

SHA-3 Submission

Algorithm Name: ECOH (Elliptic Curve Only Hash)
Principal Submitter: Daniel R. L. Brown
Email: dbrown@certicom.com
Telephone: 905-826-8353
Fax: 905-507-4230
Organization: Certicom Corporation
Street Address: 5520 Explorer Drive
City: Mississauga
Province: Ontario
Country: Canada
Postal Code: L4W 5L1
Auxiliary Submitters: Adrian Antipa, Matt Campagna, Rene Struik
Inventors: Daniel R. L. Brown, Matt Campagna, Rene Struik
Owner: Certicom Corporation

Signature:

ECOH: the Elliptic Curve Only Hash

Daniel R. L. Brown

October 30, 2008

Contents

1	Introduction	1
1.1	Efficiency	1
1.2	Security	1
1.3	A More Cautious Variant	2
2	Glossary	3
3	Specification	6
3.1	ECOH-224	7
3.2	ECOH-256	8
3.3	ECOH-384	9
3.4	ECOH-512	10
4	Efficiency Analysis	11
4.1	Reference Implementation	11
4.2	Optimized Implementation	11
4.3	Assembly Optimization	11
4.4	Simultaneous Inversion	12
4.5	AVX instructions	12
4.6	Parallelization	12
5	Generalization of ECOH	14
5.1	Generalized ECOH Parameters	14
5.2	Generalized ECOH Operations	16
5.2.1	Input	17
5.2.2	Padding	17
5.2.3	Parsing	17
5.2.4	Indexing	17
5.2.5	Tailing	18
5.2.6	Searching	18
5.2.7	Recovering	19
5.2.8	Summing	19
5.2.9	Converting	19

5.2.10	Multiplying	20
5.2.11	Shifting	20
5.2.12	Converting	20
5.2.13	Selecting	20
5.2.14	Output	20
5.3	Suggested alternative values of parameters	20
5.3.1	Curves for deterministic runtime	20
5.3.2	Inverted ECOH	21
5.4	Incremental mode of operation	21
6	Security	23
6.1	Collision Resistance	23
6.1.1	Birthday Attack	23
6.1.2	Semaev Summation Polynomial Attack	24
6.1.3	Shifted collision attacks	26
6.1.4	Truncation attack	26
6.2	Second Preimage Resistance	26
6.2.1	Exhaustive search for second preimages	26
6.2.2	Wagner's Generalized Birthday Attack	27
6.2.3	Semaev summation polynomial second preimages	27
6.3	Preimage Resistance	27
6.3.1	Exhaustive preimage search	28
6.3.2	Inverting ECOH	28
6.4	Relative Preimage Resistance	28
6.4.1	Exhaustive search for preimages	29
6.4.2	Single block inversion	29
6.5	Pseudorandomness	30
6.6	Side Channel Resistance	30
6.7	Denial of Service Resistance	30
7	Applications	32
7.1	Digital Signatures	32
7.2	Message Authentication	32
7.3	Pseudorandom Number Generation	32
7.4	Key Derivation	32
7.5	Randomized Hashing	33
A	Heuristic Security Arguments	35
A.1	Collision Resistance	35
A.2	Second Preimage Resistance	35
A.3	Preimage Resistance	35
A.4	Pseudorandomness	37

B	Examples	38
B.1	Some Examples	38
B.2	Examples of Intermediate Values	38
B.2.1	ECOH224 Example (one-block Message)	38
B.2.2	ECOH224 Example (two-block Message)	41
B.2.3	ECOH256 Example (one-block Message)	44
B.2.4	ECOH256 Example (two-block Message)	46
B.2.5	ECOH384 Example (one-block Message)	49
B.2.6	ECOH384 Example (two-block Message)	51
B.2.7	ECOH512 Example (one-block Message)	54
B.2.8	ECOH512 Example (two-block Message)	57

List of Tables

1	Common variables	3
2	Common operators and constants	3
3	Finite field representation example	5
4	Parameters for unified ECOH	6
5	Unified Pseudocode for ECOH	6
6	Pseudocode for ECOH-224	7
7	Pseudocode for ECOH-256	8
8	Pseudocode for ECOH-384	9
9	Pseudocode for ECOH-512	10
10	ECOH-224 examples	38
11	ECOH-256 examples	38
12	ECOH-384 examples	39
13	ECOH-512 examples	39

1 Introduction

The elliptic curve only hash (ECOH) algorithm is a submission to NIST’s SHA-3 competition. The ECOH algorithm is based on the MuHASH algorithm of Bellare and Micciancio [1].

1.1 Efficiency

For long messages, the ECOH algorithm is about a thousand times slower than SHA-1. Nevertheless, it does have some potential performance advantages over SHA-1.

- **Incrementality:** the ECOH hash of a message can be updated quickly, given a small change in the message and an intermediate value in the ECOH computation. Note MuHASH also has this property, but without requiring an intermediate value, in which case, the property can sometimes lead to an attack.
- **Parallelizability:** different parts of the ECOH computation can be done in parallel, because they do not depend on each other. Multiple CPUs are becoming common in personal computers.
- **Carryless multiplication:** ECOH works over binary fields, which are relatively slow on the CPU currently used in desktop because these CPUs only implement multiplication with carry operations. Intel recently announced that the upcoming AVX instructions will include carryless multiply operation.

Given the developments in desktop hardware towards parallelization and carryless multiplication, ECOH may in a few years be as fast as SHA-1 for long messages.

For short messages, ECOH is relatively slower, unless extensive tables are used. These tables are the same as one would use to speed up elliptic curve key generation (which is use in elliptic curve key agreement and signature generation).

1.2 Security

MuHASH, upon which ECOH is based, has not been successfully attacked. The closest attack has been Wagner’s attack [12] on AdHASH, a hash function faster and simpler than MuHASH, which Bellare and Micciancio [1] introduced in the same paper as MuHASH. Assuming MuHASH continues to resist attacks, it would seem that attack against ECOH may need to exploit the difference between ECOH and MuHASH. The main difference is that where MuHASH applies a random oracle, ECOH applies a padding function.

Assuming random oracles, Bellare and Micciancio [1] proved that finding a collision in MuHASH implies solving the discrete logarithm problem. For practicality, ECOH does not use a random oracle. It is shown here that finding low degree solutions to certain multivariate polynomial equations derived from Semaev’s summation polynomials [9] allows one to find collisions in ECOH. Therefore, the security of ECOH is not directly related to the discrete logarithm problem.

Semaev [10] first formulated these summation polynomial equations over binary field and proposed the problem of finding low degree solutions of them, in the context of the elliptic curve discrete logarithm over a Koblitz elliptic curve. Gaudry [4] used summation polynomials to the elliptic curve discrete logarithm problem over a cubic extension field in time faster than square root

of the curve size asymptotically. Note that Gaudry used solutions belonging a subfield, not merely low degree solutions in a binary field.

Therefore, Semaev's problem for summation polynomial fields has not so far been a given efficient algorithm to solve. Semaev [10] notes that Coppersmith's algorithm does something analogous for prime fields: it finds small solutions to certain small degree polynomials, though not necessarily summation polynomials. That Gaudry successfully applied to Semaev's idea to attack the elliptic curve discrete logarithm over cubic extension fields could be interpreted to mean that the original problem of applying them to Koblitz curves and other prime degree binary fields did not admit so easy a solution. Nevertheless, it is fair to say that this problem has not been nearly as well-studied as the elliptic curve discrete logarithm problem.

1.3 A More Cautious Variant

In a generalization of ECOH given here, an option is available that interferes with the formulation of the summation polynomial in terms of low degree solutions. Rather, the solutions instead must carry some intrinsic redundancy not easily expressible in the underlying field. The aim is to render the equations non-algebraic. Such enhancements may slow ECOH somewhat. Also, although these enhancements are simple enough to state, the effect upon security analysis is more complicated. Sometimes such an apparent complexity in the security analysis adds to security, but occasionally it does not, that is, it is sometimes does not add any computational complexity for the adversary.

H	Output of the hash function, a bit string
n	Length of H , in bits
M	The input message to be hashed, a bit string
m	Length of M , in bits
m	Length in bits of finite field elements
q	Field size (note $q = 2^m$)
$blen$	Length of message blocks
$ilen$	Length of index blocks
$clen$	Length of counter blocks
i	Index of message blocks
j	Number of zero bits used to pad
N	Padded message (sometimes also order of G)
k	Number of padded message blocks
E	Elliptic curve, including finite field and its representation.
G	A fixed point on the elliptic curve
N	A bit string obtained by padding M to a length which is a multiple of $blen$.
N_i	A substring of N of length $blen$.
I_i	A bit string of length $ilen$ representing integer i
O_i	A bit string of length $blen + ilen$
X_i	A bit string representing the x-coordinate of point
P_i	An elliptic curve point
x_i	A finite field element, the x-coordinate of P_i
y_i	A finite field element, the y-coordinate of P_i
Q	An elliptic curve point

Table 1: Common variables

2 Glossary

The elliptic curve operations used in ECOH are those used in FIPS 186-3 and [11]. Some familiarity with either of these standards is assumed. In particular, familiarity with finite fields of characteristic two is assumed and with elliptic curves defined over these fields, having equation of the form $y^2 + xy = x^3 + ax^2 + b$.

Table 1 defines the meaning of some common variables appearing in the specification of ECOH. Table 2 defines some operators and constants appearing in this specification, but variable names are not included.

Table 2: Common operators and constants

\parallel	Concatenation of bit strings. For example $0011101\parallel 110110 = 0011101110110$.
$ $	Divisibility of integers or polynomials. For example, $2 4$ and $x x^2 + x$ are true, but $2 3$ is false.
\oplus	Bit-wise exclusive of bit strings, which must be equal length. For example, $000111 \oplus 110110 = 110001$.

$\bigoplus_{i=0}^k$	Bit-wise exclusive of a sequence or array of bit strings. For example, if $B_0 = 00$, $B_1 = 01$ and $B_2 = 11$, then $\bigoplus_{i=0}^2 B_i = 00 \oplus 01 \oplus 11 = 10$.
0^j	Bit string of length j consisting of all zero bits. For example, $0^7 = 0000000$.
1	Bit string of length 1 consisting of a single one bit, or the integer, or finite field element, or polynomial over a finite field.
\mathcal{O}	The point at infinity on an elliptic curve. Defined to be the identity element of the elliptic curve group.
$(,)$	Coordinates of a point. For example, $(x, y) = (0, 1)$ is a point on the curve $y^2 + xy = x^3 + 1$.
+	Integer addition, finite field addition, finite field polynomial addition, or elliptic curve addition, depending on context. For example: $2 + 3 = 5$; $x + x = 0$; $(0, 1) + (0, 1) = \mathcal{O}$.
$\sum_{i=0}^k$	Addition of elliptic curve points indexed from $i = 0$ to $i = k$.
	Integer multiplication, or finite field multiplicataion, polynomial multiplication, or elliptic curve scalar multiplication, depending on context. That is, xy means the product of x and y . For numbers and multiletter variables, parentheses are used or an explicit \times may be used indicate multplication. For example, $i(\text{blen})$ is i times blen , and $7(11) = 77$, and $x(x + 1) = x^2 + x$, and $2(0, 1) = \mathcal{O}$.
a^b	Integer exponentiation, finite field exponentiation, or polynomial exponentiation, depending on the type of a . In all cases, b is non-negative integer. For example, $2^3 = 8$, and $(x + 1)^3 = x^3 + 3x^2 + 3x + 1$.
/	Rational number division or finite field division or polynomial division. For example, $6/2 = 3$, $3/2 = 1.5$, and $(x^2 + x)/(x + 1) = x$.
$\frac{a}{b}$	Alternative notation for division, that is, a/b .
$\lfloor \cdot \rfloor$	$\lfloor x \rfloor$ is largest integer not exceeding x . For example, $\lfloor 3/2 \rfloor = 1$ and $\lfloor 5 \rfloor = 5$.
$\lceil \cdot \rceil$	$\lceil x \rceil$ is smallest integer not less than x . For example, $\lceil 3/2 \rceil = 2$ and $\lceil 5 \rceil = 5$.
\mathbb{F}_q	Finite field with q elements.
$\langle \cdot \rangle$	$\langle G \rangle$ is the cyclic subgroup of elliptic curve group generated consisting of all elements nG for integers n .
A_i	A sequence or array of bit strings or elliptic curve points, indexed by i , where A is some variable name, which may also have a value when use alone. For example, perhaps $B_0 = 000$ and $B_1 = 110$ and $B_2 = 010$ could be a sequence of bit strings, whereas B may also indicate 0001.
$x(P)$	The integer whose bit string representation, padded with sufficiently many zeros on the left, equals the bit string representation of the x-coordinate of the elliptic curve point P . An exceptional case is that $x(\mathcal{O})$ is defined to be 0.
$x \bmod m$	The remainder obtained when dividing x by m . When x and m are integers, it is unique integer r such that $0 \leq r < m$ and $x = qm + r$ for some integer q , which can also be expressed as $x - m\lfloor x/m \rfloor$.

Finite field elements are represented as bit string using a polynomial basis representation. Table 3 gives example fields, irreducible polynomials for fields, and example elements in the field. The bit string representations of the field elements are given. Also given is the integer whose bit string representation, when padded with sufficiently zeros on the left, equals the bit string representation of the field element. Finally the hexadecimal representation is given, which is obtained by padding the bit string on the left with zeros until the length is a multiple of 8, and then converting to hexadecimal.

Field size	8	128
Irreducible polynomial	$t^3 + t + 1$	$t^7 + t + 1$
Example field element	$t + 1$	$t^4 + t^3 + t$
Bit string representation	011	0011010
Integer representation	3	26
Hexadecimal representation	03	1A

Table 3: Finite field representation example

3 Specification

The four ECOH algorithms have parameters as specified in Table 4. The four ECOH algorithms

Hash	n	E and G	m	$blen$	$ilen$	$clen$
ECOH-224	224	B-283	283	128	64	64
ECOH-256	256	B-283	283	128	64	64
ECOH-384	384	B-409	409	192	64	64
ECOH-512	512	B-571	571	256	128	128

Table 4: Parameters for unified ECOH

are specified by the pseudocode in Table 5 with the parameters above substituted as needed. The maximum bit length of a message that can be hashed is $2^{ilen} - 1$. For convenience, separate pseudocode is given for each of the four ECOH algorithms in Tables 6–9.

<ol style="list-style-type: none"> 1. Let $N = M\ 1\ 0^j$, with j chosen to be the smallest nonnegative integer such that the length of N is divisible by $blen$. 2. Parse N into blocks N_0, \dots, N_{k-1} each of length $blen$ bits. 3. Let $O_i = N_i\ I_i$, where I_i is the $ilen$-bit representation of the integer i, for i from 0 to $k - 1$. 4. Let $O_k = (\bigoplus_{i=0}^{k-1} N_i)\ I_{mlen}$ where I_{mlen} is the $ilen$-bit representation of the bit length $mlen$ of the message M. 5. Let $X_i = (0^{m-(blen+ilen+clen)}\ O_i\ 0^{clen}) \oplus C_i$ where C_i is the bit string of length m representing the smallest nonnegative integer c such that X_i represents the x-coordinate x_i of an element of $\langle G \rangle$. 6. Let $P_i = (x_i, y_i)$ be such that the rightmost bit of y_i/x_i equals the leftmost bit of N_i. 7. Let $Q = \sum_{i=0}^k P_i$. 8. Output the n-bit representation of $\lfloor x(Q + \lfloor x(Q)/2 \rfloor G)/2 \rfloor \bmod 2^n$.
--

Table 5: Unified Pseudocode for ECOH

Note that the four submitted ECOH algorithms are also instantiations of a generalized ECOH algorithm (§5), for which certain values of the parameters (§5.1) are selected.

3.1 ECOH-224

ECOH-224 uses the curve B-283, including generator point G , with parameters $(blen, ilen, clen) = (128, 64, 64)$.

1. Let $N = M\|1\|0^j$, with j chosen to be the smallest nonnegative integer such that the length of N is divisible by 128.
2. Parse N into blocks N_0, \dots, N_{k-1} each of length 128 bits.
3. Let $O_i = N_i\|I_i$, where I_i is the 64-bit representation of the integer i , for i from 0 to $k - 1$.
4. Let $O_k = (\bigoplus_{i=0}^{k-1} N_i)\|I_{mlen}$ where I_{mlen} is the 64-bit representation of the bit length $mlen$ of the message M .
5. Let $X_i = (0^{27}\|O_i\|0^{64}) \oplus C_i$ where C_i is the bit string of length 283 representing the smallest nonnegative integer c such that X_i represents the x-coordinate x_i of an element of $\langle G \rangle$.
6. Let $P_i = (x_i, y_i)$ be such that the rightmost bit of y_i/x_i equals the leftmost bit of N_i .
7. Let $Q = \sum_{i=0}^k P_i$.
8. Output the 224-bit representation of $\lfloor x(Q + \lfloor x(Q)/2 \rfloor G)/2 \rfloor \bmod 2^{224}$.

Table 6: Pseudocode for ECOH-224

3.2 ECOH-256

ECOH-256 uses the curve B-283, including generator point G , with parameters $(blen, ilen, clen) = (128, 64, 64)$.

1. Let $N = M\|1\|0^j$, with j chosen to be the smallest nonnegative integer such that the length of N is divisible by 128.
2. Parse N into blocks N_0, \dots, N_{k-1} each of length 128 bits.
3. Let $O_i = N_i\|I_i$, where I_i is the 64-bit representation of the integer i , for i from 0 to $k - 1$.
4. Let $O_k = (\bigoplus_{i=0}^{k-1} N_i)\|I_{mlen}$ where I_{mlen} is the 64-bit representation of the bit length $mlen$ of the message M .
5. Let $X_i = (0^{27}\|O_i\|0^{64}) \oplus C_i$ where C_i is the bit string of length 283 representing the smallest nonnegative integer c such that X_i represents the x-coordinate x_i of an element of $\langle G \rangle$.
6. Let $P_i = (x_i, y_i)$ be such that the rightmost bit of y_i/x_i equals the leftmost bit of N_i .
7. Let $Q = \sum_{i=0}^k P_i$.
8. Output the 256-bit representation of $\lfloor x(Q + \lfloor x(Q)/2 \rfloor G)/2 \rfloor \bmod 2^{256}$.

Table 7: Pseudocode for ECOH-256

3.3 ECOH-384

ECOH-384 uses the the curve B-409, including generator point G , with parameters $(blen, ilen, clen) = (192, 64, 64)$.

1. Let $N = M||1||0^j$, with j chosen to be the smallest nonnegative integer such that the length of N is divisible by 192.
2. Parse N into blocks N_0, \dots, N_{k-1} each of length 192 bits.
3. Let $O_i = N_i||I_i$, where I_i is the 64-bit representation of the integer i , for i from 0 to $k - 1$.
4. Let $O_k = (\bigoplus_{i=0}^{k-1} N_i)||I_{mlen}$ where I_{mlen} is the 64-bit representation of the bit length $mlen$ of the message M .
5. Let $X_i = (0^{89}||O_i||0^{64}) \oplus C_i$ where C_i is the bit string of length 409 representing the smallest nonnegative integer c such that X_i represents the x-coordinate x_i of an element of $\langle G \rangle$.
6. Let $P_i = (x_i, y_i)$ be such that the rightmost bit of y_i/x_i equals the leftmost bit of N_i .
7. Let $Q = \sum_{i=0}^k P_i$.
8. Output the 384-bit representation of $\lfloor x(Q + \lfloor x(Q)/2 \rfloor G)/2 \rfloor \bmod 2^{384}$.

Table 8: Pseudocode for ECOH-384

3.4 ECOH-512

ECOH-512 uses the curve B-571, including generator point G , with parameters $(blen, ilen, clen) = (256, 128, 128)$. It can hash messages of bit length up to $2^{128} - 1$.

1. Let $N = M\|1\|0^j$, with j chosen to be the smallest nonnegative integer such that the length of N is divisible by 256.
2. Parse N into blocks N_0, \dots, N_{k-1} each of length 256 bits.
3. Let $O_i = N_i\|I_i$, where I_i is the 128-bit representation of the integer i , for i from 0 to $k - 1$.
4. Let $O_k = (\bigoplus_{i=0}^{k-1} N_i)\|I_{mlen}$ where I_{mlen} is the 128-bit representation of the bit length $mlen$ of the message M .
5. Let $X_i = (0^{59}\|O_i\|0^{128}) \oplus C_i$ where C_i is the bit string of length 571 representing the smallest nonnegative integer c such that X_i represents the x-coordinate x_i of an element of $\langle G \rangle$.
6. Let $P_i = (x_i, y_i)$ be such that the rightmost bit y_i/x_i equals the leftmost bit of N_i .
7. Let $Q = \sum_{i=0}^k P_i$.
8. Output the 512-bit representation of $\lfloor x(Q + \lfloor x(Q)/2 \rfloor G)/2 \rfloor \bmod 2^{512}$.

Table 9: Pseudocode for ECOH-512

4 Efficiency Analysis

In this section, the efficiency of ECOH is discussed.

4.1 Reference Implementation

The reference implementation uses well-known elliptic curve arithmetic algorithms as surveyed in [7].

Our measurements of the reference implementation of ECOH give it rate of about 1000 times slower than SHA-1 for long messages.

About three quarters of the time that the reference implementation requires to compute an ECOH digest of a long message is spent computing inverses of elements in the field over which the elliptic curve is defined.

On average, three inversions must be computed for each message block. Some of these inversions are needed to search for a point on the curve corresponding to the message block. On average, two inversions are needed to find this point. Once the point is found, it must be added into an accumulator. The reference implementation uses affine coordinates, in which the addition formula includes an inversion.

Affine coordinates were chosen over projective coordinates, because affine coordinates run faster on current Intel processors. More specifically, in binary fields, the cost M of finite field multiplication and the cost I of finite field inversion have a ratio $I/M \approx 7$. The reason that projective coordinates were not used is that, although their addition formula generally avoid an inversion, they require many more multiplication operations, typically eight or more, which is more expensive than an inversion operation in the reference implementation.

After inversions the bulk of the remaining computation time in the reference implementation on long messages is spent performing finite field multiplication. Most of this time is spent on binary polynomial multiplication, which is implemented using combing. The reference implementation spends relatively little time on modular reduction, polynomial addition, computation of the trace and half-trace. Traces are implemented by examining fixed bits of the polynomial basis representation. Half-traces are computed with table lookups and polynomial addition.

For short messages, the scalar multiplication in the postprocessing requires relatively more time.

4.2 Optimized Implementation

The optimized implementation submitted differs from the reference implementation as follows.

- It uses the almost inverse algorithm, instead of the Euclidean algorithm for computing inverses.
- It unrolls some loops?

It is about twice as fast as the reference implementation.

4.3 Assembly Optimization

Hardware specific assembly optimization can sometimes result in more efficient executables than compiled C source code.

The submitted reference and optimized implementations do not use any specific assembly. That is, they are written entirely on C, and rely entirely on the compiler for assembling.

Hand assembly for specific platforms should make the ECO_H faster.

4.4 Simultaneous Inversion

Simultaneous inversion, introduced by Peter Montgomery, trades k inversions for 1 inversion and $3(k - 1)$ multiplications. Considering that $I/M > 3$ and that most of the time for performing ECO_H is spent on inversion, simultaneous inversion should yield a speed up for ECO_H.

Simultaneous inversion is not implemented in the submission. The theoretical benefit of simultaneous inversion, for long messages, is to reduce one unit of time to $\frac{3}{I/M} \frac{3}{4} + \frac{1}{4}$, assuming fraction of the time spent on inversion is $\frac{3}{4}$ and that k , the number of inversion done simultaneously, is chosen to be sufficiently large. Taking $I/M = 7$, this makes ECO_H a little less than twice as fast.

That said, however, simultaneous inversion has some overhead costs, of determining which inversion to do simultaneously.

4.5 AVX instructions

Intel recently announced *Advanced Vector Extensions* (AVX), which are a new set of instructions to be added to its upcoming processors. Included in AVX is an instruction PCLMULQDQ which may be useful for making elliptic curve cryptography over binary fields more efficient.

Gueron and Kounavis [6] suggest an approximate 21 or 66 times speed up of scalar multiplication. To get this, they assumed a reasonable cost of the PCLMULQDQ instruction. They compare projective coordinates implemented with PCLMULQDQ to affine coordinates implemented with combining multiplication and a variant of the Euclidean algorithm.

Since projective coordinates replace one inversion with 8 multiplies one may infer that they find $I/M \approx (8)(21) = 168$. This is clearly a crude estimate, but it does strongly suggest that with AVX, the ratio I/M will become considerably larger.

With so large a value I/M , it makes much more sense to use projective coordinates and simultaneous inversion.

Given the estimates above, ECO_H would be 25 times faster than the reference implementation if implemented with AVX and simultaneous inversion.

4.6 Parallelization

The ECO_H algorithm is parallelizable in the following ways:

- the search and recovery of the term points may be done in parallel,
- computing the checksum may be done in parallel,
- computing the total point from the term points may be done in parallel.
- computing the scalar multiply may be done in parallel, if a sufficiently large table is used.

The amount of parallelization possible in ECO_H depends on the hardware used and on the message size.

It should be noted that simultaneous inversion requires a serial sequence of multiplication operations. For short messages, therefore, it may be possible to benefit from both simultaneous inversion and parallelization of the point searching, recovery and additions. For longer messages, though

grouping a moderate number of simultaneous inversion together should capture almost all of the benefit of simultaneous inversion. Since there will be many of these grouped inversion, the benefits of parallelization can be obtained by doing these multiple simultaneous inversions in parallel.

For example, if the hardware permits 40 parallel operations and quick AVX binary polynomial multiplication operations, then for sufficiently long messages where simultaneous inversions gives 25 times speedup over the reference implementation, then the parallelism will give another 40 times speedup, for a final speedup of 1000 times. In this case, ECO_H would have a speed comparable to SHA-1.

5 Generalization of ECOH

This section is a generalization of the ECOH algorithm. In the event that NIST finds that ECOH is suitable for SHA-3 except for some minor deficiencies, then it is suggested that some version of the generalized ECOH specified in this section be considered.

5.1 Generalized ECOH Parameters

Generalized ECOH has the parameters list below. Parameters listed as *required* are needed to comply with certain SHA-3 requirements, or to ensure security, or at least to provide some concrete definition. Parameters listed as *recommended* include those used in the submitted version of ECOH, as specified in §3. Many of the parameters can be made higher to strengthen the security, which may be desirable for SHA-3.

Conversely, lowering some of the parameters, especially below the recommended values, should result in a weakened version of ECOH, whose security is easier to ascertain. For example, using a small elliptic curve may make finding a collision possible. The computational cost for finding collisions in such weakened version of ECOH may be interpolated to estimate the computational cost of finding a collision in the submitted version of ECOH or the generalized form of ECOH.

- Output length n bits.
 - Required: $n \in \{224, 256, 384, 512\}$.
- Binary field degree m .
 - Required: $m > n$.
 - Recommended: $m \in \{283, 409, 571\}$
- Binary field representation: a bijection $\rho : \mathbb{F}_{2^m} \rightarrow \{0, 1\}^m$.
 - Recommended: polynomial basis representation,
 - Recommended: left bits are highest degree bits,
 - Recommended: minimal Hamming weight irreducible polynomial basis representation,
 - Recommended: Representations given in FIPS 186-2 for B-283, B-409, B-571.
- Elliptic curve parameters $a, b \in \mathbb{F}_{2^m}$.
 - Recommended: the NIST curves B-283, B-409 and B-571, as appropriate with the other parameters.
- Elliptic curve generator $G \in E(\mathbb{F}_{2^m})$ (also known as base point).
 - Required: the cyclic subgroup $\langle G \rangle$ generated by G of the elliptic curve $E : y^2 + xy = x^3 + ax^2 + b$ must not be such that the discrete log problem in $\langle G \rangle$ is known to be solvable with computation less than the equivalent $2^{n/2}$ elliptic curve operations.
 - Recommended: the value of G including B-283, B-409 and B-571, as appropriate with other parameters.

- Elliptic curve subgroup index h .
 - Requirement: the order of G is $|E(\mathbb{F}_{2^m})|/h$.
 - Recommended: $|E(\mathbb{F}_{2^m})|/h$ is prime. (In which case, h is also called the cofactor.)
 - Recommended: $h = 2$.
- Message block size $blen$.
 - Required: $blen \geq 96$.
 - Recommended: $blen = 128$ if $n \in \{224, 256\}$, $blen = 192$ if $n = 384$, $blen = 256$ if $n = 512$.
 - Recommended: $blen \geq 128$.
 - Recommended: $32|blen$.
- Index block size $ilen$.
 - Required: $ilen \geq 64$.
 - Recommended: $ilen = 64$ unless $n = 512$, then $ilen = 128$.
 - Recommended: $32|ilen$.
- Counter block size $clen$.
 - Required: $clen \geq 64$.
 - Recommended: $clen = 64$, unless $n = 512$, then $clen = 128$.
 - Recommended: $32|clen$.
- Bits dropped before integer conversion: subset of $\{0, 1, \dots, m - 1\}$, corresponding to radix significance.
 - Recommended: (for use with polynomial representation) $\{0\}$, where 0 indicates the degree of the bit (so this the rightmost or last bit). (If the bits are counted from the left, then this is bit $m - 1$.)
- Unsigned integer encoding:
 - Required: as bit strings of a given length, with lower radix bits on the right.
 - Recommended: consistency with the polynomial basis representation, so the higher degree bits correspond to higher radix bits.
- Bits kept after integer conversion to hash output: subset of $\{0, 1, \dots, m - 2\}$, corresponding to radix significance.
 - Recommended: the n least significant $\{0, 1, \dots, n - 1\}$.
- Shift factor β .
 - Required: $\beta \in \{0, 1\}$.

- Recommended: $\beta = 1$.
- Tail obfuscator ω .
 - Required: $\omega \in \{0, 1\}$.
 - Required: if $\omega = 1$, then $blen \geq 128$.
 - Recommended: $\omega = 0$.
- Encryptor flag ϵ .
 - Required: $\epsilon \in \{0, 1, 2, 3\}$.
 - Recommended: $\epsilon = 0$.
- Addition flag: α .
 - Required: $\alpha \in \{0, 1\}$.
 - Recommended: $\alpha = 0$.

5.2 Generalized ECOH Operations

In this section, the generalized ECOH algorithm is specified in small modular pieces. An outline of the pieces is given below and grouped into phases in such a way to facilitate further discussion.

1. Input
2. Bit string preprocessing phase
 - (a) Padding
 - (b) Parsing
 - (c) Indexing
 - (d) Tailing
3. Elliptic curve preprocessing phase,
 - (a) Searching
 - (b) Recovering
4. Elliptic curve main summation phase
 - (a) Summing
5. Elliptic curve postprocessing phase
 - (a) Converting
 - (b) Multiplying
 - (c) Shifting
6. Bit string postprocessing phase

- (a) Converting
 - (b) Selecting
7. Output

5.2.1 Input

The input to ECOH is a message M , which is a bit string of arbitrary length. The length of M in bits is denoted by $m\text{len}$.

5.2.2 Padding

Message M is padded with a single 1 bit and then with the minimal number of 0 bits needed to obtain a bit string whose length is a multiple of blen to obtain a bit string N that is the padded version of the input bit string M . Using $\|$ to indicate bit string concatenation, 0^j to indicate a string of length j all of whose bits are 0, and $(a \bmod b)$ to indicate the remainder of a divided by b , then the formula for N may also be expressed as:

$$N = M\|1\|0^{((-1-m\text{len}) \bmod \text{blen})} \quad (1)$$

Note that the length N as a bit string may be expressed as:

$$\text{blen} \left\lceil \frac{1 + m\text{len}}{\text{blen}} \right\rceil \quad (2)$$

where $\lceil x \rceil$ indicates the smallest integer $i \geq x$. Let $k = \lceil \frac{1+m\text{len}}{\text{blen}} \rceil$, so that the bit length of N is $k(\text{blen})$. We may refer to k as the block count, if needed.

5.2.3 Parsing

The padded message N is parsed into k blocks, each a bit string of length blen bits. The blocks are indicated as N_i where i is an integer and $0 \leq i < k$. Using concatenation notation $\|$ we will have that

$$N = N_0\|N_1\|\dots\|N_{k-1}. \quad (3)$$

Using array indexing of bit strings, where $B[j]$ is the bit of the bit string B at a distance of j bit positions from the leftmost bits of B (so that $B[0]$ is the left most bit, for example), then N_i may be expressed as the unique bit string of length blen such that:

$$N_i[j] = N[j + i(\text{blen})] \quad (4)$$

for all integers j with $0 \leq j < \text{blen}$.

5.2.4 Indexing

Each parsed block N_i for integers $0 \leq i < k$ is processed to obtain an indexed block O_i as follows

$$O_i = N_i\|I_i \quad (5)$$

where I_i is the bit string of length $i\text{len}$ representing the unsigned integer i . We refer to O_i as a body indexed block.

5.2.5 Tailing

Let

$$N_k = \bigoplus_{i=0}^{k-1} N_i = N_0 \oplus N_1 \oplus \cdots \oplus N_{k-1} \quad (6)$$

where \oplus indicates the bit-wise exclusive operation upon bit strings. Let

$$O_k = N_k \parallel I_{mlen} \quad (7)$$

where I_{mlen} is the bit string of length $ilen$ representing the length $mlen$ of the original input message M . Note well that $I_{mlen} \neq I_k$ (unless $mlen = 1$) so the index portion of O_k generally does not represent the integer k , as one might erroneously assume if one just interpolated from the index portions of the previous values O_i . In other words, the index portion of O_k is higher as an integer by a factor of about $blen$ than the index portion of O_{k-1} . We refer to O_k as the tail indexed block, and to N_k as the checksum.

If $\omega = 1$ (which requires $blen \geq 127$), then the leftmost 127 bits of N_k are not zero, then they are modified as follows. They are first converted to an integer w (which will necessarily be nonzero). The polynomial $f(x) \equiv x^w \bmod x^{127} + x + 1$ is computed. This is a polynomial of degree at most 126. This is converted to a bit string W of length 127 bits, where the leftmost bit is the coefficient of x^{126} and so on. Then the leftmost 127 bits of N_k are replaced by W .

Indexed block refers to either a body indexed block or a tail indexed block.

5.2.6 Searching

For each indexed block N_i , the following loop is executed. An integer counter variable c is initialized to 0. The variable c will be incremented at each stage of the loop. The following bit string length m is initialized:

$$X = 0^{m-(blen+ilen+clen)} \parallel O_i \parallel 0^{clen} \quad (8)$$

At each iteration of the loop, the counter c is represented as a bit string C of length m bits. Then the bit string

$$X_{i,c} = X \oplus C \quad (9)$$

is computed.

For values of $\epsilon \neq 0$, the following extra transformation is performed on $X_{i,c}$.

- If $\epsilon = 1$, then let $t = blen + ilen + clen$. The rightmost t bits $X_{i,c}$ are represented as a finite field \mathbb{F}_{2^t} , using polynomial basis representation with the least lexicographically irreducible polynomial. If this element is non-zero, then its inverse is computed and its bit string representation replaces the rightmost t bits of $X_{i,c}$. Note that $\epsilon = 1$ is not recommended as part of the ECOH submission, though it may be considered for inclusion in SHA-3 as an extra safety margin for making certain attacks appear harder.
- If $\epsilon = 2$, then $X_{i,c}$ is transformed as follows. As few bits as possible are discarded from the left, so that the result has a length that is a multiple of 64 bits. Then AES key wrap is applied, using the AES-128 with key of all zero bits. As few bits as possible are discarded from the left so that the length equals m . This is the transformed value of $X_{i,c}$. Note

that $\epsilon = 2$ is not suggested as part of the ECOH submission, though it may be considered for an inclusion in SHA-3 as a safety margin for making certain attacks appear harder and for making certain heuristic security arguments more plausible [2].

- If $\epsilon = 3$, then $X_{i,c}$ is transformed as follows. It is fed into another SHA-3 submitted hash function. The resulting n bits then replace the rightmost n bits of the $X_{i,c}$. Note that $\epsilon = 3$ is not suggested as part of the ECOH submission, though it may be considered for inclusion in SHA-3 as an extra safety margin that appears to make certain attacks harder, and for making an algorithm that more closely resembles MuHASH [1].

Next $X_{i,c}$ is converted to finite field element $x_{i,c}$. Mathematically,

$$x_{i,c} = \rho^{-1}(X_{i,c}), \tag{10}$$

where ρ is the generalized parameters which is function from the field elements to bit strings. This element $x_{i,c}$ is called a candidate x-value.

The candidate x-value $x_{i,c}$ is tested as follows. If there exists a field element y such that $(x_{i,c}, y)$ is an element of the group $\langle G \rangle$, then $x_{i,c}$ is valid. Otherwise $x_{i,c}$ is invalid.

This loop runs until a valid $x_{i,c}$ is found. The resulting may be denoted as x_i . The final value of the counter may be indicate as c_i . Note the c_i is the smallest non-negative integer such that x_{i,c_i} is valid.

5.2.7 Recovering

There will generally be two possible values of y that correspond to a valid x_i . One of these will be chosen according to the criterion below, and will be indicated as y_i .

The chosen y-coordinate will be determined by the one which satisfies the following properties. When the field element y_i/x_i is represented as a bit string of length m , its rightmost bit equals the leftmost bit of N_i .

The point (x_i, y_i) will be indicated as P_i and may be referred to as a term point.

5.2.8 Summing

The point

$$Q = \sum_{i=0}^k P_i = P_0 + \dots + P_k \tag{11}$$

is computed. Note that the summation here is in the elliptic curve group, not addition in the finite field or ring of integers. The point Q may be referred as the prehash or total point.

5.2.9 Converting

The x-coordinate $x(R)$ of elliptic curve point Q is converted to a bit string $U = \rho(x(R))$. Some of the bits of U , as indicated by the parameters, are dropped, obtaining a bit string V . Then V is converted to an integer v .

Note the recommended parameters indicate that the rightmost bit of U is dropped. In this case, if the x-coordinate of U when represented as a bit string represents the integer u , then $v = \lfloor u/2 \rfloor$.

The integer v may be referred to as the multiplier.

5.2.10 Multiplying

The point $S = vG$ is computed. Note that this is elliptic curve scalar multiplication; that is, v copies of G summed using elliptic curve operations.

This point may be referred to as the total scaled point.

5.2.11 Shifting

The point $R = S + \beta Q$ is computed. Note that $\beta \in \{0, 1\}$, so $R \in \{S, S + Q\}$. This point may be referred to as the shifted total.

If $\alpha = 1$, then the x-coordinate of R and the x-coordinate of Q are converted to integers and added modulo 2^m . The resulting integer is encoded into a bit string of length m . This is encoded into a field finite element. The result field element replaces the x-coordinate of the point R . The y-coordinate of R is not changed, and is not used in the ensuing calculation. Note that R , once modified in this way, is a point, but it is very unlikely to be on the elliptic curve in question.

5.2.12 Converting

The same point-to-integer conversion process applied to Q is applied to R . This conversion is restated explicitly below, with new variable names.

The x-coordinate $x(R)$ of R is converted to a bit string $W = \rho(x(R))$. Some of the bits of W , as indicated by the parameters, are dropped, obtaining a bit string Z . Then Z is converted to an integer h .

Note the recommended parameters indicate that the rightmost bit of Z is dropped. In this case, if the x-coordinate R , when represented as a bit string, also represents the integer r , then $h = \lfloor r/2 \rfloor$.

The integer h may be referred to as the integral hash.

5.2.13 Selecting

The integral hash h is represented as a bit string F . Some subset, as indicated by the parameters of the algorithm, of the bits of F are selected for inclusion in a bit string H of length n .

Note that, for the recommended parameters, the n rightmost bits of the bit string representation of h are used to form H .

5.2.14 Output

The bit string H of length n is the output of the ECOH hash function.

5.3 Suggested alternative values of parameters

5.3.1 Curves for deterministic runtime

From certain implementation perspectives, the searching step of generalized ECOH is undesirable in that its runtime is nondeterministic. This can be overcome by use of different elliptic curves and binary fields, as follows.

- Use a binary field \mathbb{F}_{2^m} , where $m = 2p$ for some prime p . Note that \mathbb{F}_{2^p} is a subfield of \mathbb{F}_{2^m} .

- Use an elliptic curve with equation $y^2 + xy = x^3 + ax^2 + b$ such that $a, b \in \mathbb{F}_{2^p}$. Note that since the coefficients of the curve equation, the curve may be called a subfield curve.
- Use an encoding of $\rho : \mathbb{F}_{2^m} \rightarrow \{0, 1\}^m$ such that elements of \mathbb{F}_{2^p} are mapped to bit strings whose leftmost p bits are zeros. (A tower of polynomial basis representation may be used to achieve this.)
- Choose parameters $blen, ilen, clen$ such that $m \cdot len + ilen + clen \leq p$.

In this case, it can be shown any candidate x-coordinate will be such that $x_{i,c} \in \mathbb{F}_{2^p}$ (not just in \mathbb{F}_{2^m} . Each element in \mathbb{F}_{2^p} will be the x-coordinate of a point on the curve with y-coordinate in \mathbb{F}_{2^m} . Therefore, the only remaining condition on $x_{i,c}$ is that it also belong subgroup generated by G .

There exist efficient algorithms to test if a point on the curve is also in the subgroup generated by G , using only the x-coordinate of that point.

Also, it can be shown that for about half of all field elements in \mathbb{F}_{2^p} the resulting y-coordinate will also be in \mathbb{F}_{2^p} . In the other half of x-coordinates in \mathbb{F}_{2^p} , it is possible to transform the point on the curve to the twist of the curve with both coordinates in \mathbb{F}_{2^p} . In both cases, it is possible to work entirely over the field \mathbb{F}_{2^p} rather than \mathbb{F}_{2^m} , at least until the final total point needs to be calculated.

Satoh and others have developed very efficient point counting algorithms for binary curves. Therefore it is possible to fix some a , say $a = 0$, and then choose random $b \in \mathbb{F}_{2^p}$, until one obtains a curve and its twist, whose orders are twice a prime and four times another prime, respectively, or vice versa. This would be ideal for making ECOH run in less variable time.

A major disadvantage of this alternative is that the NIST recommended curves would not be used.

5.3.2 Inverted ECOH

Generalized ECOH with parameter value $\epsilon = 1$ seems to thwart the most plausible collision finding attack on ECOH: solving Semaev summation polynomials.

5.4 Incremental mode of operation

If one has already computed the ECOH hash H of a message M , and then a small change is made to the message, such as the re-assignment of a few bits, to obtain another message M' , it is possible to accelerate the computation of the ECOH hash H' of message M' by using some of the intermediate computation in the computation of the hash of H .

More precisely, let M' and M have bit lengths $m \cdot len'$ and $m \cdot len$, respectively, and let $\Delta = M' \oplus M$, where whichever of M or M' is shorter is extended by padding with zero bits on the right. If Δ is sufficiently sparser than M' , then given M' and Δ , and certain intermediate values in computation of M , it is possible to compute H' more quickly than computing H' directly from M' .

The intermediate values to allow acceleration computation of M , are $m \cdot len, Q$ and N_k (before obfuscation). The bit strings Δ and N_k can be used to compute the revised checksum N'_k before obfuscation, by parsing Δ into appropriately sized blocks, and xoring all of these into N_k . The revised total point Q' can also be computed by subtracting off any old contributions from M that differ from those of M' (which may be located using Δ) and then adding the corresponding contributions of M' .

Note that ECOH, in this mode of operation, like MuHASH, is incremental only the sense of in-place changes. In particular, insertions and deletions are not supported.

Insertions and deletions could be handled by using a different data structure, such as a linked list or tree. Specifying such data structure is beyond the scope of the ECOH specification. Nevertheless, because of the importance of insertion and deletion some discussion of this is provided.

Suppose that the actual content is a bit string. Before hashing the content undergoes two conversions. The first is conversion to a more flexible data structure, such as a linked list. The second is a conversion back to a bit string. Note that the resulting bit string would generally be considerably longer than the original content bit string, because it encodes the overhead information in the data structure used, such as the link pointer addresses in the link list records. This longer bit string is fed into the ECOH algorithm.

When an insertion occurs in the original bit string content, the linked list may be updated by the creation of new records, and changing a few link pointer addresses. When this revised linked list is encoded as a bit string, the link changes are in-place changes, while the new records may be added to the tail of the old the bit string. This therefore allows accelerated computation of the revised ECOH hash.

When a deletion occurs in the original bit string content, the linked list may generally be updated by the changing of a few link pointer addresses, and by dropping a few records, and perhaps creation of a few new records. When this revised linked list is encoded as a bit string, the link changes are in-place changes, and the creation of new records, can be done by appending them to the end of the bit string. The dropped records may have been encoded somewhere in the middle of the bit string. These records are not pointed in the revised message, so they may be left there, effectively ignored. Again all changes are in-place so computing the ECOH hash of the revised bit string can be accelerated.

With this approach, after many insertions and deletions, garbage records will collect, and the records will be place out of sequence.

6 Security

This section describes the security of ECOH. For the sake of concreteness, the scope is limited to the assessment of various plausible attack strategies. For a more theoretical analysis involving reductionist security arguments, sometimes known as proofs of security, see §A.

The focus will be on the submitted version of ECOH, with only limited attention paid to generalized ECOH.

6.1 Collision Resistance

A collision in hash function H is a pair of messages (M, M') such that $M \neq M'$ and $H(M) = H(M')$. Any hash function with a larger domain than range must have a collision. In theory, there exists an efficient algorithm, namely one that outputs M and M' directly to find a collision. In practice, we do not know how to find this algorithm. To speak of collision resistance, therefore, means to measure the efficiency of all known algorithms that can find collisions. This section lists some collision finding algorithms.

6.1.1 Birthday Attack

The birthday collision finding algorithm is as follows. Fix some finite message space, such as all bit strings of length $4n$. Choose a random message from the message space. Compute its hash. Repeat until two of the hashes computed are identical. Note that the birthday collision finding algorithm is defined for any hash function.

Consider the heuristic assumption that, when the inputs are uniformly distributed in the message space the hash function's outputs are nearly uniformly randomly distributed in the hash output range. For message spaces sufficiently large and independent of the hash function's definition, such as all bit strings of a given length larger than n , then this heuristic seems reasonable if the output of the hash function appears pseudorandom.

Under this heuristic, the birthday collision finding algorithm is expected to find a collision after computing about $2^{n/2}$ hash values.

The heuristic can fail for certain message spaces. If the message space is too small, then the outputs may not be uniformly distributed. For example, if the message space is all bit strings of length $n/3$, then there can at most be $2^{n/3}$ outputs, and therefore most outputs will not occur. In this case, the birthday collision finding algorithm will fail. The heuristic can also fail for large message spaces if they are dependent on the hash function. For example, if the message space is defined to be all message whose hash is the all zeros bit string. With this type of failure of the heuristic, the birthday collision finding algorithm finds collision faster. However, such message spaces, although they exist, may not be amenable to efficient sampling of random elements. The heuristic can also fail if the hash function is not sufficiently pseudorandom.

Regardless, it seems reasonable that the birthday collision finding algorithm will find collision at a cost of about $2^{n/2}$ hash evaluations or fewer provided the message space used is sufficiently larger than 2^n and that the message space can be sampled efficiently. The birthday collision finding algorithm is the benchmark by which other collision finding algorithms are compared.

6.1.2 Semaev Summation Polynomial Attack

This collision finding attack strategy attempts to find a collision in the x-coordinate of the total point (prehash) Q , using Semaev's summation polynomials [10], which are now reviewed.

Let \mathbb{F} be a finite field. Let E be an elliptic curve with Weierstrass equation having coefficients in \mathbb{F} . Let \mathcal{O} denote the point of infinity, which is the identity (neutral) element of the elliptic curve group. Semaev proved that there exists a multivariable polynomial $f_n = f_n(X_1, \dots, X_n)$ with the following property. For any $x_1, x_2, \dots, x_n \in \mathbb{F}$, there exists y_1, \dots, y_n such that

$$(x_1, y_1) + \dots + (x_n, y_n) = \mathcal{O} \tag{12}$$

if and only if

$$f_n(x_1, \dots, x_n) = 0. \tag{13}$$

Furthermore, the polynomial f_n has degree 2^{n-2} in each variable X_i . (Its total degree may be less than $n2^{n-2}$.)

To use summation polynomials to attack ECOH, we try to find M and M' such that $Q = Q'$, where Q' indicates the prehash obtained in computing the ECOH hash of M' . More generally, for any intermediate values that one might get in the computation of the ECOH hash of M and write with symbol s , we write s' for the corresponding symbol in the computation of M' .

A $Q = \pm Q'$ collision means that

$$P_0 + \dots + P_k \pm ((-P'_0) + \dots + (-P'_{k'})) = \mathcal{O} \tag{14}$$

According to Semaev's result, this implies that

$$f_{k+k'}(x_0, \dots, x_k, x'_0, \dots, x'_{k'}) = 0, \tag{15}$$

because, a point $-P$ and P have the same x-coordinate.

Note that (14) and (15) are not equivalent. Indeed, Semaev's result only states that if (15) is true then at least one of the $2^{k+k'}$ possible choices of possible y-coordinates $(y_0, \dots, y_k, y'_0, \dots, y'_{k'})$ gives (14).

An attack strategy is to find M and M' such that (15) holds. A reasonable heuristic is the y-coordinates used by ECOH will vary randomly and independently of the y-coordinates needed in (14). In other words, once equation (15) is satisfied, then we have probability of about $2^{-(k+k')}$ of getting the desired collision in $Q + Q'$.

This may seem to require that $k + k'$ be made low. However, in fact, the adversary may have a better variant of this strategy, where a low index summation polynomial may be used for longer messages. The adversary can fix certain blocks of M and M' , and only allow a few to vary, say v in total. For example, the adversary could fix M' entirely, and fix all but the first two blocks of M . Furthermore, the adversary could vary N_0 and N_1 such that $N_0 \oplus N_1$ is some constant value, which means that the checksum point P_k is fixed too. Then (14) may be re-written as

$$P_0 + P_1 + F = \mathcal{O} \tag{16}$$

where F is some fixed point, and P_0 and P_1 are unknowns. Using summation polynomials this implies

$$f_3(x_0, x_1, x) = 0 \tag{17}$$

where x is the x-coordinate of the fixed point F . A solution to (17) should have at least a 12.5% chance of giving a solution to (16) and thereby a collision in ECOH. The equation (17) has degree two in each of the variables x_0 and x_1 , and since x is constant, we may re-write the equation as

$$g(x_0, x_1) = 0 \tag{18}$$

where now the fixed value x is incorporated into the definition of g .

At this point, we may observe that by definition of ECOH, $x_i = a_i + n_i z$ where a_i , z and n_i are field elements of low degree (in the polynomial basis representation, which we now assume). Furthermore, n_i is derived directly from the message, and z is fixed value, used to shift the message bits to the left inside the x-coordinate. The value a_i includes the index and the counter. We can expect the counter value to be quite low, so the adversary can make reasonable guesses at b_i using low values. Therefore, the adversary can try to solve (18) by solving the following equation

$$h(n_0, n_1) = 0, \tag{19}$$

where n_0 and n_1 are now low degree field elements, and the polynomial h is derived from g by incorporation of the fixed values z and the guessed values for b_0 and b_1 .

But recall that $N_0 \oplus N_1$ was fixed in this version of the attack. Consequently, n_1 may be expressed in terms of n_0 , the constants $N_0 \oplus N_1$ and z , and the guesses b_0 and b_1 . Therefore, we get a single equation

$$q(n_0) = 0 \tag{20}$$

This equation will be quartic. It will have, therefore, at most four roots. Making a heuristic assumption that the four roots are random field elements, then the probability that they sufficiently low degree to serve as n_0 is negligible.

The attacker may choose to get around this by varying more of the message, say n_0 and n_1 and n_2 . In this case, a two variable polynomial is obtained, and the adversary's task is to find low degree solutions. Such solutions may exist. Semaev notes [10] that finding low degree solutions to multivariate polynomials allows one to find discrete logarithms in elliptic curves more quickly. This suggests that finding low degree roots is difficult, even they exist, because the elliptic curve discrete logarithm has appeared to remain difficult despite [10]. Since discrete logs are expected to take $2^{m/2}$ steps to solve, finding low degree solutions to the multivariate polynomials deduced from Semaev summation polynomials, as described above, may also take $2^{m/2}$ steps. Therefore, this attack seems no better than the birthday attack.

That said, the low degrees in [10] may be considerably lower than those needed against ECOH. Further research may be needed to assess the difficulty of these problems.

In the event that this attack is discovered to require less work than $2^{n/2}$ bit operations, such as if finding sufficiently low degree solutions to polynomials above can be done this efficiently, then this deficiency may be resolved by amending ECOH to make it more like MuHASH. This could be done by setting $\epsilon \in \{1, 2, 3\}$ in the generalized ECOH algorithm. In this case, the attack described above does not work because a transformation is applied mixing the message bits and the index bits and the counter bits, so that the resulting x-coordinate cannot so readily be expressed as a predetermined polynomial of a unknown but low degree finite field element. Specifically, a polynomial equation as in (19) cannot be directly derived from (18).

6.1.3 Shifted collision attacks

The recommended parameter $\beta = 1$ is intended to provide preimage resistance. Unlike the alternative parameters $\beta = 0$ choice, it introduces the possibility that $Q \neq \pm Q'$ but $R = \pm R'$. This would mean that

$$Q \pm Q' = (v' \pm v)G, \quad (21)$$

where v and v' are, recall, obtained from the x-coordinates of Q and Q' . Ignoring for the moment how one obtains M and M' corresponding to Q and Q' , one attack strategy is to find Q and Q' satisfying (21).

One strategy to solve (21) is to pick random Q and Q' and to see if the desired equation holds. On the heuristic the discrete logs and x-coordinate are essentially random and independent of one another, then each guess would have probability of about 2^{-m} of success. The attack would therefore take 2^m steps, which is much slower than the birthday attack.

6.1.4 Truncation attack

An attacker may try to find a collision in which $R \neq \pm R'$, but $H = H'$. Such pairs (R, R') certainly exist, are easy to find. Given such a pair, one has to find the corresponding message (M, M') . One can easily pick one of the messages first and then find the other with exhaustive search, at cost of about 2^m steps. Again this is much worse than the birthday attack.

6.2 Second Preimage Resistance

Second preimage resistance is defined relative to some message space with a given probability distribution. A message M is selected from this message space according to the given probability distribution. The adversary then must find a second message M' such that $M' \neq M$ with the same hash value, that is, $H(M) = H(M')$. Note that it is not required that M' belongs to the same message space.

If the hash has second preimage resistance of n bits, then finding this M' should require work of about 2^n operations.

6.2.1 Exhaustive search for second preimages

In the exhaustive search algorithm, a message search space is fixed, not necessarily the same space as the one for M . The search space should be larger than 2^n , such as the set of all bit strings of length $2n$. Then random messages M' are selected from the search space until one has $H(M') = H(M)$.

Under the heuristic that the hashes of random messages from the search space are nearly uniformly distributed in the hash output space, this algorithm is expect to require about 2^{n-1} hash evaluations.

This heuristic is generally reasonable, unless the hash fails in some respect to be pseudorandom, and also the distribution for M is such that the hash of M is among the less likely values of hashes of the values in the search space. In this case, the algorithm above may require more than 2^{n-1} hash evaluations.

6.2.2 Wagner’s Generalized Birthday Attack

Wagner [12, Theorem 3] describes an application of his generalized birthday attack in which many discrete logarithms are solved in order to decompose a point into any group into a sum of points each from a given list. Essentially for $t \geq 1$, then this attack takes about

$$2^{t+m(\frac{1}{2} + \frac{1}{1+t})} \tag{22}$$

elliptic curve group operations. Suppose that ECOH did not include the XOR checksum. Then, this attack could be used to find a second preimage of the weakened ECOH hash of some given message. This would be an internal collision in the sense that the intermediate Q values of the two message would be identical. Setting $t = 2$, this would be a 235, 340, or 475 bit attack for the curves B-283, B-409 and B-571. Therefore, if this attack could be launched, then it would be below NIST’s required threshold, specifically, by 13, 36 and 28 bits, respectively.

Wagner’s attack, however, does not include a method to handle any dependency between the blocks. In particular, the checksum in final the ECOH point contributing to Q , would create an unusual dependency between the discrete logs of the points P_i . Perhaps Wagner’s attack can be extended to handle relations from the checksum in the point representation space and in the discrete log space. This may have an associated cost.

6.2.3 Semaev summation polynomial second preimages

Solving Semaev summation polynomials can also be used to obtain second preimages, similarly to what was done with collisions.

6.3 Preimage Resistance

Preimage resistance of a hash function refers to difficulty of the problem of taking a random bit string H and finding a message M equals H .

SHA-3 submissions are expected to have n bits of preimage resistance. This means that an algorithm that computes preimage with significant probability should tak about 2^n operations. It would also be true if no preimage finding algorithm has probability of success more than about 2^{-n} . More generally, if an algorithm requires t bit operations and has probability of success ϵ , then t/ϵ should be approximately 2^n .

An n -bit hash function with constant output has n bits of preimage resistance under this definition. Arguably this is counterintuitive, and a good reason to consider an alternative definition of preimage resistance, as in §6.4, for which the constant function is not preimage resistant. However, the security of certain digital signature schemes employing hash functions, such as ECDSA, require the definition above of preimage resistance. This is a strong reason not to abandon this definition, despite its trivial instantiation. In this light, the explanation for the counterintuitiveness of the constant function being preimage resistant is that our intuition says it is insecure mainly because it is not collision resistant, not because it is not preimage resistant. In other words, collision resistance and preimage resistance are somewhat independent notions.¹

We consider the following two preimage attacks on ECOH.

¹Indeed, take any collision resistant hash and construct from it another that remains collision resistant but fails to be preimage resistant. Therefore, these notions are effectively independent, in that neither implies the other.

6.3.1 Exhaustive preimage search

Given a bit string H of length n , do the following to attempt to find a preimage. Fix a finite message space, such as all message some fixed length, such as $4n$. Choose a message M uniformly at random in the message space and compute its hash. If the hash of M equals H , then stop and output M as the preimage. If the hash is not H , repeat until done.

Under the heuristic that the hashes of random messages uniformly distributed in the message space are nearly uniformly distributed bit strings of length n , then we can conclude the following. If t is the time the algorithm is allowed to run, and ϵ is the probability that algorithm finds a preimage, then $t/\epsilon \approx 2^n$.

This is a generic algorithm to find preimages in a hash function. Its runtime complexity analysis depends on a heuristic of the randomness of the output of the hash. We conjecture that this heuristic is reasonable for ECOH.

More generally, a milder heuristic gives the same estimate of success. The milder heuristic is that message space is such that for t random messages one expects approximately t different outputs of the hash on these outputs.

6.3.2 Inverting ECOH

Another strategy is to find a preimage of the bit string H of length n is the following. The attack has two stages. The first stage is the following.

- For each bit string B of length $m - n - 1$ do the following steps.
- Form the bit string $X' = B||H$ of length $m - 1$.
- Append to X' a single bit to obtain a bit string X such that X represents the x-coordinate of a point $\langle G \rangle$, if it exists.
- For each of two possible points R with x-coordinate represents by X do the following.
- For each point Q in $\langle G \rangle$ do the following.
- Compute $R' = Q + \lfloor x(Q)/2 \rfloor G$. If $R' = R$, stop and output Q .

Then second stage is to find M such that the ECOH computation on M yields Q as the intermediate total of elliptic curve points. This could be done by exhaustive search, or perhaps by using Semaev summation polynomials.

Regardless of the method used in the second stage, under the heuristic assumptions $R' = R$ with probability approximately 2^{-m} , the first stage would be expected to take about $2^{m-n-1+m-1}$ iterations to find a collision. Because $m > n + 1$, the first stage takes more than 2^n operations.

6.4 Relative Preimage Resistance

As noted above, it may be worthwhile to consider another type of preimage resistance, which we will call *relative preimage resistance* and define below. It is defined relative to a known message space with a probability distribution. The adversary is given the hash of message selected at random from this message space according to the given probability distribution. The adversary must then find some message whose hash equals the hash of the unknown message.

Note that this definition is the same as second preimage resistance except that (a) the adversary is given the hash of first message rather than the first message, and (b) the adversary wins even if it finds the first message.

Under the heuristic that the hash function outputs are random, then there exists some easily sampled probability distribution of messages such that images of these messages give rise to every possible bit string in the range of the hash function with approximately equal probability. With respect to these general message spaces, the relative preimage resistance is equivalent to the preimage resistance.

Relative preimage resistance differs from standard preimage resistance primarily for specialized message spaces. Relative preimage resistance can be defined for small message spaces. If a message space has size 2^s , then an adversary can find, by exhaustive search (see below) a preimage with work of about 2^s steps. Therefore, the bit of security defining relative preimage resistance are relative to the message space, and in particular, its size in bits.

6.4.1 Exhaustive search for preimages

Given a hash value, one can exhaustively search for random messages from the message space one takes on the desired hash value.

The adversary should guess the message is order of highest probability in the distribution, if able. With one guess the adversary has probability of success 2^{-h} , where h is the min-entropy of the probability distribution. For repeated guessing, the number of guesses needed to be quite certain to find a preimage depends on the probability distribution, and the way the hash produces collisions on that probability distribution. On the heuristic the hash outputs are random and independent of the inputs, then this attack should require about at most 2^n guesses and at least about 2^h guesses, especially if $h \ll n$.

6.4.2 Single block inversion

This is a specific attack one may try against ECOH. It is a relative preimage attack whose probability distribution has entropy $blen$ bits, which is less than n bits. The aim of the attack of the attack is to be faster than exhaustive search, which one heuristically would expect to take 2^{blen} guesses.

In this attack all but one block is fixed and known to the adversary. Suppose that N_j is the secret block. The adversary is given the hash H and must determine the unknown message block. The adversary loops over all points R in $\langle G \rangle$ that truncate down to H . This loop has approximately 2^{m-n} steps.

If $\beta = 0$, then the attacker solves a discrete log to get v and thus Q . This has cost of about $2^{m/2}$ elliptic curve group operations. If $\beta = 1$, then the attacker does an exhaustive search for Q , taking 2^m elliptic curve group operations. (I do not know of a $2^{m/2}$ step algorithm to solve this).

Then adversary knows all the term points except P_j and P_k . These can be subtracted from Q to obtain $P_j + P_k = (x, y)$. Suppose that $\omega = 0$, then $x_k = x_j + \delta$ for some $\delta \in \mathbb{F}_{2^m}$ that the attacker knows, because of the way the tail indexed block is computed as checksum. So the attacker needs to solve for x_j in the equation:

$$(x_j, y_j) + (x_j + \delta, y_k) = (x, y) \tag{23}$$

The attacker does not know y_j and y_k , but here perhaps Semaev summation polynomials will again be useful. The result is a single variable polynomial equation in x_j , which the adversary can easily solve over \mathbb{F}_{2^m} .

Therefore for $\beta = 0$, the cost of the attack is approximately $2^{m-n+m/2}$, which is larger than exhaustive search 2^{blen} provide $(3/2)m > n+blen$. This is the case for the recommended parameters.

For $\beta = 1$, an extra margin of safety is provided.

For $\omega = 1$, then the checksum block is an obfuscated function of x_j , and the attack does not work. Solving for x_j can be done by exhaustive search.

6.5 Pseudorandomness

The output of ECOH is a truncated elliptic curve point. An algorithm [5, 8] to distinguish ECOH output from random bit strings is as follows. Compute all possible values of the x-coordinate that could have been truncated to obtain the ECOH hash. Test each of these for validity of being a point on the elliptic curve. If the proportion of valid points is higher than half, then guess that it is an ECOH hash.

Therefore, the output of ECOH is distinguishable from a random bit string of the same length, with work proportional to the number of bits truncated, and a mild success rate. This attack can be iterated over many ECOH outputs to obtain a higher success rate.

In many cryptographic applications, such distinguishability is not known to lead to any attacks. Indeed, users of ECOH would not attempt to hide the fact that they are using ECOH. In this attack, the adversary mainly learns the user is using ECOH, information that is public already. Furthermore, there is evidence [3, Theorem 1] that using x-coordinates as elliptic curve private keys does not compromised security. In the principal submitter's opinion, although this is technically a distinguishing attack, nothing practical can be gained from it.

Nonetheless, note that generalized ECOH provides an option $\alpha = 1$, which seems to thwart this attack. This option uses addition modulo 2^m .

6.6 Side Channel Resistance

It seems intrinsic to the design of ECOH that any implementation must inevitably run in time that varies with the message being hashed. Specifically, a counter must be incremented until a point on the curve is found. This creates a potential for a side channel, which may be harmful if the message being hashed is a secret, especially a low-entropy secret.

As an implementation-level countermeasure, the implementation can introduce dummy operations to mask the message-dependent variability, or even attempt to make things run usually in the same amount of time, with the rarer exceptions masked by some other means. The blocks can also be processed in randomized order.

As a protocol-level countermeasure, one can try to avoid hashing secrets, or at least low-entropy secrets.

6.7 Denial of Service Resistance

The following strategy could be used as a denial of service attack against ECOH. An adversary could search for a message M with one or more blocks requiring a high value of the counter C_i . Hashing the message will therefore take longer, because the hasher must search longer.

We now estimate the efficiency of this strategy. First note, that for submitted version of ECOH only odd values of the counter c need be considered. For any given message block O_i , the probability the number t of odd c that must be tried is 2^{-t} . Note $c = 2t - 1$. Furthermore, the attacker has no efficient way of finding messages with larger t . That is, the attacker must use trial and error, selecting random messages until one with high t is found.

Under this heuristic, an attacker would have try about 2^t values of message block until is found requiring t tries. In other words, the attacker must use exponentially more work to force the users to linearly more work. From this perspective, the attack is rather mild.

7 Applications

Currently, SHA-1 and SHA-2 have four main applications in NIST standards:

- computing message digests in digital signature schemes, e.g. ECDSA,
- message authentication, i.e. HMAC,
- key derivation for key agreement, e.g. Concatentation KDF,
- pseudorandom number generation, e.g. Hash-based DRBG.

The following sections discuss how to apply ECOH in the five cases, directly, and in some cases, in an alternative manner.

7.1 Digital Signatures

When using ECOH in a digital signature scheme, such as ECDSA, one can just take the output of ECOH and use it in the same manner as one would use SHA-2.

When using ECDSA with the same curve as used in ECOH, the elliptic curve implementation can be common to both algorithms, thereby providing a slightly smaller implementation.

7.2 Message Authentication

The HMAC construction was specifically intended for a hash function using Merkle-Damgard iteration, so using ECOH in HMAC would not be what HMAC was intended for. Nevertheless, we identified no attacks on using ECOH as the hash in the HMAC construction.

7.3 Pseudorandom Number Generation

If ECOH were used as part of the Hash DRBG to build a pseudorandom number generator, one would arrive at something similar in some respects to the Dual EC DRBG.

As noted earlier, an adversary willing to do some work can distinguish ECOH outputs from random bit strings. The same attack applies to the Dual EC DRBG. We submit, however, that this is the only effective information that the adversary learns: whether the bit string is generated as an ECOH output or not. In most applications of pseudorandom number generators, this would not be an issue.

For the purposes of generating random numbers to be used as elliptic curve keys [3] provides some heuristic arguments that building private keys from x-coordinates of random points is not problematic.

7.4 Key Derivation

A key derivation function (KDF) is subject to most of the concerns of a pseudorandom number generator, as outlined in §7.3.

The output of a key derivation function is sometimes used as key stream. For example, ECIES has this as an option. Suppose that ECIES is implemented with this option, and with ECOH serving the role of the hash function inside a KDF construction. If an adversary knows that the

plaintext has one of two candidate values, then given an ECIES ciphertext, the adversary can subtract the plaintext to get two candidate key streams. These streams can be tested for consistency with being produced by ECOH. Likely, one will be consistent and the other not, with the consistent key stream coming from the plaintext that was encrypted. Note the ECIES is not included as a NIST standard.

7.5 Randomized Hashing

We identified no issues with using random hashing that are specific to ECOH.

References

- [1] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, number 1233 in Lecture Notes in Computer Science, pages 163–192. Springer, 1997. Full version: <http://eprint.iacr.org/1997/001>.
- [2] D. R. L. Brown. The encrypted elliptic curve hash. ePrint 2008/012, International Association for Cryptologic Research, 2004. <http://eprint.iacr.org/2008/012>.
- [3] D. R. L. Brown and K. Gjøsteen. A security analysis of the NIST SP 800-90 elliptic curve random number generator. In A. J. Menezes, editor, *Advances in Cryptology — CRYPTO 2007*, Lecture Notes in Computer Science 4622, pages 466–481. International Association for Cryptologic Research, Springer, Aug. 2007.
- [4] P. Gaudry. Index calculus for abelian varieties and the elliptic curve discrete logarithm problem. *Journal of Symbolic Computation*, To appear. Preprint: <http://eprint.iacr.org/2004/073>.
- [5] K. Gjøsteen. Comments on Dual-EC-DRBG/NIST SP 800-90. <http://www.math.ntnu.no/~kristiag/drafts/dual-ec-drbg-comments.pdf>, Mar. 2006.
- [6] S. Gueron and M. Kounavis. A technique for accelerating characteristic 2 elliptic curve cryptography. In *Fifth International Conference on Information Technology: New Generations*, pages 265–272. IEEE Computer Society, 2008.
- [7] D. Hankerson, A. J. Menezes, and S. A. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
- [8] B. Schoenmakers and A. Sidorenko. Cryptanalysis of the dual elliptic curve pseudorandom generator. ePrint 2006/190, International Association for Cryptologic Research, 2006. <http://eprint.iacr.org/2006/190>.
- [9] I. Semaev. Evaluation of discrete logarithms in a group of p -torsion points of an elliptic curve in characteristic p . *Mathematics of Computation*, 67(221):353–356, Jan. 1998.
- [10] I. Semaev. Summation polynomials and the discrete logarithm problem on elliptic curves. ePrint 2004/031, International Association for Cryptologic Research, 2004. <http://eprint.iacr.org/2004/031>.
- [11] Standards for Efficient Cryptography Group. *SEC 1: Elliptic Curve Cryptography*, Sept. 2000. Version 1.0. Available at <http://www.secg.org>.
- [12] D. Wagner. A generalized birthday problem. In M. Yung, editor, *Advances in Cryptology — CRYPTO 2002*, Lecture Notes in Computer Science 2442, pages 288–303. International Association for Cryptologic Research, Springer, Aug. 2002. Longer version: <http://www.cs.berkeley.edu/~daw/papers/genbday-long.ps>.

A Heuristic Security Arguments

In this section, heuristic arguments for the security of ECOH are given.

A.1 Collision Resistance

Wagner [12] defined the k -sum problem. In the context of elliptic curve groups, this problem may be expressed as follows.

Problem A.1. *Given a point Y and k lists of random points L_1, \dots, L_k , find $P_i \in L_i$ such that $Y = P_1 + \dots + P_k$.*

Wagner [12] proves a theorem, attributed to Dai, that if the k -sum problem can be solved in time t , then the discrete logarithm problem can be solved time almost t .

Now consider a collision (M, M') in ECOH, where the associated intermediate values are related $Q' = \pm Q$. We call such a collision an internal collision. Then

$$P_0 + P_1 + \dots + P_k \mp (P'_0 + P'_1 + \dots + P'_{k'}) = \mathcal{O} \tag{24}$$

The problem of finding an internal ECOH problem is similar to the $(k + k' + 2)$ -sum problem, with the exception that the lists L_1 involved are not necessarily of random point but of points whose x-coordinates have a particular structure.

If one adopts the loose heuristic that the structure in x-coordinate amounts to effectively a random list of points, then finding an internal ECOH collision may be viewed as an instance of the k -sum problem. Under this heuristic, it is at least as hard as the discrete log problem, because of Wagner and Dai's theorem.

If an ECOH collision is not a internal collision, then it must be an external collision.

A.2 Second Preimage Resistance

A second preimage gives also collision, so we get at least the same amount of security as provided by collision resistance.

A.3 Preimage Resistance

The shifted scalar multiply function in ECOH takes a point Q , maps its x-coordinate to an integer v and returns the point $R = Q + vG$. We generalize this slightly. Let E be the elliptic curve group in question. A function $f : E \rightarrow \mathbb{Z}$ is called a conversion function. We will be mostly interested in the conversion function used by ECOH which maps Q to v above. Given a conversion function f , we can define function $\phi : E \times E \rightarrow E$ by

$$\phi(S, Q) = Q + f(Q)S. \tag{25}$$

We will also write this as $\phi_S(Q)$. This function is the generalized shifted scalar multiply. We formulate the following problem.

Problem A.2. *Given S and $\phi_S(Q)$, find Q .*

We call this problem the shifted log problem. We conjecture as follows.

Conjecture A.1. *If the conversion function f is almost injective, then solving the shifted log problem cost about N , the order of G , group elliptic curve group operations.*

Compared to the discrete log problem, which takes about \sqrt{N} operations to solve, this problem is conjecturally much harder.

An immediate objection to this conjecture is that the shifted log problem may in fact be easier than the discrete log problem, requiring much fewer than \sqrt{N} group operations. We observe, however, that an algorithm to solve the shifted log problem could be used to forge an ECDSA signature.

Theorem A.1. *Let f be the conversion function used in ECDSA to determine the r component of the signature from an elliptic curve point. Let A be an algorithm that solves the associated shifted log problem where f . Then an algorithm F can be built that calls A once can be used to forge an ECDSA signature. The forger is as efficient as A plus a very small extra cost.*

Proof. Algorithm F is given message M and public key P with respect to which to forge a signature. Algorithm F has access to a subroutine A such that $A(T, S)$ inverts ϕ_S on T . In other words, A is such that

$$T = A(T, S) + f(A(T, S))S. \tag{26}$$

We now describe algorithm F .

Algorithm F computes $e = H(M)$ where H is the hash used in ECDSA, including the step of conversion to an integer. Recall that a pair of integers (r, s) is a valid signature for message M and public key P if and only if:

$$r = f((e/s)G + (r/s)P) \tag{27}$$

where the divisions above are done modulo N . Observing that (27) holds if

$$R = (e/s)G + (f(R)/s)P \tag{28}$$

and $r = f(R)$, the following strategy can be used for F .

First F chooses random s and computes $S = -(1/s)P$ and $T = (e/s)G$. Then F calls A to compute

$$r = f(A(T, S)), \tag{29}$$

and output (r, s) as an ECDSA signature. To see that this is a valid signature, write $R = A(T, S)$. By (26),

$$(e/s)G = R - (f(R)/s)P, \tag{30}$$

which clearly implies (28). □

Therefore, the shifted log problem associated with the ECDSA conversion function is at least as hard as being able to forge an ECDSA signature on any message given only the public key.

Assuming the security of ECDSA, this only assures that shifted log problem is about as hard as \sqrt{N} elliptic curve group operations.

We give a heuristic argument in the generic group model suggesting that the shifted log problem takes about N group operations, not just \sqrt{N} group operations. We assume that f is almost bijective.

Theorem A.2. *An algorithm A working in the generic group that finds a point that is the shifted log of the first two independent points, that makes q queries has a probability of success of approximately q/N .*

Proof. The desired result is $T = R + f(R)S$, where T and S are first two independent points in the generic group session.

If R is also independent in the session, with public representation chosen by A , then its private value will be randomly by the generic group oracle, and the probability of the desired is exactly $1/N$, the probability that $R = T - f(R)S$ holds in the private space.

If R is a dependent point in the generic group session, then its private value is determined by the generic group oracle before its public representation is chosen by the generic group oracle. Because f is almost invertible, and the representation of R is chosen uniformly at random, the distribution of $f(R)$ is almost uniformly at random. The probability of the desired result is therefore only a little more than $1/N$, depending on the degree of departure of f from being bijective. \square

Now suppose that adversary B could find preimages in ECOH. Then B can be used to solve the shifted log problem, with a somewhat lowered probability of success, as follows. Build an algorithm A to solve the shifted log problem by calling B as subroutine. First A truncates its input and gives to B . When B returns its preimage M , we have a heuristic argument that there is some probability that the final point obtained in the computation of the hash of M before truncation is the same as the point for which A is trying to solve the shifted log. For example, in ECOH-256, this probability is 2^{-27} . In this case, the value of Q in the computation of the ECOH hash of M is the desired shifted log.

A.4 Pseudorandomness

As noted earlier, ECOH outputs are slightly distinguishable from random bit strings. Heuristic arguments suggest, however, that otherwise ECOH outputs are not generally distinguishable. For example, in the generic group model of an elliptic curve, the points map randomly into a space. In this case points in the space may be slightly distinguishable from random bit strings.

B Examples

In this section, some examples of ECOH are given.

B.1 Some Examples

Tables 10–13 give the output of ECOH on six different bit strings. The output is given in hexadecimal notation.

M	$H(M)$
	C86A6DBA2030177D298A1104EF3D575466D6B3DDF306F94EBE96CFA4
0	19725F2CF6DBAE5C80FECE71FE30DB287BF1504BB3276EC1FA7A9BD8
11	1D6F757C15908D5FF669C58AB2940CBF8F707F2B42BF9E0BF832761F
1001100	D704100A2F928565CAD79E42761B3E84EE63C9020F4D1B36ED2634D0
11001100	60102BBF4D997BE46C754A6367C0FAD8C55207D6CDE0212891D0C792
100110000	9BC7F964121BF2B70DB8CC66C90C06599665F4978AD15EBBB40A680D

Table 10: ECOH-224 examples

M	$H(M)$
	AC160817C86A6DBA2030177D298A1104EF3D575466D6B3DDF306F94EBE96CFA4
0	4A28011C19725F2CF6DBAE5C80FECE71FE30DB287BF1504BB3276EC1FA7A9BD8
11	33E772D61D6F757C15908D5FF669C58AB2940CBF8F707F2B42BF9E0BF832761F
1001100	5A297E85D704100A2F928565CAD79E42761B3E84EE63C9020F4D1B36ED2634D0
11001100	7D1EEF7860102BBF4D997BE46C754A6367C0FAD8C55207D6CDE0212891D0C792
100110000	FE46B2829BC7F964121BF2B70DB8CC66C90C06599665F4978AD15EBBB40A680D

Table 11: ECOH-256 examples

B.2 Examples of Intermediate Values

B.2.1 ECOH224 Example (one-block Message)

Let the message M be the 120-bit ASCII string “abcdefghijklmno” which is equivalent to the hexadecimal string:

$$6162636465666768696A6B6C6D6E6F$$

The message is padded by appending a one bit followed by 7 zero bits, resulting in the 128-bit value N , which in hexadecimal is:

$$6162636465666768696A6B6C6D6E6F80$$

We can parse N into one 128-bit blocks, N_0 in hexadecimal format

$$N_0 = 6162636465666768696A6B6C6D6E6F80$$

M	$H(M)$
	BD946B4998EBC6C45F55F4E575B1A5E167803F4995125BFF 881C27359351377BD323B7CE42D62C1C8173D465C554DA34
0	E1B847F4AEOE878BDB8A9D07063267515EA7FECF1CD8C4E2 1C8649E9D96A7764DOBBA7F1EE3B5601129E71BC0368FE96
11	48318BDB401E092273FBF3215680D99B5B0A1DA89E4F7619 C2FFDA3F6BBDF1795871B7E683437903757F229E4B30D9D5
1001100	594FC40AE04F3803359172BDE3C2148F96ACCD46EE7AA416 C3905FD4FA4C65CBC8F36C73ABD61FEC4F849C0F29BC351A
11001100	386B7DD30F11DD5084FD0ECC0E585C24E0EE8D9D34DF4D06 2372571BF89C680CABEC1576C72B5EBC438369C3BAB1B4FB
100110000	EAA441ADB14DB54049D2A43AB7F33CBD7C0EC57664D6F8A9 2D4DBAD96842EE2570E84C14213CC00998A97426CC695493

Table 12: ECOH-384 examples

M	$H(M)$
	757AB7847F7A720FA639B6E8CBA29EB135C2A9586DC8BC8B99CD5444AA69113D 5112147ED12C1E7BB8C9FDEDFC0BA560312C6E15E40B901A53881F3CDCFE4156
0	B366D9054215EE6418B5E48B9633B8F5B34C0CE8E7316CEB8C6EC1DC941C47E3 555357293701864470F8B4F1BD69D6A0484D748363965B184418E4EE2AE01DCD
11	D2636EF1FFF65E1431AE962E1E6D1355314F08FDA1CBC3392CE59846A6A3457D BOD635F5921163BD5A8CE089B964AB12D2A42554DF74F904A9075FD15CE2220C
1001100	946F6B1DD663364864CBAC497BC1ACF4BF6E87C36A1AD0F04310D90066178BDA BFE1C6A28C872869CA8D7B496806068EA6E7513AB1A854D24C579186D623193F
11001100	3E305821E741E4CE19DF34A1EA2D5BEFB8DB4E9A0C95586C4055EAA51B02D7BA A56631E8F61A33762C4EAA86AA9EDDFBFE179398F1D93BAABE8D656852CD820D
100110000	6D5445FF7137D57F38601DDAE02AD4FD5763CCC49E73371993B43907D2A37BF0 AA49C5734BE376C335AA97AF36DCFC638E665F0DB112534B63B3D4C6FB6ADEEC

Table 13: ECOH-512 examples

Next, O_0 is formed by appending the 64-bit counter values for 0, to N_0 to get O_0 , in hexadecimal as

$$O_0 = 6162636465666768\ 696A6B6C6D6E6F80\ 0000000000000000$$

We form $O_1 = (\bigoplus N_i) || I_{mlen}$, where $mlen$ is the 64-bit representation of the bit length of M, so that $I_{mlen} = 00000000000000078$. In totality we get O_1 in hexadecimal format

$$O_1 = 6162636465666768\ 696A6B6C6D6E6F80\ 0000000000000078$$

For O_0 we start with a counter value $c_0 = 1$ in hexadecimal format

$$0000000000000000\ 0000000000000000\ 0000000000000000\ 0000000000000000\ 0000000000000001$$

This when combined with $0^{27} || O_0 || 0^{64} \oplus C_0$ gives us X'_0 , in hexadecimal format

$$X'_0 = 0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F80\ 0000000000000000\ 0000000000000001$$

which decompresses to a point on the curve secp283r1 when $C_0 = 3$, or when X_0 in hexadecimal format is

$$X_0 = 0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F80\ 0000000000000000\ 0000000000000003$$

This decompresses using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_0 , $y_p = 0$ gives us the point P_0

$$P_0 = (0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F80\ 0000000000000000\ 0000000000000003, \\ 00000000050376A6\ A334368C305B46E0\ 802EF2781CFCC550\ D12E2983032F745D\ 25EA3D8A50D2F7BD)$$

Which is added to Q , initialized as the point at infinity to become,

$$Q_0 = (0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F80\ 0000000000000000\ 0000000000000003, \\ 00000000050376A6\ A334368C305B46E0\ 802EF2781CFCC550\ D12E2983032F745D\ 25EA3D8A50D2F7BD).$$

Now with O_1 we construct $X'_1 = 0^{27} || O_1 || 0^{64} \oplus C_1$, where C_1 starts = 1

$$X_1 = 0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F80\ 0000000000000078\ 0000000000000001$$

which decompresses using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_1 , $y_p = 0$, gives us the point P_1

$$P_1 = (0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F80\ 0000000000000078\ 0000000000000001, \\ 0000000002B39C01\ 5A3546A831E3A973\ 230EA15EA17C472B\ 09A7207588C53B68\ 023A68A6A22C9262)$$

Which is added to Q to become,

$$Q_1 = (0000000001493FB2 CA26C851E35B29AA 085E61DD2E8C93B3 2040B7430A102D71 B5C50CB6682A5063 , \\ 00000000001587CE 606D9A9EE15712A9 E2244AB827B92294 4E7D3D310FAEFCA3 4B1B5A681D39EA4D) .$$

We can interpret the x-coordinate as an multi-precision integer and divide by two taking the floor of the result, or simply dropping the remainder to get the following in hexadecimal format

$$\lfloor x(Q_1)/2 \rfloor = 0000000000A49FD9 65136428F1AD94D5 042F30EE974649D9 \\ 90205BA1850816B8 DAE2865B34152831$$

Using this value as an integer we can multiply the base point of the curve to get,

$$\lfloor x(Q_1)/2 \rfloor G = (00000000038372D3 05CF94C4D2B2D1AB A4D9A1FA975F149F \\ 982C013BF86C4692 905433FFD2ED6EF3 , \\ 0000000005501C6A DA6D0E07D2B0A2C7 0316D973D90CC645 \\ F63AB923DFBEC6A6 53E048D1513BA111) .$$

Adding that value back with Q_1 above we get.

$$Q_1 + \lfloor x(Q_1)/2 \rfloor G = (0000000005157E42 95D6FF0C5B3D9D00 FA1B0D76A04ADBF9 \\ 0252C748B2C46850 BDCF32AFBF9C5AAB, \\ 0000000002A28F1B 83177FAC4824222D \\ 412B691FA51524DF 126D535AFF08BB73 9A9F304A236397AF) .$$

And again interpreting the x-coordinate of the result as a multiprecision integer and dividing by 2 discarding the remainder to get

$$\lfloor x(Q + \lfloor x(Q_1)/2 \rfloor G)/2 \rfloor = 00000000028ABF21 4AEB7F862D9ECE80 7D0D86BB50256DFC \\ 812963A459623428 5EE79957DFCE2D55 .$$

From this result the rightmost 224-bits are extracted to get the message digest.

$$H = 2D9ECE807D0D86BB50256DFC812963A4596234285EE79957DFCE2D55$$

B.2.2 ECOH224 Example (two-block Message)

Let the message M be the 248-bit ASCII string “abcdefghijklmnopqrstuvwxyabcde” which is equivalent to the hexadecimal string:

$$6162636465666768696A6B6C6D6E6F707172737475767778797A6162636465$$

The message is padded by appending a one bit followed by 7 zero bits, resulting in the 256-bit value N, in hexadecimal string:

$$N_0 = 6162636465666768696A6B6C6D6E6F70$$

$$N_1 = 7172737475767778797A616263646580$$

O_0 is formed by prepending the 64-bit counter values for 0, 0000000000000000 to N_0 to get O_0 in hexadecimal format

$$O_0 = 6162636465666768 696A6B6C6D6E6F70 0000000000000000$$

O_1 is formed by prepending the 64-bit counter values for 1, 0000000000000001 to N_1 to get O_1 in hexadecimal format

$$O_1 = 7172737475767778 797A616263646580 0000000000000001$$

We form $O_2 = (\bigoplus N_i) || I_{mlen}$, where $mlen$ is the 64-bit representation of the bit length of M , $I_{mlen} = 0000000000000000F8$. In totality we get O_2 in hexadecimal format

$$1010101010101010 10100A0E0E0A0AF0 00000000000000F8$$

This when combined with $0^{27} || O_0 || 0^{64} \oplus C_0$ gives us X'_0 , in hexadecimal format

$$X'_0 = 0000000000000000 6162636465666768 696A6B6C6D6E6F70 0000000000000000 0000000000000001$$

which decompresses to a point on the curve sect283r1 when $C_0 = 7$, or when X_0 in hexadecimal format is

$$X_0 = 0000000000000000 6162636465666768 696A6B6C6D6E6F70 0000000000000000 0000000000000007$$

This decompresses using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_0 , $y_p = 0$ gives us the point P_0

$$P_0 = (0000000000000000 6162636465666768 696A6B6C6D6E6F70 0000000000000000 0000000000000007 , \\ 0000000005D2D2B2 6B08807376F86FC0 31616229DACDD074 EDD97B55E07F882 753192E074303C35)$$

Which is added to Q , initialized as the point at infinity to become,

$$Q_0 = (0000000000000000 6162636465666768 696A6B6C6D6E6F70 0000000000000000 0000000000000007 , \\ 0000000005D2D2B2 6B08807376F86FC0 31616229DACDD074 EDD97B55E07F882 753192E074303C35)$$

Now with O_1 we construct $X'_1 = 0^{27} || O_1 || 0^{64} \oplus C_1$, where C_1 starts = 1

$$X_1 = 0000000000000000 7172737475767778 797A616263646580 0000000000000001 0000000000000001$$

which decompresses directly to a point on the curve sect283r1 P_1

$$P_1 = (0000000000000000 7172737475767778 797A616263646580 0000000000000001 0000000000000001 , \\ 000000003554E1B E469855991ACD3A7 684E71E8DE83B716 153E2A4D9F7555E7 5C26B1C1DDD4F660)$$

Which is added to Q to become,

$$Q_1 = (000000002519C79 ABB30532C0277212 FA7527B2EF45BEAE CC90C6569765D7CC 004E63CF26E51764 , \\ 00000000034745F9 D4060AFEF1B0D247 6A42DBFE6FDD9160 12D17A335769B05B 49974C557F48FF18)$$

We can now construct the final block $X'_2 = 0^{27} \|O_2\|^{0^{64}} \oplus C_2$ with $C_2 = 1$, gives us X'_2 , in hexadecimal format

$$X'_2 = 0000000000000000 1010101010101010 10100A0E0E0A0AF0 00000000000000F8 0000000000000003$$

Which decompresses with $C_2 = 3$ to give us the point P_2 ,

$$P_2 = (0000000000000000 1010101010101010 10100A0E0E0A0AF0 00000000000000F8 0000000000000003 , \\ 0000000005101D6 63A2FC2E0F1806E7 999BC9FAE01D787A 943C2574136DAC9E 2D0913E2E39F15AE)$$

Which is added to Q to become,

$$Q_2 = (00000000207FDCC 60A06C5113C7D974 EDCF68C574606DDA \\ E011631B29590AD8 8F48C21504095B10 , \\ 0000000005CC85CD 6153FA9C2D1FECD9 D83746919C4D67AC \\ A12E9BF84AD8AD83 516402F90E9ABCC9)$$

We can interpret the x-coordinate as an multi-precision integer and divide by two taking the floor of the result, or simply dropping the remainder to get the following in hexadecimal format

$$\lfloor x(Q_2)/2 \rfloor = 000000000103FEE6 3050362889E3ECBA 76E7B462BA3036ED \\ 7008B18D94AC856C 47A4610A8204AD88$$

Using this value as an integer we can multiply the base point of the curve to get,

$$\lfloor x(Q_2)/2 \rfloor G = (0000000002703FE2 871E5F59C4E80175 8A7C3F5964B213E3 \\ D111BCFC280B33DB 7FE445CB1C699729 , \\ 000000000D12956 76446E7693FFC04C BFCA6A94206F3C34 \\ BC4305D537C1A8CC 4B2636C2D5D6C8D4)$$

Adding that value back with Q_2 above we get.

$$Q_2 + \lfloor x(Q_2)/2 \rfloor G = (00000000060883AD E578F697250191F3 0EB2F4094732BB66 \\ 7D7D90AEB0C6AB30 EC9AC49D96EB54C8 , \\ 000000000100886D 599FEB046E1F1968 2446D3D2870CB271 \\ 2B6C4432B0D42044 3F6CDA67C42F2E2D)$$

And again interpreting the x-coordinate of the result as a multiprecision integer and dividing by 2 discarding the remainder to get

$$\lfloor x(Q_2 + \lfloor x(Q_2)/2 \rfloor G)/2 \rfloor = 00000000030441D6 F2BC7B4B9280C8F9 87597A04A3995DB3 \\ 3EBEC85758635598 764D624ECB75AA64$$

From this result the rightmost 224-bits are extracted to get the message digest.

$$H = 9280C8F987597A04A3995DB33EBEC85758635598764D624ECB75AA64$$

B.2.3 ECOH256 Example (one-block Message)

Let the message M be the 120-bit ASCII string “abcdefghijklmno” which is equivalent to the hexadecimal string:

$$6162636465666768696A6B6C6D6E6F$$

The message is padded by appending a one bit followed by 7 zero bits, resulting in the 128-bit value N, the hexadecimal string:

$$6162636465666768696A6B6C6D6E6F80$$

We can parse N into one 128-bit blocks, N_0 in hexadecimal format

$$N_0 = 6162636465666768696A6B6C6D6E6F80$$

O_0 is formed by prepending the 64-bit counter values for 0, 0000000000000000 to N_0 to get O_0 in hexadecimal format

$$O_0 = 6162636465666768 696A6B6C6D6E6F80 0000000000000000$$

We form $O_1 = (\bigoplus N_i) \parallel I_{mlen}$, where $mlen$ is the 64-bit representation of the bit length of M, $I_{mlen} = 0000000000000078$. In totality we get O_1 in hexadecimal format

$$O_1 = 6162636465666768 696A6B6C6D6E6F80 0000000000000078$$

For O_0 we start with a counter value $C_0 = 1$ in hexadecimal format

$$0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000001$$

This when combined with $0^{27} \parallel O_0 \parallel 0^{64} \oplus C_0$ gives us X'_0 , in hexadecimal format

$X'_0 = 0000000000000000 6162636465666768 696A6B6C6D6E6F80 0000000000000000 0000000000000001$

which decompresses to a point on the curve sect283r1 when $C_0 = 3$, or when X_0 in hexadecimal format is

$X_0 = 0000000000000000 6162636465666768 696A6B6C6D6E6F80 0000000000000000 0000000000000003$

This decompresses using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_0 , $y_p = 0$ gives us the point P_0

$P_0 = (0000000000000000 6162636465666768 696A6B6C6D6E6F80$
 $0000000000000000 0000000000000003 ,$
 $00000000050376A6 A334368C305B46E0 802EF2781CFCC550$
 $D12E2983032F745D 25EA3D8A50D2F7BD)$

Which is added to Q , initialized as the point at infinity to become,

$Q_0 = (0000000000000000 6162636465666768 696A6B6C6D6E6F80$
 $0000000000000000 0000000000000003 ,$
 $00000000050376A6 A334368C305B46E0 802EF2781CFCC550$
 $D12E2983032F745D 25EA3D8A50D2F7BD)$.

Now with O_1 we construct $X'_1 = 0^{27} \|O_1\| 0^{64} \oplus C_1$, where C_1 starts = 1

$X_1 = 0000000000000000 6162636465666768 696A6B6C6D6E6F80$
 $0000000000000078 0000000000000001$

which decompresses using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_1 , $y_p = 0$, gives us the point P_1

$P_1 = (0000000000000000 6162636465666768 696A6B6C6D6E6F80$
 $0000000000000078 0000000000000001 ,$
 $0000000002B39C01 5A3546A831E3A973 230EA15EA17C472B$
 $09A7207588C53B68 023A68A6A22C9262)$

Which is added to Q to become,

$Q_1 = (0000000001493FB2 CA26C851E35B29AA 085E61DD2E8C93B3$
 $2040B7430A102D71 B5C50CB6682A5063 ,$
 $0000000001587CE 606D9A9EE15712A9 E2244AB827B92294$
 $4E7D3D310FAEFCA3 4B1B5A681D39EA4D)$.

We can interpret the x-coordinate as an multi-precision integer and divide by two taking the floor of the result, or simply dropping the remainder to get the following in hexadecimal format

$$\lfloor x(Q_1)/2 \rfloor = 0000000000A49FD9\ 65136428F1AD94D5\ 042F30EE974649D9\ 90205BA1850816B8\ DAE2865B34152831$$

Using this value as an integer we can multiply the base point of the curve to get,

$$\lfloor x(Q_1)/2 \rfloor G = (00000000038372D3\ 05CF94C4D2B2D1AB\ A4D9A1FA975F149F\ 982C013BF86C4692\ 905433FFD2ED6EF3\ ,\ 0000000005501C6A\ DA6D0E07D2B0A2C7\ 0316D973D90CC645\ F63AB923DFBEC6A6\ 53E048D1513BA111\).$$

Adding that value back with Q_1 above we get.

$$Q_1 + \lfloor x(Q_1)/2 \rfloor G = (0000000005157E42\ 95D6FF0C5B3D9D00\ FA1B0D76A04ADBF9\ 0252C748B2C46850\ BDCF32AFBF9C5AAB\ ,\ 0000000002A28F1B\ 83177FAC4824222D\ 412B691FA51524DF\ 126D535AFF08BB73\ 9A9F304A236397AF\).$$

And again interpreting the x-coordinate of the result as a multi-precision integer and dividing by 2 discarding the remainder to get

$$\lfloor x(Q_1 + \lfloor x(Q_1)/2 \rfloor G)/2 \rfloor = 00000000028ABF21\ 4AEB7F862D9ECE80\ 7D0D86BB50256DFC\ 812963A459623428\ 5EE79957DFCE2D55.$$

From this result the rightmost 256-bits are extracted to get the message

$$H = 4AEB7F862D9ECE807D0D86BB50256DFC812963A4596234285EE79957DFCE2D55$$

B.2.4 ECOH256 Example (two-block Message)

Let the message M be the 248-bit ASCII string “abcdefghijklmnopqrstuvwxyabcde” which is equivalent to the hexadecimal string:

$$6162636465666768696A6B6C6D6E6F707172737475767778797A6162636465$$

The message is padded by appending a one bit followed by 7 zero bits, resulting in the 256-bit value N, in hexadecimal string:

$$N_0 = 6162636465666768696A6B6C6D6E6F70$$

$$N_1 = 7172737475767778797A616263646580$$

O_0 is formed by prepending the 64-bit counter values for 0, 0000000000000000 to N_0 to get O_0 in hexadecimal format

$$O_0 = 6162636465666768\ 696A6B6C6D6E6F70\ 0000000000000000$$

O_1 is formed by prepending the 64-bit counter values for 1, 0000000000000001 to N_1 to get O_1 in hexadecimal format

$$O_1 = 7172737475767778\ 797A616263646580\ 0000000000000001$$

We form $O_2 = (\bigoplus N_i) \parallel I_{mlen}$, where $mlen$ is the 64-bit representation of the bit length of M , $I_{mlen} = 0000000000000000F8$. In totality we get O_2 in hexadecimal format

$$O_2 = 1010101010101010\ 10100A0E0E0A0AF0\ 00000000000000F8$$

Now we can build X_0 by constructing $0^{27} \parallel O_0 \parallel 0^{64} \oplus C_0$ gives us X'_0 , in hexadecimal format

$$X'_0 = 0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F70\ 0000000000000000\ 0000000000000001$$

which decompresses to a point on the curve sect283r1 when $C_0 = 7$, or when X_0 in hexadecimal format is

$$X_0 = 0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F70\ 0000000000000000\ 0000000000000007$$

This decompresses using the OctetString-to-ECPoint conversion routine in [11], where $y_p =$ leftmost bit of N_0 , $y_p = 0$ gives us the point P_0

$$P_0 = (0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F70\ 0000000000000000\ 0000000000000007 , \\ 0000000005D2D2B2\ 6B08807376F86FC0\ 31616229DACDD074\ EDD97B55E07F882\ 753192E074303C35)$$

Which is added to Q , initialized as the point at infinity to become,

$$Q_0 = (0000000000000000\ 6162636465666768\ 696A6B6C6D6E6F70\ 0000000000000000\ 0000000000000007 , \\ 0000000005D2D2B2\ 6B08807376F86FC0\ 31616229DACDD074\ EDD97B55E07F882\ 753192E074303C35)$$

Now with O_1 we construct $X'_1 = 0^{27} \parallel O_1 \parallel 0^{64} \oplus C_1$, where C_1 starts = 1

$$X_1 = 0000000000000000\ 7172737475767778\ 797A616263646580\ 0000000000000001\ 0000000000000001$$

which decompresses directly to a point on the curve sect283r1 P_1

$$P_1 = (0000000000000000\ 7172737475767778\ 797A616263646580\ 0000000000000001\ 0000000000000001 , \\ 0000000003554E1B\ E469855991ACD3A7\ 684E71E8DE83B716\ 153E2A4D9F7555E7\ 5C26B1C1DDD4F660)$$

Which is added to Q to become,

$$Q_1 = (0000000002519C79 \text{ ABB30532C0277212 FA7527B2EF45BEAE CC90C6569765D7CC 004E63CF26E51764 , } \\ 00000000034745F9 \text{ D4060AFEF1B0D247 6A42DBFE6FDD9160 12D17A335769B05B 49974C557F48FF18 })$$

We can now construct the final block $X'_2 = 0^{27} \|O_2\| 0^{64} \oplus C_2$ with $C_2 = 1$, gives us X'_2 , in hexadecimal format

$$X'_2 = 0000000000000000 \text{ 1010101010101010 10100A0E0E0A0AF0 00000000000000F8 0000000000000001}$$

Which decompresses with $C_2 = 3$ to give us the point P_2 ,

$$P_2 = (0000000000000000 \text{ 1010101010101010 10100A0E0E0A0AF0 00000000000000F8 0000000000000003 , } \\ 00000000005101D6 \text{ 63A2FC2E0F1806E7 999BC9FAE01D787A 943C2574136DAC9E 2D0913E2E39F15AE })$$

Which is added to Q to become,

$$Q_2 = (000000000207FDCC \text{ 60A06C5113C7D974 EDCF68C574606DDA E011631B29590AD8 8F48C21504095B10 , } \\ 0000000005CC85CD \text{ 6153FA9C2D1FECD9 D83746919C4D67AC A12E9BF84AD8AD83 516402F90E9ABCC9 })$$

We can interpret the x-coordinate as an multi-precision integer and divide by two taking the floor of the result, or simply dropping the remainder to get the following in hexadecimal format

$$\lfloor x(Q_2)/2 \rfloor = 000000000103FEE6 \text{ 3050362889E3ECBA 76E7B462BA3036ED } \\ 7008B18D94AC856C \text{ 47A4610A8204AD88}$$

Using this value as an integer we can multiply the base point of the curve to get,

$$\lfloor x(Q_2)/2 \rfloor G = (0000000002703FE2 \text{ 871E5F59C4E80175 8A7C3F5964B213E3 } \\ \text{D111BCFC280B33DB 7FE445CB1C699729 , } \\ 000000000D12956 \text{ 76446E7693FFC04C BFCA6A94206F3C34 } \\ \text{BC4305D537C1A8CC 4B2636C2D5D6C8D4 })$$

Adding that value back with Q_2 above we get.

$$Q_2 + \lfloor x(Q_2)/2 \rfloor G = (00000000060883AD \text{ E578F697250191F3 0EB2F4094732BB66 } \\ \text{7D7D90AEB0C6AB30 EC9AC49D96EB54C8 , } \\ 000000000100886D \text{ 599FEB046E1F1968 2446D3D2870CB271 } \\ \text{2B6C4432B0D42044 3F6CDA67C42F2E2D })$$

And again interpreting the x-coordinate of the result as a multi-precision integer and dividing by 2 discarding the remainder to get

$$\lfloor x(Q + \lfloor x(Q_2)/2 \rfloor G)/2 \rfloor = 00000000030441D6 F2BC7B4B9280C8F9 87597A04A3995DB3 \\ 3EBEC85758635598 764D624ECB75AA64$$

From this result the rightmost 256-bits are extracted to get the message digest.

$$H = F2BC7B4B9280C8F987597A04A3995DB33EBEC85758635598764D624ECB75AA64$$

B.2.5 ECOH384 Example (one-block Message)

Let the message M be the 184-bit ASCII string “abcdefghijklmnopqrstuvw” which is equivalent to the hexadecimal string:

$$M = 6162636465666768696A6B6C6D6E6F7071727374757677$$

The message is padded by appending a one bit followed by 7 zero bits, resulting in the 192-bit value N, the hexadecimal string:

$$N_0 = 6162636465666768696A6B6C6D6E6F707172737475767780$$

O_0 is formed by prepending the 64-bit counter values for 0, 0000000000000000 to N_0 to get O_0 in hexadecimal format

$$O_0 = 6162636465666768 696A6B6C6D6E6F70 7172737475767780 0000000000000000$$

We form $O_1 = (\bigoplus N_i) \parallel I_{mlen}$, where $mlen$ is the 64-bit representation of the bit length of M, $I_{mlen} = 0000000000000000B8$. In totality we get O_1 in hexadecimal format

$$O_1 = 6162636465666768 696A6B6C6D6E6F70 7172737475767780 00000000000000B8$$

For O_0 we start with a counter value $C_0 = 1$ in hexadecimal format

$$C_0 = 0000000000000000 0000000000000000 0000000000000000 0000000000000000 \\ 0000000000000000 0000000000000000 0000000000000001$$

This when combined with $0^{89} \parallel O_0 \parallel 0^{64} \oplus C_0$ gives us X'_0 , in hexadecimal format

$$X'_0 = 0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70 \\ 7172737475767780 0000000000000000 0000000000000001,$$

which decompresses to a point on the curve sect409r1 when $C_0 = 3$, or when X_0 in hexadecimal format is

$X_0 = 0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70$
 $7172737475767780 0000000000000000 0000000000000001$

This decompresses using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_0 , $y_p = 0$ gives us the point P_0

$P_0 = (0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70$
 $7172737475767780 0000000000000000 0000000000000003 ,$
 $000000000191A0B0 6ADAB93E784A6F23 C8C5CB5034646DB2 31E882BADD533D18$
 $352E7A772DC80390 DE7F091C4BB9E0CC 1BC8AC79BD3D7641)$

Which is added to Q , initialized as the point at infinity to become,

$Q_0 = (0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70$
 $7172737475767780 0000000000000000 0000000000000003 ,$
 $000000000191A0B0 6ADAB93E784A6F23 C8C5CB5034646DB2 31E882BADD533D18$
 $352E7A772DC80390 DE7F091C4BB9E0CC 1BC8AC79BD3D7641)$

Now with O_1 we construct $X'_1 = 0^{89} || O_1 || 0^{64} \oplus C_1$, where C_1 starts = 1

$X'_1 = 0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70$
 $7172737475767780 000000000000000B8 0000000000000001$

which decompresses using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_1 , $y_p = 0$, when $C_1 = 17$, this gives us the point P_1

$P_1 = (0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70$
 $7172737475767780 000000000000000B8 0000000000000011 ,$
 $00000000011B9198 B0EA854681342F4B 42C836D91BB2FB7A 4436B25671A3296E$
 $3F76AE9BCFE67022 F916732B72AC3BDC 713BD501017E25E7)$

Which is added to Q to become,

$Q_1 = (0000000001AE7127 CA01E4D745CA9624 F82AFC5D6A3B11DD 0CA8F7CB68894F4F$
 $B8D6385FF9E66419 3BF48AE768F30073 EC08BC6C6A1A0DC3 ,$
 $000000000A814E0 A50090F86016B37E CE25C8423DA416C8 9C94BF40BED02C22$
 $CE28CC4B506B2F3D 1608EDE68DDE731A 3BA86C87ACAC2FC1)$

We can interpret the x-coordinate as an multi-precision integer and divide by two taking the floor of the result, or simply dropping the remainder to get the following in hexadecimal format

$$\lfloor x(Q_1)/2 \rfloor = 0000000000D73893\ E500F26BA2E54B12\ 7C157E2EB51D88EE\ 86547BE5B444A7A7\ 3C6B1C2FFCF3320C\ 9DFA4573B4798039\ F6045E36350D06E1$$

Using this value as an integer we can multiply the base point of the curve to get,

$$\lfloor x(Q_1)/2 \rfloor G = (0000000001B78FC0\ 9C0E1CC13B6CC64F\ EFE4BAB16991E533\ 4FFC78BEAB7FCA19\ 368B43A487E1F183\ C2BBBE0170113636\ 0CF226E944F177A7\ ,\ 0000000001485DE6\ C2D7CD42A23D3C2C\ 0F2708227101A9BD\ 4FE06F4CC63C9F73\ 9E650AB49E022983\ B9933045FCF47928\ B6EFF44E4B4C1199\)$$

Adding that value back with Q_1 above we get.

$$Q_1 + \lfloor x(Q_1)/2 \rfloor G = (00000000004542EB\ 2CA38A8BCBD25140\ 5E57E89918D54FE8\ D63EE9578E9F6BA4\ 5BF7CBE94A8C9622\ 8A8A45E961A70AA8\ 9FB050B5AE3E8C25\ ,\ 00000000015C8F57\ A6DFD85FD32359FD\ 10ABE69DA053A179\ EA12ACB77D1C3AFE\ 5E6AEEAF085DA460\ B7417E72C018185C\ 1A1A9A8642AD16B0\)$$

And again interpreting the x-coordinate of the result as a multi-precision integer and dividing by 2 discarding the remainder to get

$$\lfloor x(Q + \lfloor x(Q_1)/2 \rfloor G)/2 \rfloor = 000000000022A175\ 9651C545E5E928A0\ 2F2BF44C8C6AA7F4\ 6B1F74ABC74FB5D2\ 2DFBE5F4A5464B11\ 454522F4B0D38554\ 4FD8285AD71F4612$$

From this result the rightmost 384-bits are extracted to get the message

$$H = 9651C545E5E928A02F2BF44C8C6AA7F46B1F74ABC74FB5D2\ 2DFBE5F4A5464B11454522F4B0D385544FD8285AD71F4612$$

B.2.6 ECOH384 Example (two-block Message)

Let the message M be the 376-bit ASCII string “abcdefghijklmnopqrstuvwxyabcdefghijklmnopqrstu” which is equivalent to the hexadecimal string:

$$M = 6162636465666768696A6B6C6D6E6F70717273747576777\ 8797A6162636465666768696A6B6C6D6E6F707172737475$$

The message is padded by appending a one bit followed by 7 zero bits, resulting in the 128-bit value N , the hexadecimal string:

$$N_0 = 6162636465666768\ 696A6B6C6D6E6F70\ 7172737475767778$$

N_1 797A616263646566 6768696A6B6C6D6E 6F70717273747580

O_0 is formed by prepending the 64-bit counter values for 0, 0000000000000000 to N_0 to get O_0 in hexadecimal format

O_0 =6162636465666768 696A6B6C6D6E6F70 7172737475767778 0000000000000000

O_1 is formed by prepending the 64-bit counter values for 1, 0000000000000001 to N_1 to get O_1 in hexadecimal format

O_1 =797A616263646566 6768696A6B6C6D6E 6F70717273747580 0000000000000001

We form $O_2 = (\bigoplus N_i) || I_{mlen}$, where $mlen$ is the 64-bit representation of the bit length of M, $I_{mlen} = 0000000000000178$. In totality we get O_2 in hexadecimal format

O_2 =181802060602020E 0E0202060602021E 1E020206060202F8 0000000000000178

Now we can build X_0 by constructing $0^{89} || O_0 || 0^{64} \oplus C_0$, starting twith $C_0 = 1$, gives us X'_0 , in hexadecimal format

X'_0 =0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70
7172737475767778 0000000000000000 0000000000000005

which decompresses to a point on the curve sect409r1 when $C_0 = 5$ using the OctetString-to-ECPoint conversion routine in [11], where $y_p =$ leftmost bit of N_0 , $y_p = 0$ gives us the point P_0

P_0 =(0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70
7172737475767778 0000000000000000 0000000000000005 ,
000000000002A9D4 633A89ED45FC23E9 6476C50243F01639 CB31E3755862468D
3665E103B45D4318 37C9221F30DF8347 2A19D141C426D5F6)

Which is added to Q, initialized as the point at infinity to become,

Q_0 =(0000000000000000 0000000000000000 6162636465666768 696A6B6C6D6E6F70
7172737475767778 0000000000000000 0000000000000005 ,
000000000002A9D4 633A89ED45FC23E9 6476C50243F01639 CB31E3755862468D
3665E103B45D4318 37C9221F30DF8347 2A19D141C426D5F6)

Now with O_1 we construct $X'_1 = 0^{89}||O_1||0^{64} \oplus C_1$, where C_1 starts = 1

```
X1 =0000000000000000 0000000000000000 797A616263646566 6768696A6B6C6D6E
      6F70717273747580 0000000000000001 0000000000000001
```

Which decompresses directly to a point on the curve sect409r1 P_1

```
P1 =(0000000000000000 0000000000000000 797A616263646566 6768696A6B6C6D6E
      6F70717273747580 0000000000000001 0000000000000001 ,
      0000000017ABCB5 8579B719BE59AB9A 1924229BF48A2D52 F0C99A232E5B1679
      D4D146A98DE6CA55 0474671158DA55F5 678B24D3D85FD7D6 )
```

Which is added to Q to become,

```
Q1 =(00000000129965A 444E38624E57FEAF D0B88E7FD0E2B4BE BD95E05CE5E628B4
      47F2A31B93D7BE2A E028E2238772B39B C1CFF55E90133459 ,
      000000001031F49 E841BAFBFF99A6EB 406A00AF92BED691 B214017BBD707715
      CA68AF201F47B129 92EF935A9D630BDD D7140567E370AD0B )
```

We can now construct the final block $X'_2 = 0^{89}||O_2||0^{64} \oplus C_2$ with $C_2 = 1$, gives us X'_2 , in hexadecimal format

```
X'2 =0000000000000000 0000000000000000 181802060602020E 0E0202060602021E
      1E020206060202F8 000000000000178 0000000000000001
```

Which decompresses when $C_2 = 3$ to the point P_2 ,

```
P2 =(0000000000000000 0000000000000000 181802060602020E 0E0202060602021E
      1E020206060202F8 000000000000178 0000000000000003 ,
      00000000099B9CC 7CE297AEC658E24B DAA59D3A01E53787 FBCEFA70EB5707DB
      3101CED05D713B61 341C6BFCC72130AA 57EC7CC71D1781A4 )
```

Which is added to Q to become,

```
Q2 =(0000000008A53C3 3C34D0CD168EDCBB 05B2C39C996B4021 37D726FFBCC16B08
      04891FA81733FEBC 9D8244BFE15FDDD4 027ADDA37EF366E5 ,
      000000001EEAF69 56ACF64D091381E7 06BC51323CF613F9 C9E84B88343F4C4A
      8A4D1B7CD4F3AFEB 49349155A9345A16 F2EBE6A4F6F11F37 )
```

We can interpret the x-coordinate as an multi-precision integer and divide by two taking the floor of the result, or simply dropping the remainder to get the following in hexadecimal format

$$\lfloor x(Q_2)/2 \rfloor = 00000000004529E1\ 9E1A68668B476E5D\ 82D961CE4CB5A010\ 9BEB937FDE60B584\ 02448FD40B99FF5E\ 4EC1225FF0AFEEEE\ 013D6ED1BF79B372$$

Using this value as an integer we can multiply the base point of the curve to get,

$$\lfloor x(Q_2)/2 \rfloor G = (0000000000DC6524\ 44632FA528E947A2\ 0E5D0FB8FF7E0C1B\ 842337D61EABD6A6\ 2C4359023177EF84\ FF02FC1B4D4957A0\ EF4A96F197D3DCF7\ ,\ 0000000000C04329\ FB7FA27CE5C60AFF\ 9F9769998EF12B41\ 310EBEBFC10231C9\ E9E04524694CBBB1\ 27DF3B511DDD2C5B\ 6FB5EEDB90B0BCE2\)$$

Adding that value back with Q_2 above we get.

$$Q_2 + \lfloor x(Q_2)/2 \rfloor G = (0000000001C758E8\ 0547F77A76A9B832\ 9FBBAD13C66C0843\ 7234301ACD1DE3F8\ 140CFAD7B732CD42\ 36B9A51578E871E1\ 2FF5441CD765730B\ ,\ 0000000000C2EF8E\ 4926518537EEA213\ B21A1039ED86ADB3\ 328F012E00176610\ 195F8DF69740D13F\ 42A5C67020860282\ C3DD7FDE36C4FF82\)$$

And again interpreting the x-coordinate of the result as a multi-precision integer and dividing by 2 discarding the remainder to get

$$\lfloor x(Q + \lfloor x(Q_2)/2 \rfloor G)/2 \rfloor = 0000000000E3AC74\ 02A3FBBD3B54DC19\ 4FDDD689E3360421\ B91A180D668EF1FC\ 0A067D6BDB9966A1\ 1B5CD28ABC7438F0\ 97FAA20E6BB2B985$$

From this result the rightmost 384-bits are extracted to get the message

$$H = 02A3FBBD3B54DC194FDDD689E3360421B91A180D668EF1FC\ 0A067D6BDB9966A11B5CD28ABC7438F097FAA20E6BB2B985$$

B.2.7 ECOH512 Example (one-block Message)

Let the message M be the 248-bit ASCII string “abcdefghijklmnopqrstuvwxyabcde” which is equivalent to the hexadecimal string:

$$M = 6162636465666768696A6B6C6D6E6F707172737475767778797A6162636465$$

The message is padded by appending a one bit followed by 7 zero bits, resulting in the 256-bit value N, the hexadecimal string:

$$N_0 = 6162636465666768696A6B6C6D6E6F707172737475767778797A616263646580$$

O_0 is formed by prepending the 128-bit counter values for 0, 0000000000000000 0000000000000000 to N_0 to get O_0 in hexadecimal format

$$O_0 = 6162636465666768 \ 696A6B6C6D6E6F70 \ 7172737475767778 \ 797A616263646580 \\ 0000000000000000 \ 0000000000000000$$

We form $O_1 = (\bigoplus N_i) || I_{mlen}$, where $mlen$ is the 64-bit representation of the bit length of M, $I_{mlen} = 0000000000000000 \ 0000000000000000F8$. In totality we get O_1 in hexadecimal format

$$O_1 = 6162636465666768 \ 696A6B6C6D6E6F70 \ 7172737475767778 \ 797A616263646580 \\ 0000000000000000 \ 00000000000000F8$$

For O_0 we start with a counter value $C_0 = 1$ in hexadecimal format

$$C_0 = 0000000000000000 \ 0000000000000000 \ 0000000000000000 \ 0000000000000000 \ 0000000000000000 \\ 0000000000000000 \ 0000000000000000 \ 0000000000000000 \ 0000000000000001$$

This when combined with $0^{59} || O_0 || 0^{128} \oplus C_0$ gives us X'_0 , in hexadecimal format

$$X'_0 = 0000000000000000 \ 6162636465666768 \ 696A6B6C6D6E6F70 \ 7172737475767778 \ 797A616263646580 \\ 0000000000000000 \ 0000000000000000 \ 0000000000000000 \ 0000000000000001,$$

which decompresses when $C_0 = 9$ giving the first point on the curve sect571r1, P_0

$$P_0 = (0000000000000000 \ 6162636465666768 \ 696A6B6C6D6E6F70 \ 7172737475767778 \ 797A616263646580 \\ 0000000000000000 \ 0000000000000000 \ 0000000000000000 \ 0000000000000009 , \\ 041B4ABA9CB08A2F \ F43895E48F2F7E6C \ 44AF5AA4FA946FBB \ AF17273DCACBACEE \ E5B44857D9FEA5F5 \\ D7F2D8F508257492 \ 0B3D1678050BB9F4 \ 279EF306EDF6D802 \ DB31BACE8B9E1BF8)$$

Which is added to Q, initialized as the point at infinity to become,

$$Q_0 = (0000000000000000 \ 6162636465666768 \ 696A6B6C6D6E6F70 \ 7172737475767778 \ 797A616263646580 \\ 0000000000000000 \ 0000000000000000 \ 0000000000000000 \ 0000000000000009 , \\ 041B4ABA9CB08A2F \ F43895E48F2F7E6C \ 44AF5AA4FA946FBB \ AF17273DCACBACEE \ E5B44857D9FEA5F5 \\ D7F2D8F508257492 \ 0B3D1678050BB9F4 \ 279EF306EDF6D802 \ DB31BACE8B9E1BF8)$$

Now we form $X'_1 = 0^{59} || O_1 || 0^{128} \oplus C_1$, starting with $C_1 = 1$ as an integer to get

$$X'_1 = 0000000000000000 \ 6162636465666768 \ 696A6B6C6D6E6F70 \ 7172737475767778 \ 797A616263646580 \\ 0000000000000000 \ 00000000000000F8 \ 0000000000000000 \ 0000000000000001,$$

which decompresses directly to a point on the curve sect571r1

$$P_1 = (0000000000000000 6162636465666768 696A6B6C6D6E6F70 7172737475767778 797A616263646580 \\ 0000000000000000 00000000000000F8 0000000000000000 0000000000000001 , \\ 04F740D7339C396A 0DEA35193BFA3BE0 F2A46DFCA56B0816 F44047BCDE11E988 FD012355EA07828B \\ 90F659F7994F5DA2 59BA39AD49DAF7E9 8182E605BB194C70 A6AACCAD71073F00)$$

Which is added to Q to become,

$$Q_1 = (0747F927845CBFB2 DB3250B42CE6E164 B2E18A90038C6B6D \\ CB42FFF958AB7461 E423DE8C9ED78452 1B59B431C5362C85 \\ 26EA76598FCB278A 0C6B6463A96881BF 49FC7936FD30639D , \\ 0149C28B83EAA241 007B439BBF6BEE55 11A94AF32B9B12CC \\ 9771DA9A94B46123 E79FA43D15108893 0942D4789DDA39AF \\ BD410D2F2CF8769E A75CE0CEC2A88116 6602DAB7C4F1010E)$$

We can interpret the x-coordinate as an multi-precision integer and divide by two taking the floor of the result, or simply dropping the remainder to get the following in hexadecimal format

$$[x(Q_1)/2] = 03A3FC93C22E5FD9 6D99285A167370B2 5970C54801C635B6 E5A17FFCAC55BA30 F211EF464F6BC229 \\ ODACDA18E29B1642 93753B2CC7E593C5 0635B231D4B440DF A4FE3C9B7E9831CE$$

Using this value as an integer we can multiply the base point of the curve to get,

$$[x(Q_1)/2]G = (060860C6EC61FA09 8D9EAC0A03A878C1 6C780A513ED3F8B1 DF65621ADE1E4E4B \\ B4AE44AF20E60C745 7A7C4770FE73EC8F E9F0912EF286E469 92419CEF5133CB14 \\ E86650CE7F9EF118 , \\ 0209C682B18D8F99 4AC9F4BEBFA04782 3BA5832E0F06CD81 107840A75D49A418 \\ 39611E69FFC12DA9 9E242FA8CD6603E5 F325657E38AF35AD FC536DF994E4EF2E \\ CC32CF057E8ABBA2)$$

Adding this result back into the value of Q_1 above we get

$$Q_1 + [x(Q_1)/2]G = (07FD7954FDFACFCC 2BAB2EDA39C2647C F96EF37BD2C0CB89 B39EA19F305304F3 \\ 6556404C17C2E3A9 E6D38344370CE6FC FB926B77ACE984DE D7ED35C8F53666FB \\ 2AA783C047902C5A , \\ 059D22A547B3E3FC A05BBEB41658B6A3 FC20C0AA8C5B5205 0AEAB15DB65D0E26 \\ FB20C040FFFA3FFF 3530CBFD184BD89C 1A596CD87C9CA82A C264832821CFE0E3 \\ B023BCF1EOD8099B)$$

And again interpreting the x-coordinate of the result as a multi-precision integer and dividing by 2 discarding the remainder to get

$$\lfloor x(Q + \lfloor x(Q_1)/2 \rfloor G)/2 \rfloor = 03FEBCAA7EFD67E6 15D5976D1CE1323E 7CB779BDE96065C4 D9CF50CF98298279 \\ B2AB20260BE171D4 F369C1A21B86737E 7DC935BBD674C26F 6BF69AE47A9B337D \\ 9553C1E023C8162D$$

From this result the rightmost 512-bits are extracted to get the message

$$H = 15D5976D1CE1323E7CB779BDE96065C4D9CF50CF98298279B2AB20260BE171D4 \\ F369C1A21B86737E7DC935BBD674C26F6BF69AE47A9B337D9553C1E023C8162D$$

B.2.8 ECOH512 Example (two-block Message)

Let the message M be the 504-bit ASCII string “abcdefghijklmnopqrstuvwxyabcdefghijklmnopqrstuvwxyabcdefghijklmnop” which is equivalent to the hexadecimal string:

$$M = 6162636465666768696A6B6C6D6E6F707172737475767778797A616263646566 \\ 6768696A6B6C6D6E6F707172737475767778797A6162636465666768696A6B$$

The message is padded by appending a one bit followed by 7 zero bits, resulting in the 128-bit value N, the hexadecimal string:

$$N_0 = 6162636465666768 696A6B6C6D6E6F70 7172737475767778 797A616263646566$$

$$N_1 = 6768696A6B6C6D6E 6F70717273747576 7778797A61626364 65666768696A6B80$$

O_0 is formed by prepending the 128-bit counter values for 0, 0000000000000000 0000000000000000 to N_0 to get O_0 in hexadecimal format

$$O_0 = 6162636465666768 696A6B6C6D6E6F70 7172737475767778 797A616263646566 0000000000000000 \\ 0000000000000000$$

O_1 is formed by prepending the 128-bit counter values for 1, 0000000000000000 0000000000000001 to N_1 to get O_1 in hexadecimal format

$$O_1 = 6768696A6B6C6D6E 6F70717273747576 7778797A61626364 65666768696A6B80 0000000000000000 \\ 0000000000000001$$

We form $O_2 = (\bigoplus N_i) \parallel I_{mlen}$, where $mlen$ is the 64-bit representation of the bit length of M, $I_{mlen} = 00000000000000178$. In totality we get O_2 in hexadecimal format

$O_2 = 060A0A0E0E0A0A06 061A1A1E1E1A1A06 060A0A0E1414141C 1C1C060A0A0E0EE6 0000000000000000$
 $000000000000001F8$

Now we can build X_0 by constructing $0^{59}||O_0||0^{128} \oplus C_0$, starting twith $C_0 = 1$, gives us X'_0 , in hexadecimal format

$X'_0 = 0000000000000000 61626364656666768 696A6B6C6D6E6F70 7172737475767778 797A616263646566$
 $0000000000000000 0000000000000000 0000000000000000 0000000000000001$

which decompresses to a point on the curve sect571r1 immediately using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_0 , $y_p = 0$ gives us the point P_0

$P_0 = (0000000000000000 61626364656666768 696A6B6C6D6E6F70 7172737475767778 797A616263646566$
 $0000000000000000 0000000000000000 0000000000000000 0000000000000001 ,$
 $06023280BA1084E8 53FDFA74117D8351 E4CD2928344A5FB3 B50D292CAB0487D4 ABDA5275070E1ECC$
 $68593B676134E007 48ABEC52B3B6075D 77833B892B35F67C 97EFFC9C5C7C01D9)$

Which is added to Q, initialized as the point at infinity to become,

$Q_0 = (0000000000000000 61626364656666768 696A6B6C6D6E6F70 7172737475767778 797A616263646566$
 $0000000000000000 0000000000000000 0000000000000000 0000000000000001 ,$
 $06023280BA1084E8 53FDFA74117D8351 E4CD2928344A5FB3 B50D292CAB0487D4 ABDA5275070E1ECC$
 $68593B676134E007 48ABEC52B3B6075D 77833B892B35F67C 97EFFC9C5C7C01D9)$

Now with O_1 we construct $X'_1 = 0^{59}||O_1||0^{128} \oplus C_1$, where C_1 starts = 1

$X'_1 = 0000000000000000 6768696A6B6C6D6E 6F70717273747576 7778797A61626364 65666768696A6B80$
 $0000000000000000 0000000000000001 0000000000000000 0000000000000001$

which decompresses to a point on the curve sect571r1 immediately using the OctetString-to-ECPPoint conversion routine in [11], where $y_p =$ leftmost bit of N_1 , $y_p = 0$ gives us the point P_1

$P_1 = (0000000000000000 6768696A6B6C6D6E 6F70717273747576 7778797A61626364 65666768696A6B80$
 $0000000000000000 0000000000000001 0000000000000000 0000000000000001 ,$
 $0387314F6143173F 29EBA96F515C14E7 1B6BE9E5ABC36357 DB0262505CB8B8F2 257CD69F8004EBDD$
 $5474D254F7C41A63 DED3BCD44745B0B9 354E125C083CE112 FD9350AF0CD889CD)$

Which is added to Q to become,

$Q_1 = (06311BD41C6CBB86\ 7258756A39571FE8\ 0FC4C383B985CF31\ 5C167B80042646AB\ 98E700F4BAE6AFB5$
 $ABC658FBC155AEDE\ 5498BE508598BF02\ C5C2AA72A67B844C\ 5FABCF7E74BB08D6\ ,$
 $020CD10E355A305A\ 315F9FA157F228DF\ 5330BE8B051C7A18\ FD9A1189C63C8569\ 6612006EEB9329B8$
 $2F8F92FFC8FC65B6\ B66102C48297EDED\ E9A914EF1565F78E\ 52D3C826BD3CB250\)$

We can now construct the final block $X'_2 = \text{with}0^{59}||O_2||0^{128} \oplus C_2$ with $C_2 = 1$, gives us X'_2 , in hexadecimal format

$X'_2 = 0000000000000000\ 060A0A0E0E0A0A06\ 061A1A1E1E1A1A06\ 060A0A0E1414141C\ 1C1C060A0A0E0EE6$
 $0000000000000000\ 00000000000001F8\ 0000000000000000\ 0000000000000001,$

which again decompresses immediately to the point P_2

$P_2 = (0000000000000000\ 060A0A0E0E0A0A06\ 061A1A1E1E1A1A06\ 060A0A0E1414141C\ 1C1C060A0A0E0EE6$
 $0000000000000000\ 00000000000001F8\ 0000000000000000\ 0000000000000001\ ,$
 $004D3AEE2C2E680D\ 10A4D9C7664F443D\ BE370B6AAF9D406A\ ECB053984391369F\ 7B7CF596A9DA7C53$
 $3CA556DOE498C4EE\ BE31B50EA3D7A94F\ 13CC3510E7D77A6D\ D7FA7CB19E55BDCF\)$

P_2 is added to the Q value to result in

$Q_2 = (00E2F5EC93E7F9BD\ 39D9D90F93A320DA\ DF2BF9FFA2F7EC5B\ EABC90A94DD39CFC\ 915217DC96D1BF6C$
 $B844D92BEBC170F2\ 624FE38CB7AFDCF4\ 8002400F04A18C8A\ 3D336DB871B99AC2\ ,$
 $0483978E80D268EF\ 7B9CFA5C45F96794\ 4F8B909A064BF4E7\ 3FDFD41EFBD9ED18\ 60BD39AF8DDF7BFE$
 $C4B6D1E6BE00F66A\ FBB037F94DC392DB\ 06FF42E2727F49AB\ DD753A6829C3CC85\)$

We can interpret the x-coordinate as an multi-precision integer and divide by two taking the floor of the result, or simply dropping the remainder to get the following in hexadecimal format

$\lfloor x(Q_2)/2 \rfloor = 00717AF649F3FCDE\ 9CECEC87C9D1906D\ 6F95FCFFD17BF62D\ F55E4854A6E9CE7E$
 $48A90BEE4B68DFB6\ 5C226C95F5E0B879\ 3127F1C65BD7EE7A\ 400120078250C645$
 $1E99B6DC38DCCD61$

Using this value as an integer we can multiply the base point of the curve to get,

$\lfloor x(Q_2)/2 \rfloor G = (0052F6968D4E4078\ E4CA24ACA4D2E9C3\ 305EEDD56678A890\ 4380D020F5058224$
 $2312BAC171F6B3A9\ 823CE263E5769A4D\ 9C21B4F7B8058B1B\ 00560A3DC84664F7$
 $1ED546F2F1FA3822\ ,$
 $0779600D4F72A682\ 2863B44C3B02128F\ A006CADAEB5F5146\ ECCC8977780D2693$
 $A028F6582FDFFEF7\ AFD3C3054194CF29\ FD5AAC1B66D54465\ F08A406A8E7ABC7B$
 $6AD49BC2640BE5FD\)$

Adding that value back with Q_2 above we get.

$$Q_2 + \lfloor x(Q_2)/2 \rfloor G = (07A658BF3ECCC401\ 328134D228C58B69\ AE907C02518CC92D\ 3BBDE19CD10B119D \\ D83DF56F838AB621\ A111B616425410F6\ 9C861D53DD1E7631\ 2143CF25316DDC9C \\ 81289F00785B0295\ , \\ 01C3010607099A57\ C63D7C7519B81B62\ C630E3D629B77134\ 5F9EC34D17D0A253 \\ 30D25DF1AEA2ADFB\ 5BA8CE6B3E8058D9\ B536E1BE513BA482\ F2BF257D29DE976A \\ 1324EAE5D6589280\)$$

And again interpreting the x-coordinate of the result as a multi-precision integer and dividing by 2 discarding the remainder to get

$$\lfloor x(Q + \lfloor x(Q_2)/2 \rfloor G)/2 \rfloor = 03D32C5F9F666200\ 99409A691462C5B4\ D7483E0128C66496 \\ 9DDEF0CE688588CE\ EC1EFAB7C1C55B10\ D088DB0B212A087B \\ 4E430EA9EE8F3B18\ 90A1E79298B6EE4E\ 40944F803C2D814A$$

From this result the rightmost 512-bits are extracted to get the message

$$H = 99409A691462C5B4D7483E0128C664969DDEF0CE688588CEEC1EFAB7C1C55B10 \\ D088DB0B212A087B4E430EA9EE8F3B1890A1E79298B6EE4E40944F803C2D814A$$