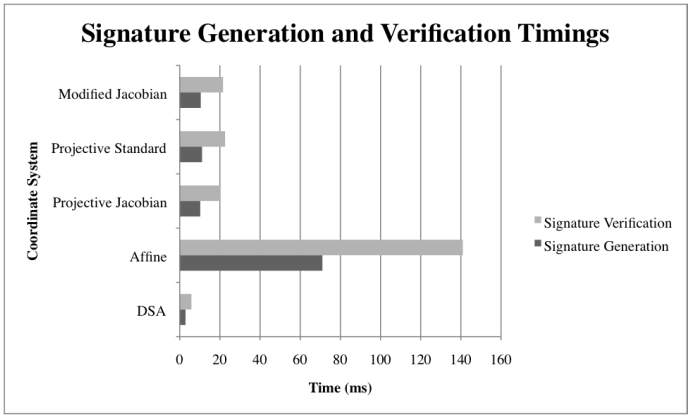

Fig. 2: Timings for signature generation and verification

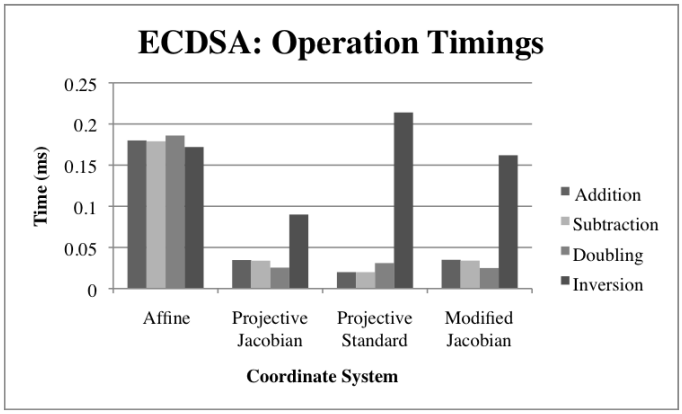

Fig. 3: Timings for various operations

from a single point multiplication for a specific value of k [4]). Addition, subtraction, and doubling were performed the same number of times in all four versions of ECDSA, but affine involved a much larger number of inversions.

Coord. System	Addition	Subtraction	Doubling	Inversion	Total
DSA	77	0	158	0	235
Affine	94	113	671	878	1756
Proj. Jacobian	94	113	671	6	884
Proj. Standard	94	113	671	3	881
Mod. Jacobian	94	113	671	3	881

TABLE II: Operation Counts for Different Coordinate Systems (and DSA)

The operation counts for DSA are from an equivalent modular exponentiation with the same value of k [4].

V. Conclusion

From our implementations, we conclude that projective Jacobian is more efficient than the other three coordinate systems, but projective Jacobian, projective standard, and modified Jacobian are all comparable. Affine is the least efficient.

We also note that implementation details have a large effect on performance. For example, switching from Fermat's little theorem to the extended Euclidean algorithm gave a significant performance boost. Similarly, altering the division by 2 in projective Jacobian coordinates made it faster than the other coordinate systems. Finally, we believe that Python might optimize modulation. This would explain our varied results.

References

[1] https://en.wikipedia.org/wiki/Digital_Signature_Algorithm
[2] http://www.itl.nist.gov/fipspubs/fip180-1.htm
[3] http://csrc.nist.gov/groups/ST/toolkit/documents/dss/Examples-1024bit.pdf
[4] http://tools.ietf.org/html/draft-pornin-deterministic-dsa-01#appendix-A.2.4
[5] http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf
[6] http://www.hyperelliptic.org/EFD/