Recommended Test Plan for Arithmetic and Cryptographic Operations in DTCP Signing Facility

Ç. K. Koç

Technical Report
Intel Corporation
October 1998

Abstract

This document outlines a set of methods for testing the correctness of the implementations of the arithmetic and cryptographic functions within the Digital Transmission Content Protection Signing Facility.

1 Testing Strategies

There are basically two types of tests used in the verification of arithmetic and cryptographic functions: Comparative tests and self-checking tests. In comparative tests, the target library (in this case, it is the SF) is compared against the reference library (in this case, it is the ICL). In self-checking tests, the programs in the target library are checked using basic identities of arithmetic operations, for example, we check whether $a \cdot 1 = a$; if it is, then our confidence in the correctness of the program performing the function '$\cdot$' increases. If the identity is not satisfied, then the program is faulty.

In comparative tests, the intermediate and final results obtained from the execution of the two programs are compared. If there is a complete match, the particular test is said to be successful and the code is verified for this particular test. Otherwise, the verification fails and the programs will need to be revised until the intermediate and the final results match. A disadvantage of this type of testing is that it relies on the accuracy (correctness) of the reference library.

The self-checking tests are implemented using the target library only. The task of verification in this type of test is mostly the equivalence of simple identity checks. The right and left sides of the identity are implemented using the library intended for verification. The two computed values are compared. If there is a match the test is successful, then the library is verified for this particular test. If the verification fails, the library will need to be revised. This type of testing appears to be simpler than the former method. However, the test in this type incorrectly verifies a library when the computed values of both sides match in an incorrect value.

2 Arithmetic and Elliptic Curve Operations

The arithmetic operations are applied on integers using simple addition, subtraction, multiplication, squaring, and the inverse. We first give the definitions of the arithmetic and the elliptic curve

operations and the corresponding code segments below. The following mathematical and notational assumptions are made:

- The inputs x and y are in the range $[0, p-1]$.
- The number of bits in the representation p is k. Thus, we have $2^{k-1} < p < 2^k$.
- The Montgomery constant is taken as $r = 2^k$.
- When it is used as an input, $r = 2^k \pmod{p}$ or $r = 2^k - p$.
- The elliptic curve is built upon the field $GF(p)$.
- The base point of the elliptic curve is $\mathcal{P}$.
- The order of the base point is equal to n.
- The points on the elliptic curve are denoted by P, Q, and R.
- The point at infinity is denoted by O and has a representation.
- In the elliptic curve point multiplication, the number e is an integer less than the order n.

Arithmetic Operations		
Description	Mathematical Notation	Code Segment
Integer Addition	$z = x + y$	`z := IntAdd(x,y)`
Integer Subtraction	$z = x - y$	`z := IntSub(x,y)`
Integer Multiplication	$z = x \cdot y$	`z := IntMul(x,y)`
Integer Squaring	$z = x \cdot x$	`z := IntSqu(x)`
Integer Division (Quotient)	$z = x \div y$	`z := IntDiv(x,y)`
Modular Addition	$z = x + y \pmod{p}$	`z := ModAdd(x,y,p)`
Modular Subtraction	$z = x - y \pmod{p}$	`z := ModSub(x,y,p)`
Modular Multiplication	$z = x \cdot y \pmod{p}$	`z := ModMul(x,y,p)`
Modular Squaring	$z = x \cdot x \pmod{p}$	`z := ModSqu(x,p)`
Modular Inversion	$z = x^{-1} \pmod{p}$	`z := ModInv(x,p)`
Montgomery Addition[1]	$z = x + y \pmod{p}$	`z := MonAdd(x,y,p)`
Montgomery Subtraction[1]	$z = x - y \pmod{p}$	`z := MonSub(x,y,p)`
Montgomery Multiplication	$z = x \cdot y \cdot r^{-1} \pmod{p}$	`z := MonMul(x,y,p)`
Montgomery Squaring	$z = x \cdot x \cdot r^{-1} \pmod{p}$	`z := MonSqu(x,p)`
Montgomery Inversion	$z = x^{-1} \cdot r \pmod{p}$	`z := MonInv(x,p)`
Elliptic Curve Operations		
Description	Mathematical Notation	Code Segment
EC Point Addition	$R = P + Q$	`R := ECAdd(P,Q)`
EC Point Negation	$R = -P$	`R := ECNeg(P)`
EC Point Subtraction	$R = P - Q$	`R := ECSub(P,Q)`
EC Point Doubling	$R = P + P = 2P$	`R := ECTwo(P)`
EC Point Multiplication	$R = e \cdot P$	`R := ECMul(e,P)`

[1]Montgomery Addition and Montgomery Subtraction operations are equivalent to Modular Addition and Modular Subtraction operations, respectively. These will be subsequently ignored.

In addition to the above basic arithmetic operations, we assume that two programs `ModExp(x,e,p)` and `MonExp(x,e,p)` are also available. They compute the same result $x^e \pmod p$ for a given x and e, where x is in the range $(0, p)$, and e is a k-bit positive integer $e = (e_{k-1}e_{k-2}\cdots e_0)$. They can be implemented as follows:

```
ModExp(x,e,p):
y := 1
for i=k-1 downto 0
 y := ModSqu(y,p)
 if e_i = 1 then y := ModMul(x,y,p)
return y
```

```
MonExp(x,e,p):
y := ModMul(1,r,p)
x := ModMul(x,r,p)
for i=k-1 downto 0
  y := MonSqu(y,p)
  if e_i = 1 then y := MonMul(x,y,p)
y := MonMul(y,1,p)
return y
```

3 Self-Checking Type of Tests

The following are simple arithmetic operations. They are performed for several randomly selected input values x, y, and z. These values are in the range $[0, p-1]$.

Integer Operation	Modular Operation	Montgomery Operation	Comment
`IntAdd(x,0) =? x`	`ModAdd(x,0,p) =? x`	-	$x + 0 = x$
`IntSub(x,0) =? x`	`ModSub(x,0,p) =? x`	-	$x - 0 = x$
`IntSub(x,x) =? 0`	`ModSub(x,x,p) =? 0`	-	$x - x = 0$

`t1 := IntAdd(y,z)`
`t2 := IntAdd(x,t1)`
`t3 := IntAdd(x,y)`
`t4 := IntAdd(t3,z)`
`t4 =? t2`
	`t1 := ModAdd(y,z,p)`
	`t2 := ModAdd(x,t1,p)`
	`t3 := ModAdd(x,y,p)`
	`t4 := ModAdd(t3,z,p)`
	`t4 =? t2`
		-
		-
		-
		-
		-
			$x + (y + z) =$
			$(x + y) + z$

`t1 := IntSub(x,y)`
`t2 := IntSub(x,t1)`
`t2 =? y`
	`t1 := ModSub(x,y,p)`
	`t2 := ModSub(x,t1,p)`
	`t2 =? y`
		-
		-
		-
			$x - (x - y) = y$

`IntMul(x,1) =? x`	`ModMul(x,1,p) =? x`	`MonMul(x,r,p) =? x`	$x \cdot r \cdot r^{-1} = x$
`IntMul(x,0) =? 0`	`ModMul(x,0,p) =? 0`	`MonMul(x,0,p) =? 0`	$x \cdot 0 = 0$

`t1 := IntMul(x,y)`
`t2 := IntMul(y,x)`
`t1 =? t2`
	`t1 := ModMul(x,y,p)`
	`t2 := ModMul(y,x,p)`
	`t1 =? t2`
		`t1 := MonMul(x,y,p)`
		`t2 := MonMul(y,x,p)`
		`t1 =? t2`
			$x \cdot y = y \cdot x$

`t1 := IntMul(x,x)`
`t2 := IntSqu(x)`
`t1 =? t2`
	`t1 := ModMul(x,x,p)`
	`t2 := ModSqu(x,p)`
	`t1 =? t2`
		`t1 := MonMul(x,x,p)`
		`t2 := MonSqu(x,p)`
		`t1 =? t2`
			$x \cdot x = x^2$

- -	`t1 := ModInv(x,p)` `ModMul(t1,x,p) =? 1`	`t1 := MonInv(x,p)` `MonMul(t1,x,p) =? 1`	$x^{-1} \cdot x = 1$ $x^{-1} \cdot r \cdot x \cdot r^{-1} = 1$
- -	`t1 := ModInv(x,p)` `ModInv(t1,p) =? x`	`t1 := MonInv(x,p)` `MonInv(t1,p) =? x`	$(x^{-1})^{-1} = x$ $(x^{-1} \cdot r)^{-1} \cdot r = x$

The following advanced arithmetic operations are performed for several randomly selected input values x, e_1, and e_2, where $x \in [0, p-1]$ and e_1 and e_2 are positive integers in the range $[1, p-2]$.

Modular Operation	Montgomery Operation	Comment
`ModExp(x,p-1,p) =? 1`	`MonExp(x,p-1,p) =? 1`	$x^{p-1} = 1 \pmod{p}$
`t1 := ModExp(x,p-2,p)` `ModInv(x,p) =? t1`	`t1 := MonExp(x,p-2,p)` `ModInv(x,p) =? t1`	$x^{p-2} = x^{-1} \pmod{p}$
`t1 := ModExp(x,p-2,p)` `ModMul(t1,x,p) =? 1`	`t1 := MonExp(x,p-2,p)` `ModMul(t1,x,p) =? 1`	$x^{p-2} \cdot x = 1 \pmod{p}$

`e2 := ModAdd(e1,0,p-1)`
`t1 := ModExp(x,e1,p)`
`ModExp(x,e2,p) =? t1`
	`e2 := ModAdd(e1,0,p-1)`
	`t1 := MonExp(x,e1,p)`
	`MonExp(x,e2,p) =? t1`
		$e_1 = e_2 \pmod{p-1}$
		$x^{e_1} = x^{e_2} \pmod{p}$

`e1 := IntDiv(p-1,2)`
`t1 := ModExp(x,e1,p)`
`t1 =? 1 or -1`
	`e1 := IntDiv(p-1,2)`
	`t1 := MonExp(x,e1,p)`
	`t1 =? 1 or -1`
		$x^{(p-1)/2} = \pm 1 \pmod{p}$

The following are simple elliptic curve operations. They are performed for several randomly selected points on the curve P, Q, and R. The point O is the unity point. A random point P on the elliptic curve can be generated by first randomly generating a positive integer e, and then computing $P = e\mathcal{P}$, where $\mathcal{P}$ is the base point (generator) of the elliptic curve.

EC Operation	Comment
`T1 := ECAdd(P,Q)` `T1 =? ECAdd(Q,P)`	$P + Q = Q + P$

`T1 := ECAdd(Q,R)`
`T2 := ECAdd(P,T1)`
`T3 := ECAdd(P,Q)`
`ECAdd(T3,R) =? T2`
	$P + (Q + R) = (P + Q) + R$

`T1 := ECNeg(P)` `ECAdd(T1,P) =? O`	$P + (-P) = O$
`ECAdd(O,O) =? O`	$O + O = O$
`ECSub(O,O) =? O`	$O - O = O$
`ECSub(P,P) =? O`	$P - P = O$
`ECSub(P,O) =? P`	$P - O = P$
`T1 := ECSub(O,P)` `ECNeg(P) =? T1`	$O - P = -P$
`T1 := ECTwo(P)` `ECSub(T1,P) =? P`	$2P - P = P$

`T1 := ECMul(e1,P)`
`T2 := ECMul(e2,P)`
`T3 := ECAdd(T1,T2)`
`e3 := ModAdd(e1,e2,n)`
`ECMul(e3,P) =? T3`
	$e_1P + e_2P = (e_1 + e_2)P$
	
	$e_3 = e_1 + e_2 \pmod{n}$

`T1 := ECMul(e1,P)`
`T2 := ECMul(e2,T1)`
`e3 := ModMul(e1,e2,n)`
`ECMul(e3,P) =? T2`
	$e_2(e_1P) = (e_2e_1)P$
	
	$e_3 = e_1 \cdot e_2 \pmod{n}$

`ECMul(n,Base) =? O`	$n \cdot \mathcal{P} = O$
`T1 := ECMul(n-1,Base)` `ECNeg(Base) =? T1`	$(n-1) \cdot \mathcal{P} = -\mathcal{P}$

4 Comparative Type of Tests

The arithmetic operations are performed using both the SF and ICL and the temporary results t_i for $i = 1, 2, \ldots, 40$ are compared.
The procedure:

- Randomly generate x_1, x_2, e_1, and e_2.
- Compute the results using SF and ICL, and compare.

t1 := IntAdd(x1,x2)	t21 := ModAdd(x1,x2,p)
t2 := IntAdd(x2,x1)	t22 := ModAdd(x2,x1,p)
t3 := IntSub(x1,x2)	t23 := ModSub(x1,x2,p)
t4 := IntSub(x2,x1)	t24 := ModSub(x2,x1,p)
t5 := IntMul(x1,x2)	t25 := ModMul(x1,x2,p)
t6 := IntMul(x2,x1)	t26 := ModMul(x2,x1,p)
t7 := IntSqu(x1)	t27 := ModSqu(x1,p)
t8 := IntSqu(x2)	t28 := ModSqu(x2,p)
t9 := IntDiv(x1,x2)	t29 := ModInv(x1,p)
t10 := IntDiv(x2,x1)	t30 := ModInv(x2,p)
t11 := MonAdd(x1,x2,p)	t31 := ModExp(x1,e1,p)
t12 := MonAdd(x2,x1,p)	t32 := ModExp(x1,e2,p)
t13 := MonSub(x1,x2,p)	t33 := ModExp(x2,e1,p)
t14 := MonSub(x2,x1,p)	t34 := ModExp(x2,e2,p)
t15 := MonMul(x1,x2,p)	t35 := ModExp(x1,p,p)
t16 := MonMul(x2,x1,p)	t36 := ModExp(x2,p,p)
t17 := MonSqu(x1,p)	t37 := MonExp(x1,p-1,p)
t18 := MonSqu(x2,p)	t38 := MonExp(x2,p-1,p)
t19 := MonInv(x1,p)	t39 := MonExp(x1,p-2,p)
t20 := MonInv(x2,p)	t40 := MonExp(x2,p-2,p)

Furthermore, the EC extension of the ICL and the SF implementations of the elliptic curve operations are compared below. Here, P_1 and P_2 are random points on the curve, and e_1 and e_2 are random integers. The results Q_i for $i = 1, 2, \ldots, 18$ are compared.
The procedure:

- Randomly generate P_1, P_2, e_1, and e_2.
- Compute the results using SF and ICL, and compare.

Q1 := ECAdd(P1,P2)	Q10 := ECAdd(P2,P1)
Q2 := ECSub(P1,P2)	Q11 := ECSub(P2,P1)
Q3 := ECNeg(P1)	Q12 := ECNeg(P2)
Q4 := ECTwo(P1)	Q13 := ECTwo(P2)
Q5 := ECMul(e1,P1)	Q14 := ECMul(e1,P2)
Q6 := ECMul(e2,P1)	Q15 := ECMul(e2,P2)
Q7 := ECMul(n,P1)	Q16 := ECMul(n,P2)
Q8 := ECMul(n-1,P1)	Q17 := ECMul(n-1,P2)
Q9 := ECMul(n-2,P1)	Q18 := ECMul(n-2,P2)

5 Comparing Against Maple or Mathematica

I propose to write a set of Maple (or Mathematica) routines for the standard arithmetic, Montgomery arithmetic, and elliptic curve operations (i.e., all operations in § 2 including `ModExp` and `MonExp`) in order to test the SF functions. The results obtained from Maple using randomly selected input values are compared to those obtained from the SF. This test will increase our confidence in the SF functions immensely since the arithmetic operations in Maple are used by many people, and their reliability is very high.

6 Testing SHA, EC Sign and Verify Operations

The validation of EC Sign and Verif operation first requires the validation of Secure Hash Standard (SHS) used in this project, since both the ECDSA generation and verification algorithms use the hash function before they actually sign or verify the messages. Then, the signature generation and verification algorithms on these hashed messages should be tested for correctness.

6.1 SHA Hash Function Tests

A hash function H is a transformation that takes a variable-size input M and returns a fixed-size string, which is called the hash value h that is $h = H(M)$. The basic requirements for a cryptographic hash function are:

- The input M can be of any length.
- The output h has a fixed length (160 bits).
- $H(M)$ is relatively easy to compute for any given M.
- $H(M)$ is one-way.
- $H(M)$ is collision-free.

The Signing Facility software uses SHA with 160-bit output value. By the birthday paradox, $2^{160/2} = 2^{80}$ possible input values should be tried to find a collision; this is computationally infeasible. The following tests are being proposed:

- The first test would be giving the test vectors presented in FIPS 180-1 Appendix A and Appendix B to the SF version of SHA as inputs. If the outputs match the ones given by FIPS 180-1 then the SHA implementation of SF passes the first test.
- The SHA is used to compute a message digest for a message or data file that is provided as input. The SHA sequentially processes blocks of 512 bits when computing the message digest. The purpose of message padding is to make the total length of a padded message a multiple of 512. In some cases, the whole data may not be available as a 512-bit block. Therefore, testing the implementation of SHA for varying input lengths is important. For example, processing the whole data at once and inputting the same data with different lengths of chunks to the algorithm should give the same message digest.

  An implementation of the SHA must be able to correctly generate message digests for messages of arbitrary length. Pseudorandomly generated n messages with lengths from 0 to 1024

bits. An implementation will probably hash byte-oriented data, thus, messages of length $8, 16, 24, ..., 1024$ bits may need to be supplied. We provide these data as inputs to both the ICL and SF implementations and then compare the final (as well as the temporary) results. If all messages compare correctly, then this test is passed.

- A similar test to the given test above could be operated on selected long messages. A list of n messages is supplied to the SF implementation of SHA, which then generates message digest for each. These are then compared to the ones generated by the ICL code.
- Another test is the following: After generating a message M with randomly generated length L, the SF implementation SHA calculates SHA$[M]$. Then the input value M is changed in a single (randomly selected) location to obtain M'. The hashed value SHA$[M']$ should not match the earlier computed value SHA$[M]$.

6.2 ECDSA Signature Generation and Verification Tests

There are six areas of the ECDSA for which SF can actually provide conformance testing:

- Primality testing.
- Generation of the parameters p, a, b, and n.
- Generation of public and private key pairs (X^1, X^{-1}).
- Signature generation.
- Signature verification.

We assume that the parameters p, a, b, and n are already available. The method for the generation of public and private key pair is given in IEEE P1363 Draft. The methods to test accuracy of the SF ECDSA implementation are given as follows.

The first method generates a message of random length L, where $0 < L < 256$ in bytes, and computes the signature of this message by using the SF ECDSA implementation. The ICL code with ECC extension concurrently calculates the signature of the same message. The signatures should match. This first step should be done with several new randomly generated messages. If both implementations yield the same signatures for all the messages, then it can be concluded that the signature generation algorithm of SF is error-free. The first part of the test is shown below.

- Generate random length L with $0 < L < 256$ in bytes.
- Generate buffer M of length L consisting of random values.
- Change the state of a randomly selected bit in M to make M'.
- Compute $S1 = \text{Sign}(X^{-1})[M]$ and $S2 = \text{Sign}(X^{-1})[M']$.
- Compare $S1$ and $S2$ with signatures calculated using the ICL code with ECC extensions.

Note that:

- $S1$ computed by SF and $S1$ computed by ICL should match.

- $S2$ computed by SF and $S2$ computed by ICL should match.
- $S1$ should not match $S2$ in either case.

The second test is for signature verification. From now on, the validation system does not need to use the ICL code. Previously generated signature $S1$ is supplied to the SF ECDSA verification algorithm to see that verify is successful. The signature $S1$ should be verified. If not, then it should be concluded that the verification algorithm fails. This can be shown in following notation:

- Compute Verify$(X)[S1, M]$. The result is True or False.
- Check to see that result is True. If not, the implementation is faulty.

The third test consists of three subtests. The first subtest is to keep the signature S and the public key X in their original form, however, flip a randomly selected bit in M to obtain M'. The validation system then runs the verification algorithm to see if the signature is verified. The second subtest is to keep M and X in their original form, however, flip a randomly selected bit of S resulting in S', and check the signature verification. Similarly, in the third subtest we keep the signature and the message in their original form, and flip a randomly selected bit of X resulting in X'. The verification of the signature should fail in the three cases. The third test is summarized below:

- Compute Verify$(X)[S, M']$. The result should be False.
- Compute Verify$(X)[S', M]$. The result should be False.
- Compute Verify$(X')[S, M]$. The result should be False.

7 Tests for Randomness

The generation of random numbers is critical to cryptographic systems. For example, the symmetric ciphers require randomly selected encryption keys and elliptic curve cryptosystems need randomly generated seeds for parameter generation. At a higher level, many cryptographic protocols use random challenges in the authentication process.

A good random number generator (RNG) which can be used for these purposes usually consists of two main components: a truly random seed and a pseudo-random number generator (PRNG). The random seed is desired to be generated by a truly random source and is given to the input of the PRNG as an initial state. Using this initial value, the PRNG starts generating random numbers.

There are several commercially available RNG integrated circuits which utilize different natural sources of randomness such as the noise in an electrical circuit. However, because of either sampling patterns employed or some inherent problems of the integrated circuits used, these RNGs should be tested.

The PRNGs generate the same stream of numbers when it starts with the same seed. The quality of a PRNG is determined by the high degree of unpredictability. It also has to withstand strong cryptanalytic techniques.

Due to their different characteristics, the randomness tests for the RNG and PRNG will be examined in two separate categories. Since most of the tests applied to the PRNG can also be applied to the RNG, the PRNG tests will be given first.

7.1 Tests for Pseudo-Random Number Generators

The tests in this category are statistic tests which compare the appearance of the output stream of a PRNG against that of truly random numbers. If the PR stream is hard to distinguish from truly random numbers, then it is accepted. Although there are many statistical tests that have been proposed so far, six of them are sufficient.

- **Frequency Test:** The aim of the frequency test is to determine how the proportion of ones in the sample stream of length n bits fits into the hypothesised distribution where the proportion of ones is 0.5.
- **Binary Derivative Test:** The binary derivative of a given stream is defined as a new stream obtained by the modulo 2 addition of successive bits in the stream. Then the frequency test is applied to the binary derivatives of the original stream.
- **Change Point Test:** At each bit position t in the stream, the proportion of ones to that point is compared to the proportion of ones in the remaining stream. The difference or change in these proportions is compared for all positions in the bit stream. The bit where the maximum change occurs is called the change point. The test determines whether this change is significant for a binomial distribution where the proportion of ones in the stream is expected to be 0.5.
- **Poker Test:** The poker test partitions the stream into F blocks of m bits. The aim of the poker test is to show that there is an equal number of each of the 2^m possible blocks.
- **Runs Test:** The runs test counts the number of runs of ones (blocks) and runs of zeros (gap) for each possible run length. For random data, there should be an equal number of blocks and gaps.
- **Entropy (Universal) Test:** The average entropy or information of bits or blocks in the stream must be as high as possible. This test computes a value for a stream which can be considered a measure of the entropy of the stream.

These tests should be applied to as many output streams as possible.

7.2 Tests for Random Number Generator Chips

In addition to the tests described above some other tests should be applied to the output of the RNG chips.

- **Diversity Test:** If 32-bit samples are taken from the RNG, it is highly probable to find collisions (values sampled twice) after If the collisions start to happen very soon, then the randomness that the chip provides is not sufficient for cryptographic purposes.
- **Population Test:** The population is the universe of all the objects from which a sample could be drawn for an experiment. If a representative random sample is chosen, the results of the experiment should be generalizable to the population from which the sample was drawn, but not necessarily to a larger population. For example, the results obtained for the 32-bit samples might not be true for the 64-bit samples. Therefore, the diversity test should be applied to the samples of different lenghts.