Applications of Computer Security in Internet of Things

Practical Computer Security

CSCI-GA.3033-020

Project Report and POC

Munir Contractor

Pranit Arora

Aashish Dugar

Table of Contents

Table of Contents **2**

Introduction **2**

Research and Analysis of Security Algorithms **2**

Elliptic Curve Cryptography **3**
Elliptic Curve Diffie Hellman 4

Implementation **5**
Baseline ECDH Algorithm 5
Memory Reduced ECDH Algorithm 5

Source Code **6**

Evaluations and Results **9**

Discussion **10**

Introduction

There are numerous fields in the tech world that have experienced a sudden surge in usage,and of those fields is said to be Internet of Things. With things like smart homes, voice-controlled toasters and self-setting thermostats, IoT has come a long way. By the end 2018, mobile phones are expected to be surpassed in numbers by IoT devices. IoT itself is expected to grow to about 50 billion devices by 2020-21. However this pace of software based growth has not been consistent with the security-based growth.

Due to IoT still being in its nascent stages(the oldest prominent hacks in this field date back to just 2012-13) of dev and implementation, the security aspect has been ignored or underdeveloped to say the least. This gave us an incentive and motivation to carry out our research in the security of IoT devices.

Research and Analysis of Security Algorithms

We primarily began our research in exploring what the current security stack looks like in IoT devices, what the current protocols are and what renders them insecure.

a) The current structure features a 3 layer architecture - perception layer, transportation layer and application layer. In terms of security being implemented for them, conventional cybersecurity measures have used, although there are no previously-created response methods for unknown problems, so even today's common attacks can get through.

b) Some of the protocols most commonly used today in IoTs, such as Constrained Application Protocol(CoAP), XMPP(eXtensible Messaging and Presence Protocol), MQTT haven't been scaled properly to be implemented in IoT devices. Problems such as outsourcing the entire to a third-party source(CoAP), lack of end-to-end encryption and also quality-of-service functionality(XMPP), incompatibility due to heavy algorithms(MQTT) have brought in a need to either alter these algorithms or bring in something more exclusive to IoTs.
There also exists the possibility that Embedded Systems itself may outlive the algorithm life-cycle, again basically rendering it prone to attacks

c) Another reason for the lack of IoT security, as stated before, was its existence in its preliminary stages. Companies already have existing software corrections, such as which device to favor and prioritize while using a Low Power and Lossy Network Routing (RPL algorithm) and to bring the device itself into public usage first before actually further advancing the security aspect of it.

We came to the conclusion that with a minor sacrifice to the the resource pool of the IoT device, we could make the device a lot more secure. With the above points in mind, here are few of the solutions I came across -

a) An ultra-lightweight algorithm, i.e. ,combination of symmetric(Hummingbird-2 etc.) and asymmetric encryption(Passerine etc.). However, not much information was given as to how it fits into the IoT model, sort of closed source.

b) Stream ciphering - Plain text is enciphered entirely with a pseudo-random key stream. This drawback makes them unusable in some communication protocols. Also we didn't really have a straightforward set of data of what an IoT stream looks like.

c) Low Resource based security algorithms. They offer a similar level of security to larger devices, but have a lower resource usage than today's algorithms for provisioning a better level of security.

This brought us to our proposed usage of the Elliptical Curve Cryptography-Diffie Hellman Algorithm(ECDH).

Elliptic Curve Cryptography

During the initial research for the project, three algorithms with low resource usage were identified – Rabin's Scheme, NtruEncrypt and Elliptic Curve Cryptography (ECC). ECC was chosen due its popularity, easy availability of reference materials and existing standard implementations. There are two major types of elliptic curves used for cryptography, prime filed curves and binary field curves. In prime field curves, all elements of the curve are between 0 and *p*-1 where *p* is a large prime number. In binary field curves, all elements have at most *m* bits, and hence are between 0 and 2^m. The prime field curves were picked for this project because the reference implementation used a prime field curve. There is no difference in the cryptographic strength of the two curves, however prime field curve require fewer computations and hence are desirable for IoT applications.

The security of ECC depends on the difficulty of Elliptic Curve Discrete Logarithm Problem. Let P and Q be two points on an elliptic curve such that kP = Q, where k is a scalar. Given P and Q, it is computationally infeasible to obtain k, if k is sufficiently large. k is the discrete logarithm of Q to the base P. Hence the main operation involved in ECC is point multiplication. i.e. multiplication of a scalar k with any point P on the curve to obtain another point Q on the curve.

Elliptic Curve Diffie Hellman

Elliptic Curve Diffie Hellman is a key agreement protocol that allows two parties to establish a shared secret key that can be used for private key algorithms. Both parties exchange some public information and using their own private information calculate a shared secret. This shared secret can then be used as a key to encrypt and decrypt messages. Any third party which doesn't have access to the private details of each party will not be able to calculate the shared secret using the public information.

The pseudo code for ECDH is as follows:

1. End A calculates $P_A = (x_B, y_B) = d_A * G$, where d_A is a randomly generated number, which is the private key of A, and G is the generator point of the curve, which is public knowledge. P_A is the public key of A.
2. Similarly, end B calculates $P_B = (x_B, y_B) = d_B * G$.
3. A send P_A to B and B sends P_B to A.
4. A calculates $d_A * P_B = d_A * d_B * G$ and B calculates $d_B * P_A = d_B * d_A * G$. Thus A and B have successfully shared a secret.

Since it is practically impossible to find the private keys from the public key, the secret cannot be derived by a third party from the public information alone.[1]

[1] http://www.infosecwriters.com/Papers/Anoopms_ECC.pdf

Implementation

Baseline ECDH Algorithm

The baseline code is implemented using OpenSSL. OpenSSL is a widely used library for encrypting data transferred over the network. However, it is primarily designed for desktop and server use, and not IoT use. So, it serves as a good reference for resource utilization for non-IoT security algorithms.

The OpenSSL implementation for ECDH algorithm was based on the code available from the OpenSSL wiki.[2] It uses NID X9.62 Prime 256v1 curve for generating key pairs and has a maximum private key size of 256 bits.

Memory Reduced ECDH Algorithm

The memory reduced ECDH algorithm is a from-scratch implementation of ECDH. It uses gmplib[3] to handle integers with arbitrary precision.

It implements the secp192k1 and secp192r1 curve which were chosen based on the fact that they are simple elliptic curves and require small key size. The curve parameters were obtained from SEC 2[4] and the encoding for the point and scalar string representation was done using the method described in SEC 1[5]. The maximum private key size supported by the curve is 192 bits, which was reduced to 160 bits to further reduce memory requirement for the algorithm without compromising significantly on security.

The modular arithmetic and point operations were implemented using the pseudo-code and algorithms described in Elliptic Curve Cryptography – An Implementation Guide[6] and Elliptic Curve Cryptography Tutorial[7]. These sources only provided the algorithm and tools required for ECDH, not the implementation itself.

The implementation first generates two key pairs, called *alice* and *bob*, and then exchanges their public keys and calculates the secret. It then compares the two secrets and asserts that they are the same. This ensures that the ECDH implementation is correct.

---

[2] https://wiki.openssl.org/index.php/Elliptic_Curve_Diffie_Hellman
[3] https://gmplib.org/
[4] http://www.secg.org/sec2-v2.pdf
[5] http://www.secg.org/sec1-v2.pdf
[6] http://www.infosecwriters.com/Papers/Anoopms_ECC.pdf
[7] https://www.johannes-bauer.com/compsci/ecc/

The random numbers are generated using Linux specific /dev/urandom device, which may not be cryptographically safe and is not portable. In a production quality application, it should be replaced with cryptographically secure random number generator, which is portable. Production quality applications should also implement more curves if possible.

Source Code

The source code for the project is available at https://gitlab.com/mmc691/pcs-project. The benchmark code is in ecdh-openssl.c and the memory reduced code is in ecdh.h and ecdh.c. This report will go over some of the parts of the source code for the memory reduced ECDH algorithm, since the entire source code is too large to attach here.

First, the gen_key_pair function in the ecdh.c file is used to generate a public and private key pair. The private key is a random number of 160 bits, and the public key is obtained by performing a scalar multiplication with the generator point of the curve as described in the pseudo-code in the implementation section. The gen_key_pair function is:

```
/**
 * Generates a new public-private key pair using the specified curve
 */
struct KeyPair *gen_key_pair(enum Curves curve)
{
    struct Curve *ec;
    switch (curve) {
    case SECP_192_R1:
        ec = get_secp192r1_curve();
        break;
    case SECP_192_K1:
    default:
        ec = get_secp192k1_curve();
    }

    size_t len;
    struct KeyPair *key_pair;
    struct Point *public_key;

    key_pair = malloc(sizeof(*key_pair));
    if (key_pair == NULL)
        printf("Failed to allocate memory for key pair");

    size_t bytes = ec->key_size_bits / 8;
    char *buf = calloc(1 + bytes, sizeof(*buf));
    FILE *fp = fopen("/dev/urandom", "r");
    fread(buf, 1, bytes, fp);
    fclose(fp);

    mpz_init(key_pair->private);
    mpz_import(key_pair->private, bytes, 1, sizeof(*buf), 1, 0, buf);

    public_key = scalar_mult(ec->G, key_pair->private, ec);
    key_pair->public = point_to_charp(public_key, &len);
    key_pair->ec = ec;

    free(buf);
    free_point(public_key);
    return key_pair;
}
```

Next the public keys of the two key pairs are exchanged and the secret is calculated using the get_secret function in ecdh.c. The function does another scalar multiplication of the private key of one party with the public key of the second party.

```
/**
 * Calculates the secret from the public key of the peer and the private
 * key of self.
 * The returned string is null terminated but the calculated length
 * excludes the null terminator.
 *
 * key_pair is the public-private key pair of self
 * peer is the public key of the peer
 * *len is the length of the secret
 *
 * Returns a string representing the secret
 */
char *get_secret(struct KeyPair *key_pair, char *peer, size_t *len)
{
    struct Point *peer_point = charp_to_point(peer);
    struct Point *res_point = scalar_mult(peer_point, key_pair->private,
                                   key_pair->ec);
    char *res = point_to_charp(res_point, len);
    free_point(peer_point);
    free_point(res_point);
    return res;
}
```

The main function in ecdh.c drives the code. If you want to see the generated private keys and the secrets, remove the comments from around the printf statements and execute the code. The main function also checks that the secrets generated are equal to make sure the implementation is correct.

```
/**
 * Main function
 * This function runs the ECDH key exchange algorithm and verifies
 * that it was successful.
 */
```

```c
int main(int argc, char *argv[])
{
    struct KeyPair *alice = gen_key_pair(SECP_192_K1);
    struct KeyPair *bob = gen_key_pair(SECP_192_K1);
    assert(alice != NULL && bob != NULL);

    size_t alice_secret_len;
    size_t bob_secret_len;
    /*
    char *alice_private = scalar_to_charp(alice->private, &alice_secret_len);
    char *bob_private = scalar_to_charp(bob->private, &bob_secret_len);
    printf("Alice's private key is %s\n", alice_private);
    printf("Bob's private key is   %s\n", bob_private);
    printf("Alice's public key is %s\n", alice->public);
    printf("Bob's public key is   %s\n", bob->public);
    free(alice_private);
    free(bob_private);
    */
    char *alice_secret = get_secret(alice, bob->public, &alice_secret_len);
    char *bob_secret = get_secret(bob, alice->public, &bob_secret_len);
    assert(alice_secret != NULL && bob_secret != NULL
       && alice_secret_len == bob_secret_len);

    for (int i = 0; i < alice_secret_len; i++)
       assert(alice_secret[i] == bob_secret[i]);
    /*
    printf("Alice's secret is %s\n", alice_secret);
    printf("Bob's secret is   %s\n", bob_secret);
    */
    free_key(alice);
    free_key(bob);
    free(alice_secret);
    free(bob_secret);
    return 0;
}
```

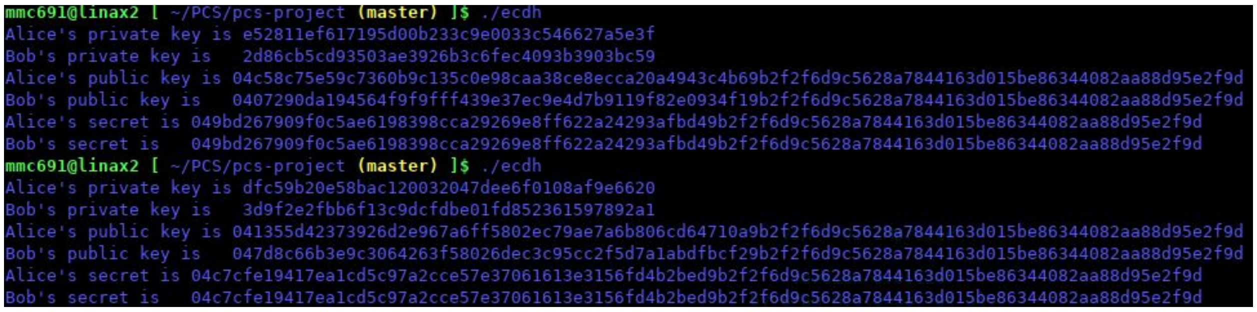
Both the baseline OpenSSL implementation and the memory reduced can be compiled on CIMS server with the following commands:

```
git clone https://gitlab.com/mmc691/pcs-project pcs-project-iot
cd pcs-project-iot
module load gcc-6.2.0
CC=gcc make
```

To run the benchmark code, execute ./ecdh-openssl. The code does not print anything, but runs a similar main function as the one described above.
To run the memory reduced code, execute ./ecdh. To see some output from the memory reduced code, uncomment the printf statements in ecdh.c in the main function, recompile the code and execute.

Evaluations and Results

In order to evaluate the two algorithms, we performed a simple experiment of key generation and exchange in order to form a shared secret. We create two users, Alice and Bob. For both users, we generate a set of a public and a private key. Then, we exchange the public key of the users, and have each user generate a shared secret.
Both users should be able to generate a shared key, which should be identical for both the users. Lastly, the keys are freed from memory. On each call, the keys assigned to both users should be distinct and should be randomly generated.
The first part of evaluating the code was to ensure that the code does what is intended, ie, we are able to generate 2 unique sets of keys, and that we are able to create a shared secret. As the keys and secrets are based on ECC, we can be assured of the security of the algorithm, ie, we can be sure that it is extremely difficult to obtain the private key given a public key and a secret(The details of which are discussed in the previous sections).



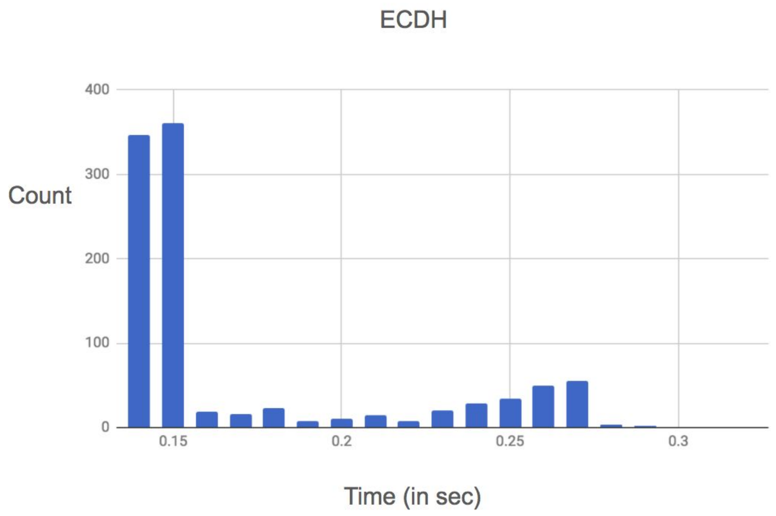
```
mmc691@linax2 [ ~/PCS/pcs-project (master) ]$ ./ecdh
Alice's private key is e52811ef617195d00b233c9e0033c546627a5e3f
Bob's private key is   2d86cb5cd93503ae3926b3c6fec4093b3903bc59
Alice's public key is 04c58c75e59c7360b9c135c0e98caa38ce8ecca20a4943c4b69b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d
Bob's public key is   0407290da194564f9f9fff439e37ec9e4d7b9119f82e0934f19b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d
Alice's secret is 049bd267909f0c5ae6198398cca29269e8ff622a24293afbd49b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d
Bob's secret is   049bd267909f0c5ae6198398cca29269e8ff622a24293afbd49b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d
mmc691@linax2 [ ~/PCS/pcs-project (master) ]$ ./ecdh
Alice's private key is dfc59b20e58bac120032047dee6f0108af9e6620
Bob's private key is   3d9f2e2fbb6f13c9dcfdbe01fd852361597892a1
Alice's public key is 041355d42373926d2e967a6ff5802ec79ae7a6b806cd64710a9b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d
Bob's public key is   047d8c66b3e9c3064263f58026dec3c95cc2f5d7a1abdfbcf29b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d
Alice's secret is 04c7cfe19417ea1cd5c97a2cce57e37061613e3156fd4b2bed9b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d
Bob's secret is   04c7cfe19417ea1cd5c97a2cce57e37061613e3156fd4b2bed9b2f2f6d9c5628a7844163d015be86344082aa88d95e2f9d
```

Figure 1: Image showing verification of successful key exchange to obtain a common secret

The next part of the evaluation is to do the performance evaluation of the two models. We will do the evaluation across 3 key performance factors: Maximum Memory Usage, CPU usage, and Time taken.
In order to remove any bias which may affect a single run of the algorithm, we perform 1000 iterations and then take an average of the data. We do this analysis for both our implementation and for the open-ssl implementation. The results from this analysis are available in the tables below.

Model	Avg Mem(kB)	Avg Time(sec)	CPU usage(%)
ECDH	587.588	0.17058	97.794
ECDH-openSSL	1510.964	0.00639	71.136
Performance Ratio of models	2.571468	1/26.6948	1.3747

Table 1: Relative performance of the two algorithms on the key metrics



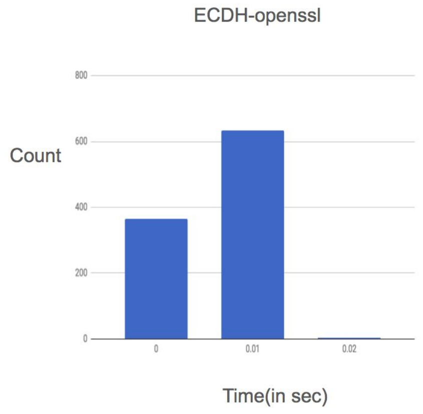


Figure 2: Distribution of time taken to complete 1 iteration of secret exchange

Discussion

There is a significant reduction in the memory consumed by the implementation presented here as compared to the implementation using OpenSSL. Also, since OpenSSL is a shared library, it always occupies memory as long as the device is operational. The memory reduced implementation is application specific and will only occupy memory when executing. Once the private key has been generated it can be saved to a file and read from there to further reduce memory usage. The reduction in power utilization due to less memory required over the lifecycle of the device will outweigh the increased runtime and CPU usage when the device is operated over extended periods of time. Even if the device is not operated for extended periods of time, having a low memory, application specific algorithm allows devices that do not use any encryption to build in a basic encryption algorithm and makes them less vulnerable to attacks.

However, this implementation is only a proof of concept. There may be bugs and other issues in the code that need to be rectified before it can used in actual applications. While a secure coding practice was followed while implementing the code, writing cryptographic algorithms requires experience that we do not claim to have. One known issue the use portability and cryptographic strength of the pseudo random number generator used.

Having said that, as a proof of concept that resource consumption of cryptographic algorithms can be reduced and they can be made suitable enough to run in IoT devices, the implementation achieves what was targeted.