Diffie-Hellman Key Exchange on the TI MSP430F2618 Microprocessor

Shawn Johnson and Ben Sattelberg

May 3, 2015

Contents

List of Figures

1 Background

Cryptography is defined as the practice and study of techniques for secure communication in the presence of third parties. The importance of cryptography is to keep information transferred between two parties, whether they are public individuals or government agents, safe from from prying eyes.

1.1 Private-Key Cryptosystems

Private-key systems require the transfer of a private, shared key before encryption can take place. Since this key is used for both decryption and encryption, it is important that it is not accessible by third parties. The most common modern private-key cryptosystems have "symmetric" keys, meaning that encryption and decryption follow the same process. They also encode bits by applying certain bitwise operations involving the message and some set of subkeys generated by the key.

1.2 The Advanced Encryption Standard

The Advanced Encryption Standard is a private-key cryptosystem with symmetric keys that is based on the Rijndael cipher. It uses multiple subkeys to encrypt 128 bit blocks of a message using multiple bitwise operations and table lookups.

In 2001, The Advanced Encryption Standard (AES) was announced by the National Institute of Standards and Technology (NIST) as sufficient for sensitive (unclassified) data [2]. It was later certified by the National Security Association (NSA) as valid for up to top secret information [3].

AES encryption has become the standard for civilian and governmental use. However, it requires transfer of a key between the parties sending information before encryption. These transfers usually take place before encryption must take place, or by using a key exchange system.

1.3 Public-Key Cryptosystems

Public-key systems utilize some computationally difficult problem to be able to encode messages that are difficult to decode. Some private key is first generated — for example, in RSA encryption, this private key is two large primes. This private key is used to generate a public key - for RSA, the product of the two primes. The public key is used to encrypt messages, and the private key is used to decrypt them.

Encryption and decryption using public-key systems are typically computationally intensive. Because of this, public-key systems are often used to exchange keys for a private-key system so that the bulk of the encryption is done using the private-key system.

1.4 Diffie-Hellman Encryption

Diffie-Hellman encryption is a form of encryption that relies on the difficulty of the discrete logarithm problem to generate keys. It creates both public and private keys using exponentiation in a cyclic group, and encrypts under the

group's operation. For proper choices of group and exponents, Diffie-Hellman encryption is safe and the generation of the keys can be done quickly. [1]

1.5 Our Problem

A medical device that operates using a Texas Instruments MSP430F2618 microprocessor with an AES256 encryption module needs to be able to transfer sensitive data. It is infeasible for the AES256 keys to be stored on the processor, so we would like to transfer the AES256 keys using a Diffie-Hellman key exchange. We want to be able to generate AES256 keys daily and a Diffie-Hellman key each month. To do this, we need to be able to generate the Diffie-Hellman key within one hour and each AES256 key within 30 seconds. We also want the Diffie-Hellman key to be able to withstand a month long attack costing less than $100,000 in hardware in 2025.

1.5.1 The TI MSP430F2618

The TI MSP430F2618 is a microprocessor designed to have low power consumption. It has a 16 MHz processor, and so can perform 16,000,000 byte operations per second. It has 8KB of RAM and 116KB of flash memory. It stores signed integers, and so multiplications and additions are done using the signed integer representation of bit strings. Operations can also operate either on bytes (8 bits) or on "words" (16 bits). Byte operations typically take one clock cycle and word operations typically take two clock cycles. [4] [7]

2 Mathematical Theory

2.1 Discrete Logarithm Problem

Definition 2.1. Given the elements g and b of a cyclic group G, the *Discrete Logarithm Problem* is to find a natural number k such that

$$g^k = b.$$

The discrete logarithm problem is currently considered to be "difficult." The current algorithms for general problems to solve it have run times of at least $O(\sqrt{|G|})$ where $|G|$ is the number of elements in the group G. For cryptographic purposes, the run time is typically considered in terms of the number of digits in $|G|$. In that case, the time complexity of the algorithms is exponential. Although the discrete logarithm problem is considered difficult, no formal proofs of its difficulty (NP-completeness) exist for anything but generic groups.

2.2 Diffie-Hellman Problem

Definition 2.2. Given a cyclic group G and the elements g, g^a, and g^b in the group, then find g^{ab}.

The current most efficient solutions to a general Diffie-Hellman problem rely on solving the discrete logarithm problem for either a or b and then taking

$$(g^a)^b = (g^b)^a = g^{ab}.$$

Since the discrete logarithm problem is difficult, so fare the Diffie-Hellman problem is considered difficult. The Diffie-Hellman problem has not been proven to rely on the discrete logarithm problem.

2.3 Diffie-Hellman Encryption

Diffie-Hellman encryption uses the difficulty of the Diffie-Hellman problem to create a secure public key cryptosystem. It uses g^{ab} as the private key and both g^a and g^b as public keys. A third party cannot understand encrypted messages without having determined g^{ab}.

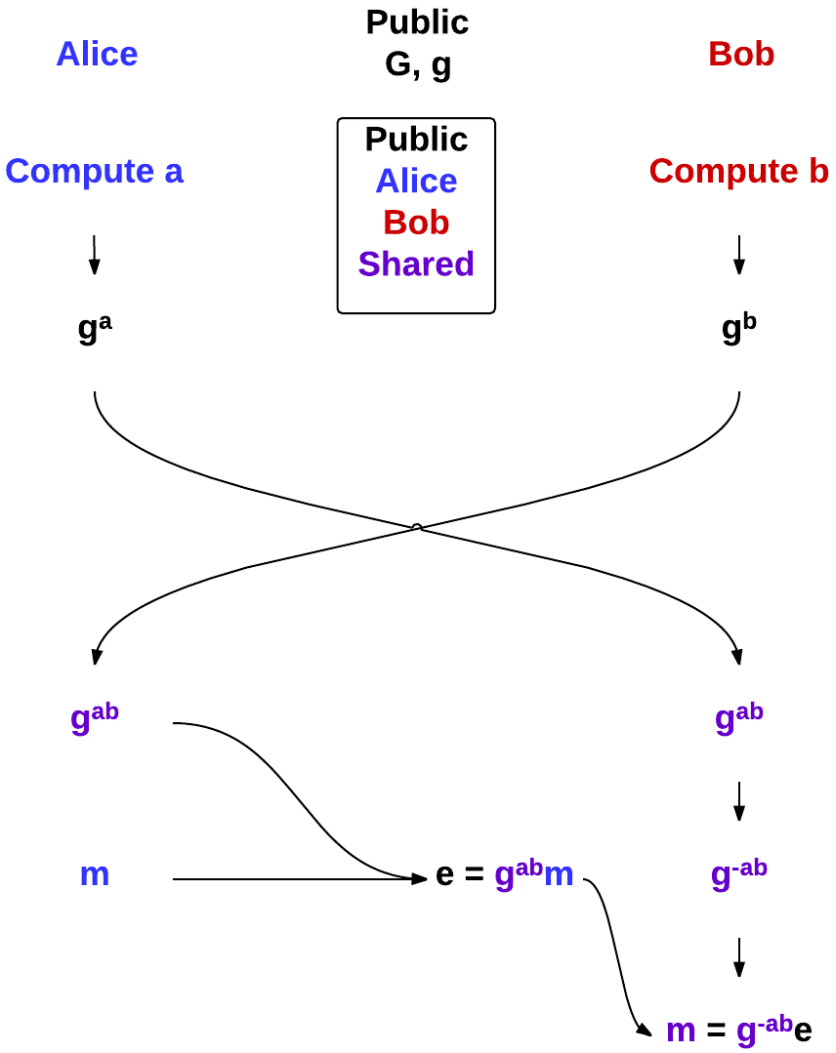

Figure 1: Diffie-Hellman Encryption

We use figure 1 to illustrate the process of Diffie-Hellman encryption. For this example we will let Alice and Bob be two parties who want to share information using a Diffie-Hellman scheme. In figure 1, the legend represents the availability of information. Black is used to represent publicly known variables, blue represents variables known only to Alice, red represents variables known only to Bob, and purple represents variables that both Alice and Bob know. The first step for Alice and Bob is to determine what cyclic group, G, and generator, $g \in G$, they will use. After they decide this, they will both determine their respective private key. They generate random $a, b \in \mathbb{N}$, respectively. Bob only knows b and Alice only knows a. Alice and Bob transfer g^a and g^b publicly. They use these to compute

$$(g^a)^b = g^{ab} = g^{ba} = (g^b)^a.$$

They each then have the secret key, g^{ab} and can exchange information privately.

To illustrate the process of sending information, assume Alice wishes to send Bob a message, m. Alice would convert m into a set of group elements, and take $e = m * g^{ab}$, where $*$ represents the group operation. Bob is then able to calculate $m = e * g^{-ab}$ and receive the original message.

We now consider a third party, Eve, who wishes to know the message Alice and Bob are transferring. Eve would know G, g, g^a, g^b, and e. To compute m, Eve must find the element g^{-ab} — to do this, she must solve the Diffie-Hellman problem for g^{ab}. Since this is difficult, the message transferred between Alice and Bob is safe. [1]

Diffie-Hellman encryption can be extended to include more than two parties who want to exchange information. For this, each party calculates an $x_i \in \mathbb{N}$ and transfers g^{x_i} publicly to a different party. That party then sends $(g^{x_i})^{x_j}$ to another and so on. This process is repeated until everyone has the secret key

$$g^{\prod_{i=1}^{N} x_i}$$

Where N is the number of parties that are exchanging information. The information that is public here is $G, g, g^{x_i}, g^{x_i x_j}, ..., g^{\prod_{j \in C} x_j}$, $i = 1, ..., N$, $C = \{1, ..., N\} - \{i\}$, with the private information being $x_i, g^{\prod_{j=1}^{N} x_j}$, $i = 1, ..., N$.

2.4 Group Choices for Diffie-Hellman encryption

Although Diffie-Hellman encryption works over any finite cyclic group, the most common choices are elliptic curves over a finite field and the multiplicative integers modulo a prime p [6].

2.4.1 Elliptic Curves over a finite field

Definition 2.3. An *elliptic curve* is a smooth projective algebraic curve of genus one, on which there is a specified point O.

This curve can be considered to be a commutative group under algebraic multiplication with the point O being the identity. The elements of this group are the points on the curve that the finite field creates. We will use figure 2 to help explain how the elliptic curves work.

Here, we will denote the operation of the group with $+$. To find $R = Q + P$ we first make a straight line intersecting both Q and P. We then follow the line

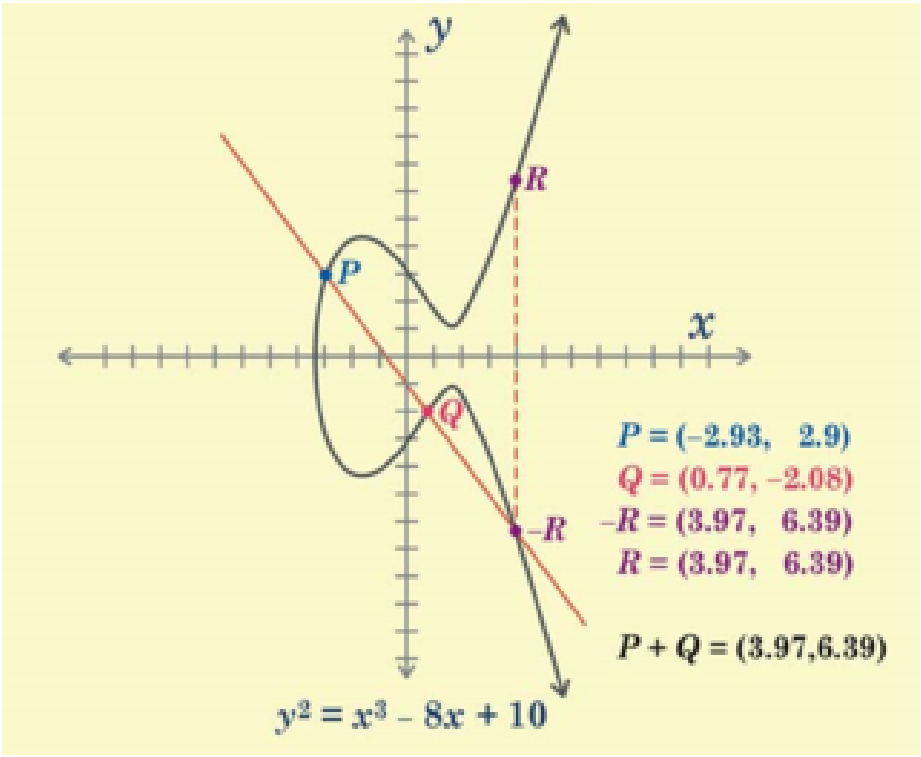

Figure 2: Addition of P and Q on an Elliptic Curve
Available: http://a5.typepad.com/6a01a510678336970c01a511bfe2c5970c-320wi

until it intersect the elliptic curve a third time at the point $-R$. We then find the inverse of $-R$, R, so that $-R + R = O$.

The numerical computation of R requires more steps. Q and P are defined by their 2-D points, that is $P(x_P, y_P)$ and $Q(x_Q, y_Q)$. The first step is to determine if $x_P \neq x_Q$. If this is true,

$$\begin{aligned} s &= \frac{y_P - y_Q}{x_P - x_Q}, \\ x_R &= s^2 - x_P - x_Q \quad \text{and} \\ y_R &= y_P + s(x_R - x_P) \end{aligned}$$

Otherwise, must change s, x_R to

$$\begin{aligned} s &= \frac{3x_P^2 - p}{2y_P} \\ x_P &= s^2 - x_P - x_Q. \end{aligned}$$

Here, p is determined from the elliptic curve of the form $y^2 = x^3 - px - q$. Without the proper hardware, these calculations are slow for useful finite fields. [8]

2.4.2 Multiplicative integers modulo a prime p

Definition 2.4. $U(p)$ is the set of all $x \in \mathbb{N}$ such that $x < p$ and x is relatively prime to p.

This is a group under the operation "$\cdot_p$":

$$a \cdot_p b = ab \mod p.$$

Unlike elliptic curves over finite fields, calculations in $U(p)$ are easy to implement.

When p is prime, this group is cyclic, and so it can be used for Diffie-Hellman Encryption. It will also have $|U(p)| = p - 1$, so the discrete logarithm problem will be difficult for large p.

For the actual implementation, we would like $|U(p)|$ to have a large prime factor. To accomplish this, we pick p to be a large Sophie Germain prime, that is, $p = 2q + 1$ where q is another large prime. We do this to limit the order of subgroups generated by an element in $U(p)$ to be either $1, 2, q$ or $2q$. We know this will occur because $|U(p)| = 2q$, and by Lagrange's Theorem that the order of a subgroup must divide the order of a group.

Next, the element g, as discussed in the Diffie-Hellman Encryption section, is chosen to generate a subgroup with order q. If g was chosen to generate the entire group $U(p)$, there are attacks that are able to determine the least significant bit of a (whether a is even or odd) and thus halve the amount of security. Any such g is equally secure, so it is typically chosen to be small to aid in computation.

2.5 Choice of Cyclic Group

We will implement our Diffie-Hellman scheme using the multiplicative integers modulo a prime p. Although this requires larger group sizes and more data to be stored, the computations required to implement elliptic curves would make it infeasible to stay within the time parameters.

3 Algorithms

Implementing the AES256 key generation and the Diffie-Hellman key generation on the MSP430F2618 requires implementations of certain operations. Although MSP430F2618 includes binary operations for signed, 8 or 16 bit integers, we use arbitrary length unsigned integers. We also need algorithms that the MSP430F2618 does not have naturally. The following sections discuss our implementations, and code written for the MSPF2618 for these algorithms is included in the appendix.

3.1 Random Number Generation

Random numbers are required for true security. If the variables were not created randomly, there could be attacks that analyze the generation method instead of trying to crack the key. We would like to generate our AES256 key and a, our private key, randomly to keep our system secure. To do this, we will use the Very Low frequency Oscillator (VLO) and Digitally Controlled

Oscillator (DCO) in the MSP430F2618 to generate sequences of random bits. This process is described in figure 3.

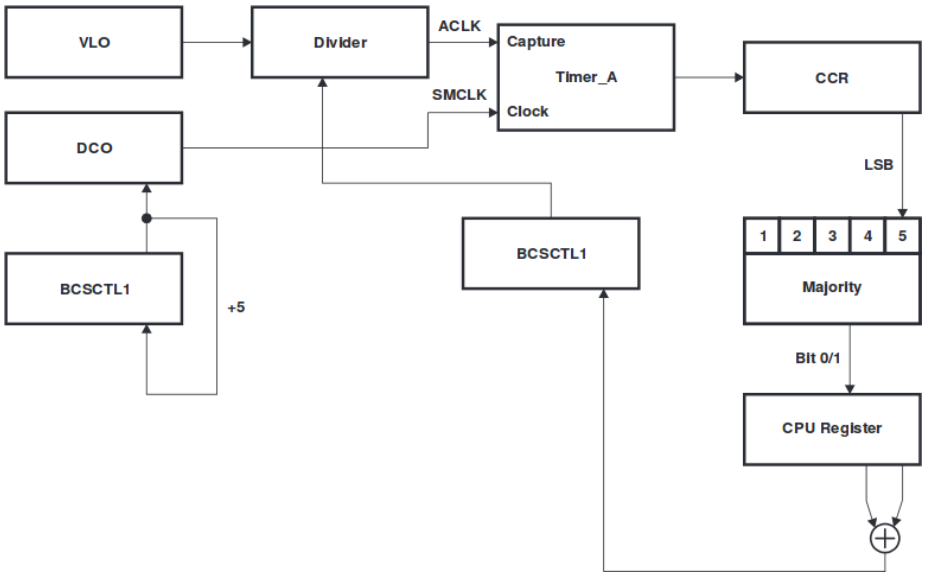

Figure 3: Random Number Generation on the MSP430F2618 [5]

In one clock cycle of the VLO, there are roughly the same number of DCO pulses. Since the two systems are independent, whether the DCO oscillates an odd or even number of times per VLO oscillation is random. Additionally, the previous number of pulses doesn't affect the number for the next VLO clock cycle. The system determines how many DCO pulses are made in one VLO clock cycle, and saves whether this number is odd or even. We do this 16 times to fill a register. To add more randomness, each time we save a bit we modify the settings of the DCO based upon our sequence of random bits.[5]

This generation of random numbers has not been officially certified as having sufficient entropy for cryptographic use, but it satisfies all NIST tests. We assume that it generates sufficient entropy, but if further research shows this is not true another method could be used.

```
1  Set up the clock
2  Clear R12, R13
3  for i = 16, ..., 1
4       Clear R11
5       for j = 5, ..., 1
6            C = Output from the clock
7            R11 = R11 + 0x0001 & C
8       for j = 1, ..., 15
9            R12(j) = R12(j + 1)
10      if R11 ≥ 3
11           R12(16) = 1
12      else
13           R12(16) = 0
14      Change the clock settings based on R12
15 Return R12
```

Here, R_m, $m = 1, ..., 16$ refers to the m^{th} register, and $R_m(j)$, $j = 1, ..., 16$ refers to the j^{th} bit of R_m, starting at the least significant bit.

Figure 4: Pseudocode for the Generation of Random Numbers

3.2 AES256 Key Generation

The AES256 key must have 224 randomly generated bits, and 32 parity bits. We generate the key by sampling 256 bits from our random number generator. We then modify every eighth bit based on the sum of the preceding seven bits to ensure that the key is transferred without added noise.

3.3 Diffie-Hellman Key Generation

For the Diffie-Hellman key we have p and g, and we need to calculate g^{ab}. For this process we need to generate a by sampling some number of bits from the random number generator and then take g^a. We then send our g^a to the other party and receive their g^b. Once we receive g^b we can then calculate g^{ab}. For these calculations, we require algorithms beyond what the MSP430F2618 offers [4].

3.3.1 Exponentiation

The main operation we need is exponentiation modulo p. We use a modified exponentiation formula based on the binary representation of a to allow for quick computation on a binary machine. We can represent a as $a = \sum_{i=1}^{|a|} a_i 2^{i-1}$, where $|a|$ is the number of binary digits of a and a_i is the ith binary digit of a. This gives that

$$g^a \mod p = g^{\sum_{i=1}^{|a|}(a_i)2^{i-1}} \mod p = \prod_{i=1}^{|a|} \left(g^{2^{i-1}}\right)^{a_i} \mod p.$$

To do this we need to be able to multiply arbitrarily long unsigned integers, and take the modulus of the results of these multiplications.

```
1  g_2 = g
2  r = 1
3  for each bit a_i of a
4       g_2 = g_2 g_2  mod p
5       if  a_i = 1
6            r = r g_2  mod p
```

Figure 5: Pseudocode for the exponential algorithm

3.3.2 Multiplication

We need to be able to multiply unsigned integers with arbitrary numbers of digits, so we use the formula

$$xy = y \sum_{i=1}^{|x|} x_i 2^{i-1} = \sum_{i=1}^{|x|} x_i (y \ll i - 1),$$

where $|x|$ is the number of digits in x and $a \ll b$ represents b arithmetic left shifts of a. Since our numbers are stored in binary, multiplication by 2 is equivalent to a left shift — similarly to how a multiplication by 10 in decimal is equivalent to a left shift in decimal.

```
1  r = 0
2  for each bit x_i of x
3       if  x_i = 1
4            r = r + y ≪ i − 1
```

Figure 6: Pseudocode for the multiplication algorithm

3.3.3 Addition & Subtraction

We need the ability to add and subtract unsigned integers with arbitrary numbers of digits. For addition we simply add each bit together. We include a carry bit so that if the result is 2 it can be added to the next bits. For subtraction we use the formula for unsigned subtraction of binary integers,

$$x - y = x + \neg(y + C) + 1,$$

where C is the carry bit from previous subtractions and we assume $x \geq y$. We apply this to each word, and use our addition formula to avoid issues with signed representations.

```
1  Zero extend y to have the same number of bits as x
2  for each bit x_i of x
3       r_{i+1,i} = x_i + y_i + C
4       C = r_{i+1}
5       r_{i+1} = 0
```

Figure 7: Pseudocode for the addition algorithm

```
1  for each word x_i in x
2       r_i = x_i + ¬(y_i + C) + 1
3       if y_i > x_i
4            C = 1
5       else
6            C = 0
```

Figure 8: Pseudocode for the subtraction algorithm

3.3.4 Modulus

To implement our modulus operation, we use the basic binary long division formula and ignore the quotient.

```
1  r = 0
2  for each bit x_i of x, staring with the most significant
3       r = r ≪ 1
4       r_0 = x_i
5       if r >= y
6            r = r - y
```

Figure 9: Pseudocode for the modulus algorithm, $r = x \mod y$

3.4 Security

For our security constrain, we would like the Diffie-Hellman key to stay secure for a one month attack costing $100,000 of machinery in 2025. This is equivalent to having the discrete logarithm problem be unsolvable in that time. NIST recommendations suggest that p and a should have 2048 bits and 224 bits, respectively, to be computationally unsolvable until 2030 [6]. These constraints are conservative compared to other estimates, and they are for unsolvability until 2030 — smaller values could be used to maintain unsolvability for a month long attack in 2025.

3.5 Time Complexity

3.5.1 AES256 Generation

To generate the AES256 key, we read 16 words from the random generation algorithm and set every eight bit to be a parity check bit. The majority of the time taken to do this will be spent generating the random numbers, so we consider that time. Each run of the random generation takes 1290 clock cycles, assuming there is no data race for usage of the clock. This gives that the entire generation will take 10^{-3} seconds. This allows for significant waits during data races for the generation to fall within the 30 second bound.

3.5.2 Diffie-Hellman Generation

The majority of the time spent generating the Diffie-Hellman key will be in the exponentiation of g^a and $(g^b)^a$. Computing a should take fewer than 10^{-3} seconds, as it is shorter than the AES256 key. There should also be no data races for use of the clock, as the Diffie-Hellman generation is run when starting the microprocessor.

The number of clock cycles it takes for an exponentiation is given by twice the number of commands executed — each command operates on a word, and so takes two clock cycles. We assume $|a|$ and $|p|$ to be the length of a and p in bits and assume the number of bits in each multiplication is g, so that the time taken for an exponentiation is

$$c_{\texttt{exp}} = \frac{|a|}{16}1290 + 12 + \sum_{i=1}^{|a|} 66 + 2\left(\texttt{mult}(g) + \texttt{mod}(g, |p|)\right).$$

We know the time taken for `mult` to be

$$c_{\texttt{mult}} = 14 + g(42 + 32g)$$

and the time taken for `mod` to be

$$c_{\texttt{mod}} = 22 + \frac{g}{16}(82 + 14p) + 94p.$$

This gives that the number of clock cycles per exponentiation is given by

$$c_{\texttt{exp}} = 165132 + 126|a| + \sum_{i=1}^{|a|} 64(g)^2 + \frac{159pg}{4} + \frac{999g}{4}.$$

We use the recommendations from NIST of 2048 bits for p and 224 bits for a and assume that the size of g is on average $p/2$. This gives that

$$c_{\texttt{exp}} = 35.1697 \text{ minutes}.$$

Then, the time for the generation would be approximately 70 minutes. Although this is more than one hour, it is a worst case estimation. If the time constraint of one hour is necessary, we recommend using a p having 1890 binary digits to ensure that each exponentiation takes at most 30 minutes. The security loss from doing this should not affect the solvability of the problem using a $100,000 machine in 2025.

4 Results

Our algorithm for the generation of AES256 keys is able to finish within the time constraint, so that generation is sufficient. We recommend allowing at least 70 minutes for the Diffie-Hellman key to be generated for security lasting through 2030 using NIST recommendations. If the constraint of one hour is necessary, we recommend doing more research into the security of the Diffie-Hellman key with respect to a $100,000 machine in 2025.

5 Bibliography

References

[1] W. Diffie and M. Hellman, "New Directions in Cryptography" in *IEEE Transactions on Information Theory* (Vol. IT-22 No. 6), 1976 [Online]. Available: `http://ee.stanford.edu/~hellman/publications/24.pdf`

[2] "Announcing the Advanced Encryption Standard (AES)" in *Federal Information Processing Standards Publication*, 2001 [Online]. Available: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[3] "National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security information" in *CNSS Policy No. 15, Fact Sheet No. 1* June 2003 [Online]. Available: `http://csrc.nist.gov/groups/ST/toolkit/documents/aes/CNSS15FS.pdf`

[4] "MSP430x2xx Family User's Guide" 2013 [Online]. Available: `http://www.ti.com/lit/ug/slau144j/slau144j.pdf`

[5] L. Westlund, "Random Number Generation Using the MSP430," Texas Instruments, 2006 [Online.]. Available: `http://www.ti.com/mcu/docs/litabsmultiplefilelist.tsp?sectionId=96&tabId=1502&literatureNumber=slaa338&docCategoryId=1&familyId=912`

[6] E. Barker *et al.*, "Recommendation for Key Management - Part 1: General (Revision 3)," National Institute of Standards and Technology, Gaithersburg MD, July 2012 [Online]. Available: `http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf`

[7] Texas Instruments, 2012 [Online]. Available: `http://www.ti.com/lit/ds/symlink/msp430f2618.pdf`

[8] S. Seo, D. Han, S. Hong, "TinyECCK: Efficient Elliptic Curve Cryptography Implementation over $GF(2^m)$ on 8-bit MICAz Mote," Korea University. [Online]. `https://eprint.iacr.org/2008/122.pdf`

A Code

We do not consider specific memory locations for our storage. We also leave out code for the generation of our AES256 keys, code for Diffie-Hellman encryption step, and code for setting up for the exponentiation steps.

A.1 Random Number Generation

Code for the generation of random numbers is written by L. Westlund of Texas Instruments and is available at `http://www.ti.com/mcu/docs/litabsmultiplefilelist.tsp?sectionId=96&tabId=1502&literatureNumber=slaa338&docCategoryId=1&familyId=912` [5].

A.2 Exponentiation

```
1               ; r4 log of g
2               ; r5 length of g
3               ; r8 loc of a
4               ; r9 length of a
5               ; r10 loc of p
6               ; r11 length of p
7   expo
8               push r6
9               push r7
10              push r12
11              push r13
12              move r4, r6
13              clear r13
14  oloop
15              ; We need 16 iterations per word
16              bis 0x01, r13
17              bis 16, r14
18  loop
19              ; set up for multiply
20              push r4
21              push r5
22              push r6
23              push r11
24              mov r6, r4
25              mov r11, r5
26              pop r11
27              pop r6
28              push r7
29              mov r11, r7
30              call mult
31              call modulus
32              pop r7
33              pop r6
34              pop r5
35              pop r4
36              ; Check if we need to multiply into the result
37              cmp @(r8 +16(r9-r13)), r13
38              jne loopend
39              ; multiply result=r6, length r12 and r7,length r13
40              push r4
41              push r5
42              push r7
43              mov r7, r4
44              mov r12, r5
45              mov r13, r7
46              call mult
47              pop r7
48              pop r5
49              pop r4
50  loopend
51              dec r14
52              jnz loop
53              inc r13
54              cmp r13, r9
55              jne oloop
```

A.3 Multiplication

```
1                ; r4 loc of msw of x
2                ; r5 length of x
3                ; r6 loc of msw of y
4                ; r7 length of y
5                ; r8 loc of prime p
6                ; r9 length of prime p
7   mult
8                push r10 ; result
9                push r11 ; length of result
10               bis 0x00, r10
11               clear r11
12               clear r12
13  loop
14               bis 16, r13
15               bis 0x01, r14
16               cmp @(r6 + 16(r7-r12)) r14
17               jeq loopend
18               add r12, r5
19               ; set up for addition for y and result
20               push r13
21               push r14
22               push r15
23               push r16
24               mov r6, r13
25               mov r7, r14
26               mov r10, r15
27               mov r11, r16
28               call addition
29               pop r16
30               pop r15
31               pop r14
32               pop r13
33  loopend
34               inc r12
35               cmp r12, r7
36               jne loop
37
38               pop r11
39               pop r10
```

A.4 Modulus

```
1               ; r4 = loc of MSW g^a
2               ; r5 = length of g^a
3               ; r6 loc of msw of g^b
4               ; r7  length of g^b
5
6   modulus
7               push r5
8               push r8
9               push r9
10              bis 0x00, r8
11              clear r9
12  loop        bis 16, r10
13              bis 0x01, r11
14  loop2       mov r9, r12
15              ; Rotate the result
16  rl          mov C, r13
17              rla @(r8 + 16r12)
18              xor r13, @(r8 + 16r12)
19              dec r12
20              jnz rl
21
22              mov r11, r12
23              and @(r4 + 16r5), r12
24              bis r12, @(r8+16r9)
25
26              push r12
27              push r11
28
29              mov r4, r11
30              mov r6  r12
31
32              call goe
33
34              pop r11
35              pop r12
36              jnz sub
37              rla r11
38              dec r10
39              jnz loop
40              dec r5
41              jnz loop2
42              jump end
43  sub
44              ; set up for subtraction
45              push r9
46              push r10
47              push r11
48              push r12
49              mov r12 r9
50              mov r14, r10  ; r14 is the length of g^ab
51
52              mov r15, r11  ; r15 and r16 hold the value of p
53              move r16, r12 ; and number of words p uses
54
55              call subtraction
56
57              pop r12
58              pop r11
59              pop r10
60              pop r9
61
62  end         pop r9
63              pop r8
64              pop r5
```

A.5 Addition

```
1                ; rx = r13 r#x = r14 ry = r15 r#y = r16
2  addition      push r12
3                push r8
4                ; Ensure that x > y
5                cmp r14, r15
6                jge xLarge
7                mov r14, r12
8                mov r16, r14
9                mov r12, r16
10               mov r13, r12
11               mov r15, r13
12               mov r12, r15
13               clear r12
14
15 xLarge        clrc
16               bis 0x00, r12
17               ; Add the current word with the last carry
18 loop          dadd @(r15 + 16 r12), C
19               ; Add the current words of x and y
20               dadd @(r13+16 r12) @(r13+16 r 12)
21               ; Move to the next word
22               inc r12
23               cmp r12, r14
24               jne loop
25               pop r8
26               pop r12
```

A.6 Subtraction

```
1
2               ; r#y = r12 ry = r11 r#x = r10, rx = r9
3
4   subtraction
5               clrc
6               push r8
7               bis 0x00, r8
8   nextword    adc @(r11 +16 r8)    ; adds carry to ith word of y
9               inv @(r11+16 r8) ; takes the opposite of ith word of y.
10              push r16
11              push r15
12              push r14
13              push r13
14              bis 16, r16
15              bis 16, r14
16              mov @(r9 +16 r8),r13     ; sets up xi for addition
17              mov @(r11 + 16 r8),r15     ; sets up yi for addition
18              call  addition
19              pop r13
20              pop r14
21              pop r15
22              pop r16
23              dadd @(r9 + 16 r8), 0x001 ; adds one more to resulting xi
24              inc r8
25              cmp r8, r10
26              JNE nextWord
27
28              pop r8
```

A.7 Greater than or equal to

```
1               ; r11 loc of x
2               ; r12 loc of y
3               ; r10 length of x
4               ; r9 length of y
5   goe
6               ; Make sure x has the longer length
7               cmp r10, r9
8               jge xgey
9               mov r11, r13
10              mov r12, r11
11              mov r13, r11
12              mov r10, r13
13              mov r9, r10
14              mov r13, r9
15              bis 0x01, C
16              ; Get how much longer x is than y
17  xgey
18              mov r11, r14
19              sub r12, r14
20              bis 0x00, r13
21  loop        ; Make sure the extra bits are 0
22              cmp @(r11+16(R10- r13)), 0x00
23              jeq true
24              inc r13
25              cmp r14, r13
26              jne loop
27              ; Check all the words are bigger
28  loop2
29              bis 0x00, r13
30  loop3       xor @(r11+16r13), @(r12+16r13)
31              jnz different
32              inc r13
33              cmp r13, r11
34              jne loop3
35              bis 0x00, r13
36              jump end
37              ; Go here if the first >= second
38  true
39              inv C
40              bis C, r13
41              jump end
42  different
43              bit @(r11+16r13), 0x80000000
44              bit @(r12+16r13), 0x80000000
45              jeq loop4
46              cmp @(r12+16r13), @(r11+16r13)
47              jl false
48              jgeq true
49  loop4
50              bic @(r11+16r13), 0x80000000
51              bic @(r12 +16r13), 0x80000000
52              cmp @(r11+16r13), @(r12 + 16r13)
53              jqe true
54              inv C
55              bis C, r13
56              jump end
57  end         ; Return r13
```