Diffie-Hellman Key Exchange: A Comparison

Sharon Levy and Jenna Cryan
{sharonlevy,jrcryan}@cs.ucsb.edu
Department of Computer Science
University of California Santa Barbara

June 1, 2017

Abstract

The Diffie-Hellman Key Exchange is a Public Key Cryptography protocol that enables users to safely create and transmit encryption keys over the network. This protocol uses modular exponentiation, which can be implemented in several ways. In our paper, we will describe the Diffie-Hellman Key Exchange and implement it on the Google Cloud Platform. The modular exponentiation will be computed using the binary (m-ary), factor, and power tree methods. We will then compare the addition chains for each of these methods and evaluate our results.

1 Introduction

Security remains a critical issue in technology going forward, as the number of Internet connected devices continues to grow and companies increasingly rely on digital infrastructure. The massive amounts of information stored and transmitted everyday must be properly protected against malicious actors. A recent increase in Internet of Things devices has dramatically increased the security vulnerabilities. Already, significant DDoS attacks on IoT connected devices have shown how fragile our Internet infrastructure is by taking down key Internet connectivity points for hours [7], they must reduce the cryptographic overhead to ensure secure communication channels efficiently [4].

Public Key Cryptography has historically been used to help secure exchanges of critical information [10], allowing two parties to securely communicate a shared secret key. As part of the protocol, each party must perform two computations of modular exponentiation. Through a series of improvements, modular exponentiation can become incredibly more efficient in terms of number of multiplications performed [6]. In this paper, we will explore three different ways to perform these computations: Power Tree Method, Factor Method, and Binary (m-ary) Method. We will show that the Power Tree and Factor methods are too inefficient for modern day secure cryptography protocols, which require 2048 bit keys.

2 Diffie-Hellman Key Exchange

To collectively communicate the shared secret between two parties, *e.g.,* Alice and Bob, they must:

- First, agree on a prime p and generator g. These values do not need to remain secret for the protocol to remain secure against outside observers. Although, g must be a primitive element of Z_p^*.[2]
- Alice and Bob select their private keys, such that, $a, b \in Z_p^*$. These values should have lengths proportional to p and g
- Alice computes $g^a \pmod{p}$ and sends the result s to Bob.
- Bob computes $g^b \pmod{p}$ and sends the result r to Alice.
- Using Bob's public key r, Alice computes $r^a \pmod{p} = (g^b)^a \pmod{p}$
- Using Alice's public key s, Bob computes $s^b \pmod{p} = (g^a)^b \pmod{p}$
- At this point, Alice and Bob should have computed the same final value, which is their shared secret key.

Now, even though the eavesdropper may know the prime p, generator g, and even the intermediary keys r and s, it would still be a very difficult problem to backwards compute a or b, which is needed to compute the final secret key. This way, Alice and Bob can use this shared key to, for example, encrypt and decrypt messages to ensure secure communication over a public channel.

2.1 Security

The security behind this protocol relies on using a large enough group order to ensure long enough key lengths, and the Discrete Logarithm Problem. The Discrete Logarithm Problem (DLP) relies on the difficulty in computing the private keys, a or b. It simply states that while it is easy to compute x in $x = g^a \pmod{p}$, given g, a, and p, it is not easy to compute a in $x = g^a \pmod{p}$, given g, x, and p. Essentially, the DLP describes the hardness of solving a one-way function.[9] Given this definition, one way to break Diffie-Hellman is to compute either a or b, since the shared secret key is equal to $g^{ab} \bmod n$. If the attacker can compute one of the two values, he or she can use the public key of the other value as a base to compute the shared secret key.

3 Modular Exponentiation

For Alice and Bob to compute their shared secret key, they must perform 2 iterations of modular exponentiation. Although they use different private keys, the resulting shared key will still result in the same outcome.

3.1 Addition Chains

The number of multiplications involved in modular exponentiation can be represented by an Addition Chain. Each link in an addition chain represents an addition with some previous value in the chain. A variety of methods have been developed to calculate the shortest addition chains, to simplify the calculations involved in modular exponentiation. In the problem of calculating M^d mod n, addition chains can help efficiently calculate the number of multiplications and squarings to generate d. The goal is to achieve the fewest number of multiplications and squarings in order to use the lowest amount of compute power. However, finding the shortest addition chain is an NP-Complete problem. Although it falls within this category, the upper and lower bounds of the length of the shortest addition chain have been computed to be $\lfloor \log_2 d \rfloor$ + $H(d)$ - 1 and $\log_2 d$ + $\log_2 H(d)$ - 2.13, respectively, where $H(d)$ is the Hamming weight of d. The upper limit uses the length of the binary method as a worst case scenario.[6] Here, we focus on 3 methods: Power Tree, Factor, and Binary. We will further explore the last method to evaluate optimization possibilities.

3.2 Power Tree Method

We first implemented the Power Tree method. To find the shortest addition chain, the power tree starts with a single node, 1. From there, the levels are scanned in a breadth-first manner. Additional levels are created by adding together previous nodes together along the same path to generate new (non-duplicate) nodes below. Once fully constructed, each path to a node should be the shortest chain for that exponent value. While this method may be useful for small exponents, its exponential space constraints make it unreasonable to use for large numbers.

```
function Power Tree
input: $M, d, n, tree, q, visited$
output: $S = M^d \bmod n$
1:  while $q$ not empty and $d$ not in $tree$ do
2:      $S \leftarrow q.pop$
3:      if $S$ not in $visited$ then
3a:         $visited.add(S)$
4:          for $node$ in $path$ do
4a:             if $S + node$ not in $tree$ then
4b:                 $path.append(S + node)$
4c:                 $tree.add(S + node)$
4d:                 $q.append(S + node)$
5:  return $S$
```

3.3 Factor Method

The Factor method uses factorization of the exponent to break down the problem recursively until the exponent becomes small enough that the Power Tree method can then be utilized. However, this method relies on finding the smallest prime factor of the exponent at each iteration, which becomes too costly for large numbers.[8]

```
function Factor
input: $M, d, n$
output: $S = M^d \bmod n$
1:   if $d = 1$ then return $M \bmod n$
2:   else if $d = 2$ then return $M^2 \bmod n$
3:   else if $d$ is prime then return $factor(M, d-1, n) * M \bmod n$
4:   else
4a:      $x$ = smallest prime factor of $d$
4b:      $y = d \; / \; x$
4c:      return $factor(factor(M, x, n), y, n) \bmod n$
```

3.4 Binary Method

The Binary method iterates over each bit of the exponent and squares the value, until all bits have been iterated over. This method performs reasonably well for larger numbers, but the addition chains are non-optimal compared to those found through the Power Tree or Factor methods. The number of multiplications in the worst case for this method is $2\lfloor \log d \rfloor$, where d is the exponent to be computed.[3]

```
function Binary
input: $M, d, n$
output: $S = M^d \bmod n$
1:   if $d_{k-1} = 1$ then $S \leftarrow M$ else $S \leftarrow 1$
2:   for $i = k-2$ downto $0$
2a:      $S \leftarrow S \cdot S \pmod{n}$
2b:      if $d_i = 1$ then $S \leftarrow S \cdot M \pmod{n}$
3:   return $S$
```

3.5 m-ary Method

The Binary method can be further generalized to scan multiple bits at a time. While this increases the preprocessing requirements, it may reduce the length of the addition chains for larger numbers. The worst case number of multiplications is 2^k - 2 + $(1 + 1/k)\lfloor \log d \rfloor$, where $m = 2^k$.[3]

```
function $m$-ary
input: $M, d, n$
output: $S = M^d \bmod n$
1:   Compute and store $M^w \pmod{n}$ for all $w = 2, 3, 4, ..., m-1$.
2:   Decompose $d$ into $r$-bit words $F_i$ for $i = 0, 1, 2, ..., k-1$.
3:   $S \leftarrow M^{F_{k-1}} \pmod{n}$
```

```
4:    for $i = k - 2$ downto $0$
4a:       $S \leftarrow S^{2^r} \pmod{n}$
4b:       if $F_i \neq 0$ then $S \leftarrow S \cdot M^{F_i} \pmod{n}$
5:    return $S$
```

3.5.1 Quaternary Method

When scanning 2 bits at a time, we call this the Quaternary Method of exponentiation. This method improves upon the Binary method, but remains lacking in finding an optimal solution.

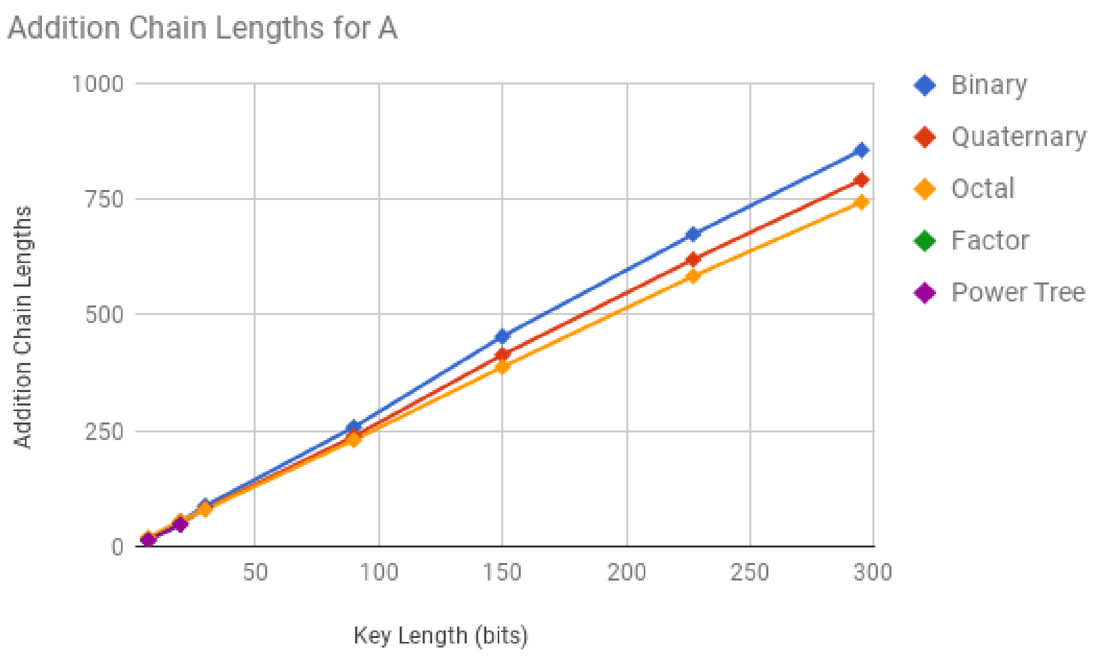
3.5.2 Octal Method

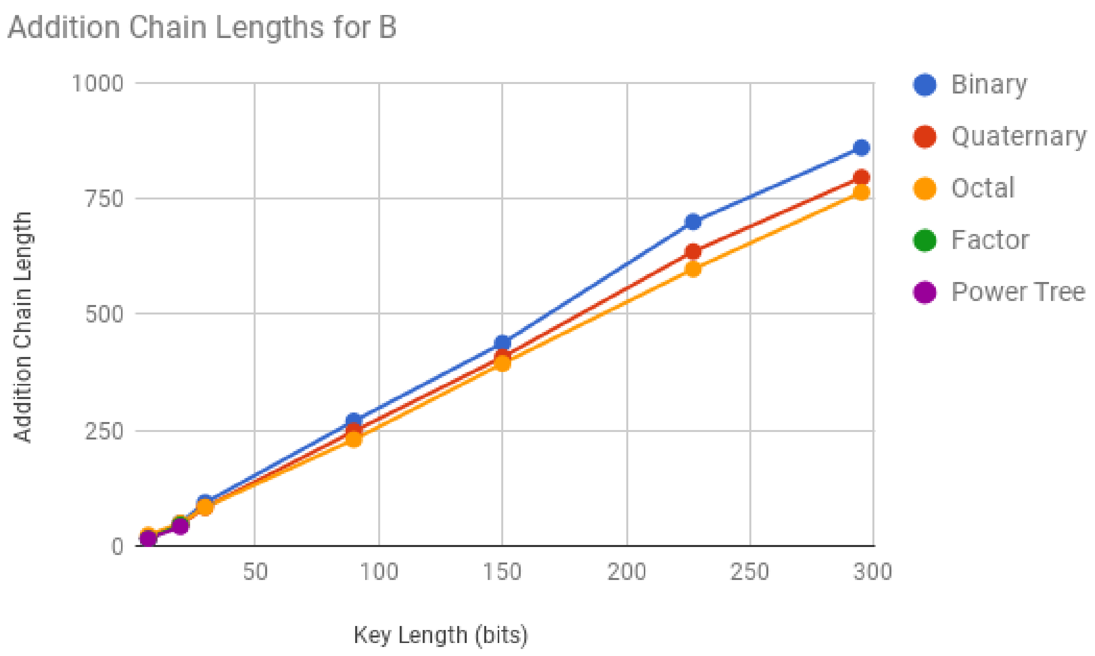
Similar to the Quaternary Method, the Octal Method scans 3 bits at a time. While this also improves further upon the Quaternary method, the shortest chains could still be improved.

4 Implementation

To evaluate these different methods, we tested on varying length keys, until the computations became too costly. We tested using virtual machines on Google Cloud Platform to use the most computing power possible. In particular, we chose a machine with high CPU tolerance to perform the computations as fast as possible. The machine we used ran CentOS on 16 virtual CPUs, with 14.4GB of memory. However, even with a high CPU machine running only this code, we still ran into bottlenecks as the numbers became large (> 30 bits). We implemented the methods using Python, which does not parallelize efficiently, so the multiple CPUs did not help and the bottlenecks grew exponentially. Specifically, the major bottleneck we found related to needing to iterate through very large numbers to find the smallest prime factor. For large numbers, this operation became too time-consuming. For the sake of space, we have omitted the raw data used for testing, but each method was run on the same combinations of g, p, a, and b. The graphs below show our results for each method. Specifically, the graphs compare key length (in bits) to the shortest addition chain we found.

4.1 Results





As seen in the graphs above, the Binary Method performs the worst for the five largest sets of exponents a and b. Meanwhile, Octal and Quaternary started off with larger addition chains in the smallest set and ended up with smaller addition chains in the later sets. The Factor and Power Tree Methods were not able to compute addition chains for the larger sets of a and b, and this will be discussed in the section below. It is clear that the Octal Method creates the shortest addition chains. This is due to the preprocessing that occurs in the beginning of the method, so that M^0 - M^7 are already calculated when iterating over the blocks of bits instead of iterating bit by bit.

4.2 Limitations

We were able to run the Diffie-Hellman Key Exchange on all values of a and b for each of the m-ary methods. However, the Power Tree method was only able to compute the public keys for the first two sets of a and b. Since the Power Tree method continuously adds numbers to the tree, it essentially calculates addition chains for all these numbers. When calculating an addition chain for a large number using this method, it must first process the addition chains for all of the numbers that come before it in the tree. Doing this requires a large amount of computing power and thus, we were not able to complete this for the last five sets of exponents in our table.

The Factor Method was able to compute the first three sets of a and b, as opposed to only the first two for Power Tree. The problem that arose when using the Factor Method was that it called two functions, one to find whether the current exponent is a prime number and another to find the current exponent's smallest prime factor. In our implementation, we determined whether a number was prime by iterating through all the numbers up to its square root and dividing those numbers to see if they were factors of the exponent. The same was done to determine the smallest prime factor of a number. When these functions were called with large exponents, they took a long time to iterate over all the numbers and we were not able to get results for the last 4 sets of a and b. Being able to solve the smallest prime factor function efficiently was not possible as it would lead to solving the prime factorization problem, which is what the Public Key protocol RSA is based on.[8]

5 Conclusion

While most public key encryption nowadays uses RSA because of known security vulnerabilities [1], Diffie-Hellman remains an important protocol for sharing a private key over public channels. As encryption cracking algorithms continue to advance, efficient methods for computing modular exponentiation must progress as well. Modern standards require the order of the group to be at least 2048 bits to maintain security [5]. As we have shown, not all methods for exponentiation could come remotely close to effectively computing with values so large, which puts security in serious question.

The results of our implementation show that Factor and Power Tree methods are useful for modular exponentiation with small exponents. When using large exponents, as in the case of secure Diffie-Hellman, it is better to use the Octal or even Quaternary Method to calculate the public and private keys.

References

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17. ACM, 2015.

[2] Johannes Buchmann. *Introduction to Cryptography.* Springer-Verlag New York, 2004.

[3] Daniel M. Gordon. A survey of fast exponentiation methods. Technical report, Center for Communications Research, 1997.

[4] Qi Jing, Athanasios V Vasilakos, Jiafu Wan, Jingwei Lu, and Dechao Qiu. Security of the internet of things: perspectives and challenges. *Wireless Networks*, 20(8):2481–2501, 2014.

[5] Tero Kivinen. More modular exponential (modp) diffie-hellman groups for internet key exchange (ike). 2003.

[6] Cetin Kaya Koc. High-speed rsa implementation. Technical report, Technical Report, RSA Laboratories, 1994.

[7] Yungee Lee, Wangkwang Lee, Giwon Shin, and Kyungbaek Kim. Assessing the impact of dos attacks on iot gateway. In *Advanced Multimedia and Ubiquitous Engineering*, pages 252–257. Springer, 2017.

[8] Martin Otto. Brauer addition-subtraction chains. Technical report, Universitat Paderborn, 2001.

[9] C. Pomerance and S. Goldwasser. *Cryptology and Computational Number Theory.* American Mathematical Society, 1989.

[10] Arto Salomaa. *Public-key cryptography.* Springer Science & Business Media, 2013.